# Analysis and Improvement of the Variable Binding Scheme in LOGFLOW*

Zsolt Németh Péter Kacsuk

KFKI Research Institute for Measurement and Computing Techniques
H-1525 Budapest, P.O. Box 49.
Phone: (36 1) 395 9220, Fax: (36 1) 395 9027
{zsnemeth, kacsuk}@sunserv.kfki.hu

**Abstract**
*The handling of variables is a crucial issue in designing a parallel Prolog system. The so called closed binding environment is an excellent candidate for a binding scheme in a distributed system, however, it has several weaknesses when the computation is not migrating. In this paper the closed binding method is analysed, a possible modification is introduced and the implementation and performance measurement of the hybrid binding scheme in LOGFLOW is described.*
**Keywords:** distributed Prolog implementation, binding schemes, closed binding environment, performance evaluation

## 1 Introduction

The central question at implementing an OR-parallel Prolog system is how the multiple bindings can be handled consistently in such a way that the resulting binding scheme accomplishes most of these issues: variable access is in constant time, task switching is in constant time, task creation is in constant time. In [GuJa90] it was shown that all the three conditions can not be satisfied. Additionally, low memory consumption must be taken into consideration at designing a binding scheme.

There are a number of techniques for treating multiple bindings [KacsWise92][ChassCod94]. In common, they intended to assure a binding environment so that each worker (process) could handle the shared data as if the worker was traversing the search graph alone. All these techniques have a common feature: references (either physical or logical) are allowed from an environment to any other environments.

In a distributed memory system (e.g. [MarCun96]) these environments may reside on different processing elements, and dereferencing a variable through several processing elements is a tremendously expensive operation. From one point of view the recomputation model aimed at implementing Prolog in a distributed environment. The other method proposed so far for distributed systems is the closed binding environment.

A closed environment in Conery's definition is a set of frames E so that no pointers or links originating from E reference to slots in frames that are not members of E. A closed environment containing just one frame is called a closed frame [Conery87]. In another way the environment holds all the necessary information to continue the computation at a given point. It has two special procedures to close an environment with respect to another one: the (two-phase) unification and the back-unification. The details of closed binding environments are in [Conery87] [Kacsuk92]. The most important features of the unification and back-unification in the LOGFLOW[1] are described in [Kacsuk92].

The LOGFLOW is a distributed-memory implementation of Prolog [Kacsuk94]. Its abstract execution model is the logicflow model based on dataflow principles [Kacsuk97]. Prolog programs are transformed into the so called Dataflow Search Graph (DSG.) Nodes in this graph represent Prolog procedures, e.g. unification (Unify), handling the alternatives (Or), goals (And), facts (Unit). The execution is governed by token streams.

---

* The work presented in this paper was supported partially by the Hungarian-Portuguese joint project "PAR-PROLOPPE" registered under No. H-9305-02/1095 and by two National Science Research Fund (OTKA) projects under grant number T016091 and T022106.
[1]There are slight differences between [Conery87] and the closed environment scheme in LOGFLOW.

Tokens represent a given state of the machine where the computation is continued. There are different tokens according to the different roles they play: representing a query (Do), representing an answer: solution (Succ) or failure (Fail).

Prolog programs are executed in LOGFLOW by the so called Distributed Data Driven Prolog Abstract Machine (3DPAM.) The machine takes a token from the waiting queue, then executes a portion of code represented by the token and handles the tokens resulted from the previous action. In other words, tokens are moved across the DSG by the engine while it executes the appropriate procedures represented by the traversed nodes. For the sake of granularity control, 3DPAM machines are extended with conventional, sequential WAM machines. Thus, LOGFLOW forms logically a two-layered architecture, where the lower level consists of 3DPAM machines and handles fine grained tasks and communication while the upper level consists of WAM machines and executes coarse-grained tasks sequentially. A given predicate can be annotated parallel or sequential by the programmer and the type of a call is controlled dynamically by the scheduler.

LOGFLOW is aimed at exploiting OR- and pipeline AND-parallelism. Although the latter resembles AND-parallelism by its name, the source of pipeline AND-parallelism hides in the alternatives, i.e. the pipeline AND-parallelism is a very special way of OR-parallelism.

LOGFLOW is based entirely on the closed binding environment scheme. Thus, tokens holding an environment are allowed to be distributed over the processor space freely, since the piece of work represented by the token can be executed using the information held by the token only.

However, the implementation showed some of the drawbacks of the closed binding scheme. After analysing the possible alternatives, the Quasi Closed Environment (QCE) idea [KimGaudiot94] seemed to be the closest to solving the problems in LOGFLOW. A new, combined binding scheme was elaborated for LOGFLOW that relies on the ideas of the QCE. However, its technical details, some crucial points, e.g. migration between the processing elements or the handling of hash tables are new and tailored especially for the LOGFLOW.

In Section 2 the overheads of the closed environment are summarised then in Section 3 some of the existing variations are introduced. In Section 4 the design and implementation of the new binding scheme in LOGFLOW are described then in Section 5 its performance is analysed.

## 2 The overhead in closed binding environments

The environment closing is a costly operation both in time and in memory consumption. In [KimGaudiot94] the sources of overhead were investigated and divided into three groups as: overhead from environment scanning, overhead from environment extension and overhead from structure copying. Comparing these overheads to each other concludes that the main sources of overhead are: scanning the compound terms and copying the compound terms.

### 2.1 The scanning overhead in LOGFLOW

Compound terms must be scanned each time when a unification or back-unification takes place. Scanning a compound term means to check each argument of the term whether it is a reference to a variable or not and perform the necessary actions. Since compound terms (structures and lists) are the natural data structures in logic programming languages, they are used extensively. Furthermore, compound terms can have a number of arguments, and can embed other compound terms.

The scanning has three goals: changing the tags from working environment to the reference environment or vice versa, extending the environment whenever the structure refers to a variable that is not in the working environment and finally, instantiating the variables that became bound during the last unification.

### 2.2 The copying overhead in LOGFLOW

Arguments in structures may be changed by bounding variables or by modifying a reference item during the environment closing. As a consequence, structures must not be shared by different environments. If a token is duplicated (e.g. in Or nodes each incoming token is duplicated; in Unit nodes an incoming token may produce several answer tokens), all the structures in its environment must be copied for each new environment. This results in an enormous memory consumption and slows down the computation.

In LOGFLOW three kinds of optimization has been implemented in order to reduce the structure copying. Structures, which contain atoms and constants only in their arguments (or references to structures of such type)

are called Ground terms. These structures are detected at compile time, stored in a special table and handled by dedicated instructions. They never need to be copied or traced, even their unification is much simpler.

Obviously, the reason why structures must be copied is that they contain unbound variables or references. As a second optimization, when all these variables become bound, the structure needs not be copied again, and may be shared by several environments.

Third, although logically it does not cause any problem when a value is reached via references, because of the optimization of structure copying, values are instantiated into the structures when the corresponding variable becomes bound.

In summary, the copying overhead in LOGFLOW is strongly reduced however, it still remains a critical problem regarding the memory consumption.

## 3 The proposed solutions

The previously listed overheads may occur because of the very special handling of addresses in the closed binding environment scheme. While in conventional sequential implementations (and in multisequential parallel implementations as well) a variable has an absolute address, in the closed environment scheme a variable can be addressed by a frame address and a relative offset within the frame. Since a process can see at most two environments, the frame address is coded into the type of reference, i.e. the referred item is in the working or reference frame. Variables are never kept on the heap. Whenever a compound term contains a variable, it is placed into the environment and the compound term contains a reference to it.

Each time a unification or back-unification is performed and a compound term is passed between the two environments, the frame address must be changed in each reference within its arguments (since the referred item is moving from reference to working frame.) Additionally, if there has been an extension of the environment, the offset value of the appropriate references must be changed as well.

However, the closing procedure is really necessary only when the computation is migrated to a different processing element. When it remains on the same processor, the complete closure is superfluous.

The proposed solutions so far try to combine the local addressing scheme of the closed environment with the global addressing scheme of non-closed environments. In [KalRam92] the molecules are introduced to avoid copying when all the arguments of a structure became ground. In [SasPat91] the so called tagged variable scheme is used which is a variation of hash tables for dataflow machines. In [KimGaudiot94] variables are divided into two groups whether they can occur in a structure or not. Variables occurring in a structure are stored globally whereas others are stored still locally. In such a way, the most expensive part of the closing procedure, the structure scanning and copying, can be discarded, more precisely, they can be postponed until the task is really migrated between processors.

## 4 Combining the closed and non-closed binding schemes in LOGFLOW

The basic idea of avoiding the costly operations with structures is the division of variables into two groups according to their occurrence. By default, variables are handled in the usual way according to the closed binding environment principles. Variables are made *global* and thus, are handled differently when they are in the arguments of a structure (hence they are called global variables). The fundamental elements of the new binding scheme are the followings:

1. The special storage of global variables in such a way that the variable access and the duplication of environment can be done efficiently
2. The way how the global variables are detected, created and named according to a global naming scheme
3. Binding the global variables to a value
4. Handling different OR-branches (the question of conditional global variables)
5. The new binding scheme manipulates within a single PE. Some special procedures should be done when the computation migrates between PEs

In the rest of Section 4 these issues are described.

## 4.1 The auxiliary storage: hash tables

There are several possibilities to store the global variables. In [KimGaudiot94] the hash table is proposed due to its efficiency in variable access and handling the alternative branches.

In LOGFLOW the Hash Table (HT) is accessed through a Pointer Array (PA) which can be considered as an access path for different OR-branches. Each token is augmented by an extra field that is used to index in the Pointer Array (PA), called Pointer Array Index (PAI) (Figure 1). The records in the Pointer Array have as many fields as many elements the Hash Table (HT) has. The Pointer Array is used to point to the appropriate lines in the Hash Table, so each field contains a pointer to a line in the Hash Table. In such a way, each token (environment) has different Pointer Array Indices. They correspond to different Pointer Array records, i.e. lines in the Pointer Array thus, they reach different areas of the Hash Table. The columns in the Pointer Array correspond to different hash values. In the example in Figure 1 the variable X has a hash value 0. The first token can access X=5 through the nth row of Pointer Array, while the other token can get X=6 through the n+1st row. The clashing items, i.e. variables whose hash values are the same form a chain. The length of the chain can be influenced by the size of the Hash Table. The Pointer Array is used to save some parameters as well, so it has two extra fields. The role of these fields is explained in Section 4.7.1.

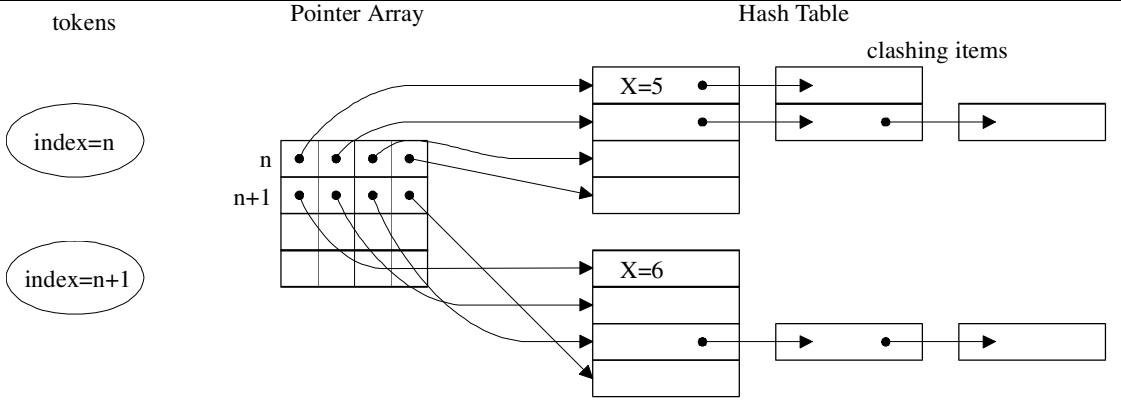


**Figure 1. The usage of hash tables. Two tokens representing different binding environments can bind the same variable to different values by using the hash table. The size of the hash table in this example is 4. The key is that they see different areas of the Hash Table through the Pointer Array.**

The newly introduced Pointer Array and Hash Table are implemented as Dynamically Allocated Static (DAS) [Kacsuk93]. The DAS is much faster[2] than the conventional 'malloc-free' memory management, however, it statically occupies a large amount of memory. Apart from the greater speed this solution can handle the available memory more economically. This technique avoids the loss of memory due to inner fragmentation.

## 4.2 Detecting global variables

Variables can be put into a structure in two cases: whenever a structure is built up as an argument before a call to a predicate or, the structure is built up during a unification when it is in the head of the called clause. Thus, the scope of instructions to be modified is quite small: they are the write_var[3], write_value and the unify_var, unify_value.

The unify_void and write_void instructions represent a special case. Although these instructions exist analogously to WAM, in closed binding environment there are not *real* void variables due to the nature of the closing procedure, because it incorporates all the unbound variables into the environment. In such a way the

[2]The DAS is slightly more than 4 times faster than the conventional method when allocating and reclaiming 50000 data items. On smaller tables the ratio is higher and can reach approximatelly 6. Measurements were done on a T800 transputer and may vary on other systems.

[3] These 3DPAM abstract instructions correspond to set_variable, set_value, set_void, unify_variable, unify_value, unify_void instructions of WAM respectively.

unify_void and write_void instructions must be handled the same way as unify_var, write_var regarding the global variables.

As an illustration here follows the pseudo code of unify_var and write_value.

```
unify_var()
{
  if (mode == READ)
      .../* Do the usual way */
  else
    {
      /* Create a global variable */
      new_var = get_global_name();
      add tag to new_var;
      environment_reg[i]=new_var;
      heap[H] = new_var;
     }
}
```

```
write_value()
{
  dereference(environment_reg[i]...)
  /* dereference returns many important
     parameters about the references as
     well. They are used in the next
     lines*/

  if (dereferred variable is unbound)
    {
     if(dereferred variable is global)
       {
       /* It is a global variable
          already. Put it onto the heap*/
        var = the dereferred variable;
        heap[H] = var;
       }
     else
       {
       /* The dereferred variable is not
          GLOBAL. Since it is put into a
          structure, it must be
          transformed to global */
       /* Create a global variable */
        new_var = get_global_name();
        add tag to new_var;

       /* Transform the environment
          variable into a GLOBAL one */
        if(at least one reference)
         environment_reg[last_reference]=
                           new_var;
        else
          environment_reg[i]=new_var;

        /* Put onto the heap */
        heap[H] = new_var;
       }
    }
  else
    /* environment_reg[i] is a constant
       or referring to a bound variable*/
    heap[H] = environment_reg[i];
}
```

Although, for technical reasons other instructions were necessary to be modified as well, primarily these six instructions are the ones where global variables are created and thus, detected.

## 4.3 Naming global variables

Variables in LOGFLOW are named according to the naming convention of closed binding environments: an environment and an offset in the environment. There may be several variables in the system with the same name, and there is no clashing, since there are two environments visible at a time and the environment names of them are always different.

This is not the case when some variables are global and thus, visible from any environment. These variables must have a unique name for the whole system. The easiest way to generate a new name is assigning a number to each newly generated global variable. The number is coming from a counter, i.e. the successively generated global variables have successive identifiers. It may happen that two variables on two different processing elements have the same identifier, however, as it will be shown in Section 4.7.2, it does not cause any conflict.

## 4.4 Creating global variables

A newly created global variable is unbound, obviously. The Hash Table contains the bound variables only. Interestingly, creating a new global variable is simply an increment of the counter used for generating identifiers and nothing is put into the Hash Table.

## 4.5 Binding global variables

Only the bound global variables are in the Hash Table, unbound global variables are never placed into the table. The newly bound global variables are put into the first position of the linked list in the hash table. These important properties of our solution ensure that the problem of multiple bindings of conditional variables can be handled easily.

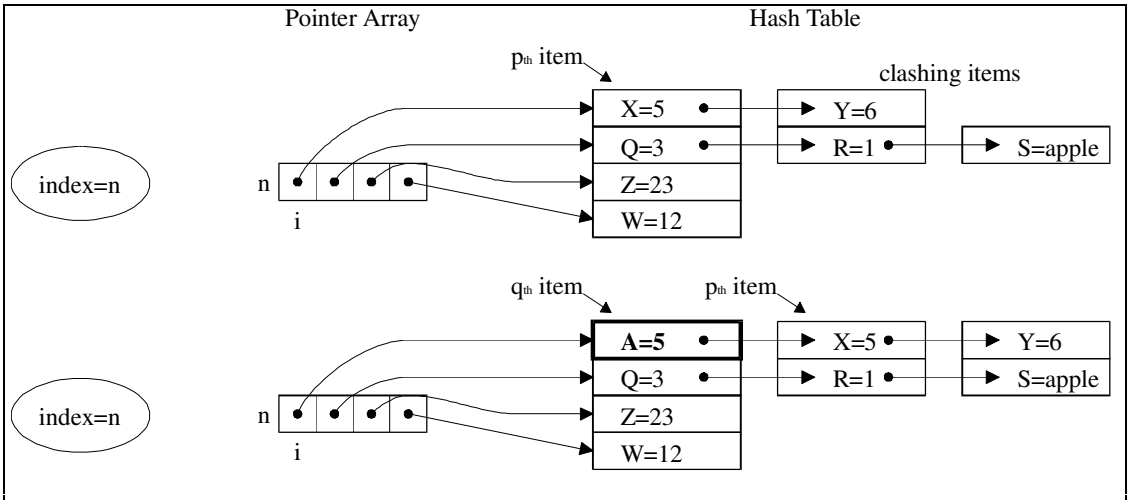


**Figure 2. Binding a global variable.**

First the engine calculates the hash function of the global variable (*i=f(var)*) and allocates a new free item (q) in the Hash Table (Figure 2.) Then it sets the *i*th pointer field in the Pointer Array to point to q, the newly allocated Hash Table item, puts the variable name and value into the q then sets the pointer of q to the one that was previously the first (p).

## 4.6 Handling different OR-branches

It is a central and crucial question in a parallel Prolog implementation how the multiple bindings are handled. When several alternatives of a predicate (several branches in the execution tree) are processed in parallel, variables, which were created in the common parent branch may be bound to different values. To solve the contradiction with the single assignment property of logical variables, these conditional variables are handled in a different way.

The usage of the hash tables for such purposes is an old idea. The proposed usage of hash tables in LOGFLOW is different from Borgwardt's implementation. [Borgwardt84]

The key is in the pointer array that can be considered as an access path to the hash table. In this way different tokens (representing different binding environments) can reach different points in the hash table.

Each time a request token arrives at the incoming arc of the Or node, it is duplicated and put onto the two outgoing arcs. The environment of the incoming token must be inherited by both of the outgoing tokens, however, from this point they represent different environments.

The solution is to allocate a new item in the Pointer Array and copy the content of the item pointed by the incoming token. At this point there are two identical items. Each outgoing token has one of them. It means that they inherited the environment of the incoming token since, they can reach the same variables.

Recall the two very important rules of handling the global variables:

1. Unbound variables are never put into the Hash Table. It means that there are no unbound variables in the inherited part and thus, the problem of multiple bound variables is eliminated. Unbound variables are detected as they can not be found in the table. The price we must pay for this simple solution is that in some cases a relatively long pointer chain must be traversed to check whether a variable is unbound. There are optimizations for this problem in the later sections.
2. The newly bound variables are put into the first position in the chain of clashing elements, i.e. the element in the Pointer Array points directly to the newly bound variable. It means that the newly bound variables are out of the inherited common environment and are visible to that process that has bound them.

In Figure 3 and Figure 4 a simple example can be seen how the alternative branches are treated.

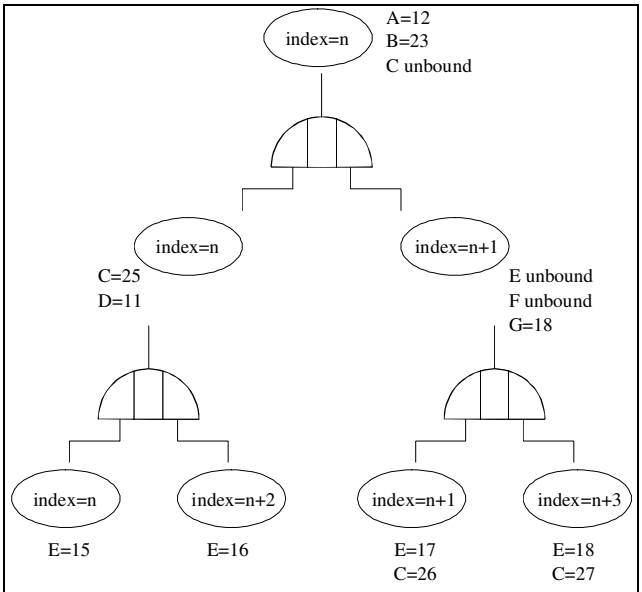


**Figure 3. Tokens on different OR-branches.**

In Figure 3 the first Or node produces 2 request tokens. The tokens have their own environments where A and B are bound while C is unbound. This is the right end of the Hash Table in Figure 4. There are 2 pairs of arrows pointing to the items representing the two environments.

On the left branch in Figure 3 C and D become bound. This is the upper branch in Figure 4. OR-nodes appear in the hash table as multiple pointers to the same elements. Then the execution is split again. One alternative is E=15 the other is E=16. These are the upper branches in Figure 4. This represents the current situation. Indices n and n+2 point to the pointer array selecting the pointers (the 'head') to the hash table. In such a way the first environment has A=12, B=23, C=25, D=11,E=15 while the second has A=12, B=23, C=25, D=11,E=16. The handling of the right branches in Figure 3 is similar.

In this aspect Unit nodes can be regarded as special Or nodes. Unit nodes try to generate all the possible solutions, i.e. the Unit node embodies all the alternatives of a fact. In this way there are (hidden) Or branches and the different solutions must be treated so that they have different binding environments. Therefore, a new Pointer Array item must be allocated for each new solution token.

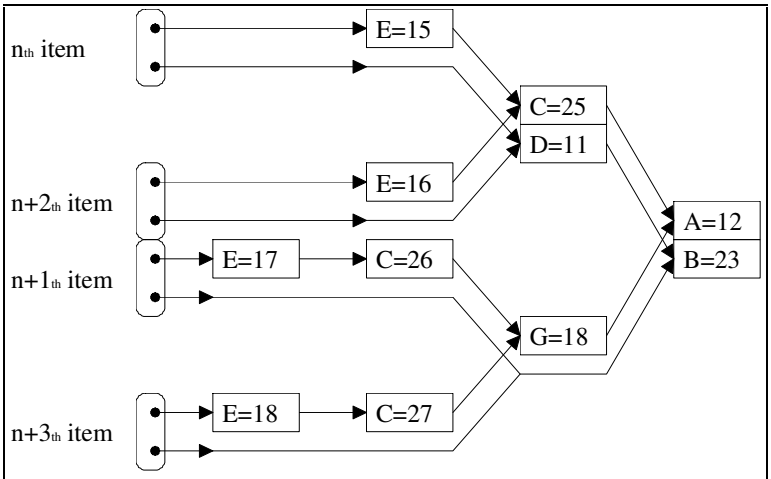


**Figure 4. A snapshot from the Hash Table according to the situation in Figure 3. The size of the hash table in this example is 2. The address of a variable is calculated by taking its alphabetical order mod 2, i.e. the clashing items are A, C, E, etc. and B, D, F, etc. respectively.**

## 4.7 Migration between processing elements

So far all the operations were executed on the same processing element. Since LOGFLOW is implemented in a distributed environment, the new binding scheme must tackle with the arising problems when tokens and thus, binding environments are migrating from one processing element to another.

### *4.7.1 Moving the content of the hash table*

When tokens are transferred between two processing elements, the data packet contains all the token context, its environment, arguments and all the structures that are referred from either the environment or the arguments. On the other site structures are extracted and put onto the heap. Thus, when tokens are transferred

structures must be scanned and copied no matter what the binding scheme is. This operation is used in our solution to encapsulate the necessary information from the Hash Table into the message packet.

There are processes responsible for the inter-processor communication. In case of request tokens the process, handling the output, takes the token and the relevant structures to put them into a data packet and to send to another processing element. In this phase the process tries to get the value for each global variable and substitute the reference by the value in the environment, arguments and structures. In such a way, all the relevant global information is extracted from the hash table and packed into the existing token structure. This packet will be migrated. The hash table does not need to be moved since, the bound variables are within the packet, *unbound variables are not in the hash table.*

When the request token arrives to the remote site, the process, responsible for handling the input, extracts the data packet. The token has no valid index to the pointer array therefore, the process assigns a new item in the array where each pointer point to an empty slot in the hash table. So, whenever the engine, processing this token, tries to get a variable from the hash table it will find nothing, which corresponds to the situation that the newly received token has unbound global variables only. Of course, variables that become bound during the execution on this site will be placed into the hash table.

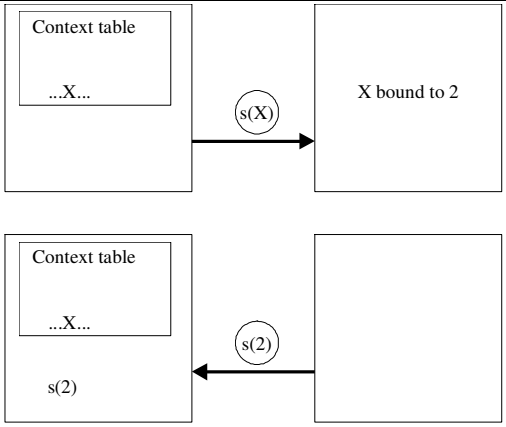


**Figure 5. A problem with reply tokens.**

This basic mechanism introduced above corresponds to the case of request tokens. However, answer tokens require some additional activities:

It may happen that an unbound global variable is stored in a Unify context table[4]. The outgoing request token from the Unify node is processed on a different PE where, let us suppose, the global variable becomes bound. When the answer token is returned to the Unify node, although the bound value is present, the link between the value and its predecessor unbound variable is lost (Figure 5.) Therefore, answer tokens must contain an extract from the Hash Table to update the unbound variables on the original PE. To reduce the amount of information, the current value of global name counter on the originating PE is recorded, and variables below this limit are traced only. The mechanism is introduced in Figure 6.

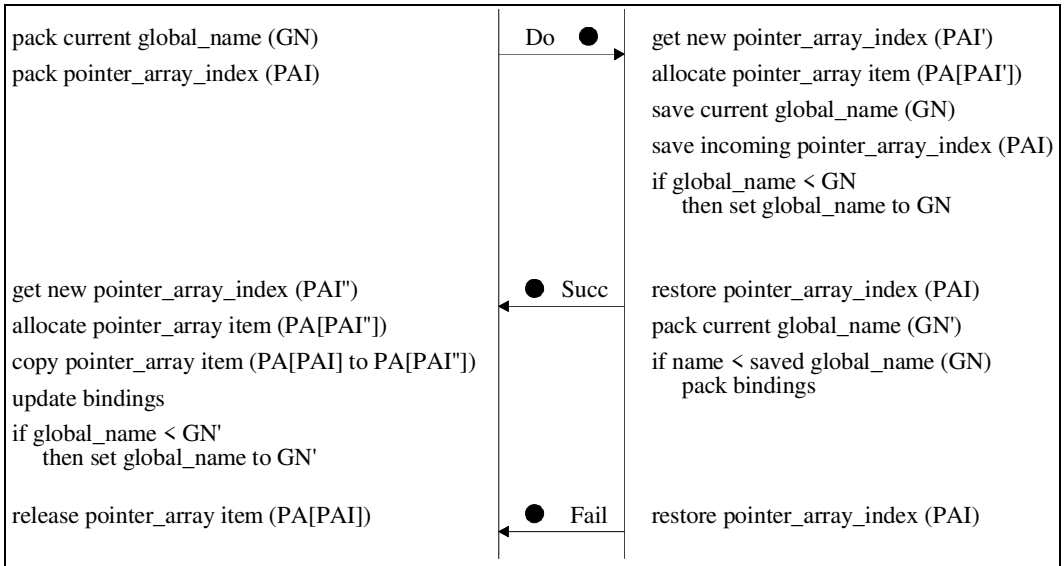


**Figure 6. Token transfer between processors.**

[4]Context tables are used to store a state in the computation. Later the computation can be continued from the saved state.

First the sender PE packs the current value of the global name counter. The Pointer Array Index (PAI) is in the token. On the other PE a new Pointer Array item is allocated to the newly arrived token and the incoming global name value (GN) and PAI is saved to the extra fields introduced earlier. If the global name counter on the given PE is less than the incoming GN, it is set to GN. Otherwise, the engine could create a new global variable that already exists in the newly arrived environment.

When a solution token is going to be sent back, the original Hash Table will be updated by new bindings. Therefore, the saved PAI is restored from the Pointer Array. It is packed into the message as well as the value of the global name counter (GN'). At this point all the variable-value pairs must be included in the message that were bound on this PE, however the variable was created on the other PE. This condition can be detected by comparing the identifier of the variable to the saved GN. If the identifier is less than GN, and a value to the variable can be found in the Hash Table, this pair is going to be sent with the message packet.

On the other side (this is the original PE where the request token was sent from) a new Pointer Array item is allocated and pointers from PA[PAI] are copied. The bindings are updated in this new environment. The copy is needed because there may be several solutions (answer tokens) to a single request. Then the global name counter is set to GN' if necessary. This step is repeated for each solution token.

Finally, a Fail token terminates the solution stream. At this point the original PA[PAI] item can be deallocated.

### *4.7.2 Naming the global variables on different PEs*

There may be existing records in the hash table when the token arrives. Yet, there is no confusion nor discrepancies when the newly arrived token uses the same Hash Table. The key is that the Hash Table is accessed via the Pointer Array, and it is guaranteed that the pointers point to new hash items that are not in any chain. To put in other way, although the tokens representing different computations use the same hash table, they are logically fully separated.

Theoretically, each global variable must have a unique name (identifier) in the whole system. By the introduced method where the identifier is generated by a counter, the condition above is guaranteed within a single processing element only. However, it was shown above that tokens originating from different processing elements (*different token environments*) use completely disjoint clusters in the hash table so, there may be variables with the same identifier (from different processing elements) but it does not lead to any conflict. The reason why global name counters must be set is to avoid to create two identical identifiers in the *same token environment*.

## 4.8 Reclaiming the unused hash items (garbage collection)

In LOGFLOW a token stream is generated as an answer for each request token. The stream is terminated by a Fail token. It can be shown that when the Fail token in the answer stream has arrived to a node, there are no more tokens belonging to the stream in the subgraph [Kacsuk97]. It means that when the Fail token has arrived for the initial query, there are no more tokens in the whole system.

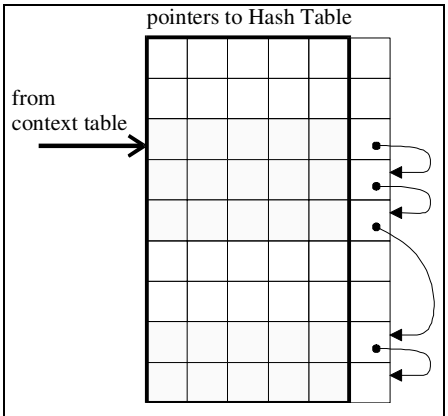


**Figure 7. All the hash table items pointed by any of the pointer array items in the chain can be released.**

Each request token represents a jump to a Unify, Or or Unit node. On a remote site the engine executes the corresponding subgraph and the answer stream appears on the output arc of the node listed above. When the Fail token appears, there is no more token in the subgraph belonging to the same stream. The answer stream must be sent back to the processing element where the request token came from. This implies that the resources used by the tokens involved in the stream can be reclaimed. Token tables and context tables are used in this way. The heap is difficult to be reallocated since the unused structures are scattered over the heap and therefore, a sophisticated garbage collection technique must be introduced [Podh97]. However, the Pointer Array and the Hash Table can be organized in such a way that it supports the reallocation of unused hash items.

The incoming request token first appears on the input of a Unify or Or node. The engine can detect that this is the first appearance of a token from a remote node. Therefore, it records the assigned Pointer Array Index in the context table of the given node. The Pointer Array is augmented with an extra pointer field pointing to the

item, which was allocated at reaching an Or node (Figure 7.) The generated pointer chain represents all the tokens that are derived from the original remote request token. In such a way, all the Pointer Array items that are used in the computation of the given request token can be traced. When the Fail token arrives terminating the computation the hash items can be released by traversing the chain of Pointer Array items starting from the recorded one. Thus, the garbage collection in the hash table can be implemented quite easily.

# 5 Evaluation of the Hybrid Binding Scheme in LOGFLOW

The performance test of the new hybrid binding scheme aimed at investigating how the new binding method supports different types of programs, different sizes of programs and different sizes of transputer networks.

1. First the 3DPAM execution was tested, since the main goal of the new binding scheme is to reduce the cost associated with closing procedures. This cost appears in its most clean form in pure parallel execution (i.e. 3DPAM engines without WAM) on a single processing element.
2. Then the same pure 3DPAM tests were repeated on multiple processing elements where the cost of token transfer (token packing, communication, token unpacking) appears.
3. Since in real applications the 3DPAM engines are supported by WAM engines, the overall performance must be tested in such a case as well.

## 5.1 3DPAM Execution speed

### *5.1.1 Bit-palindromes*

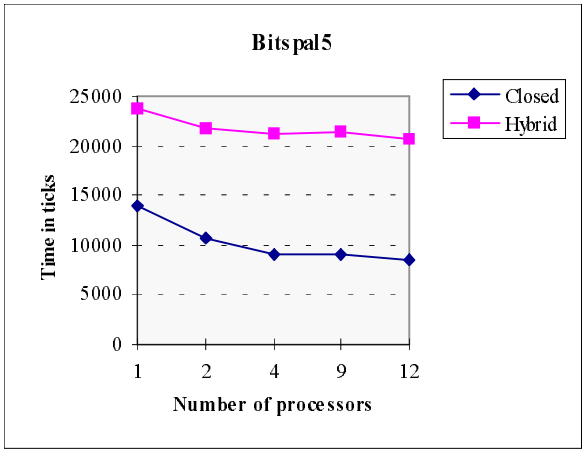


**Figure 8. Execution speed of bitspal5**

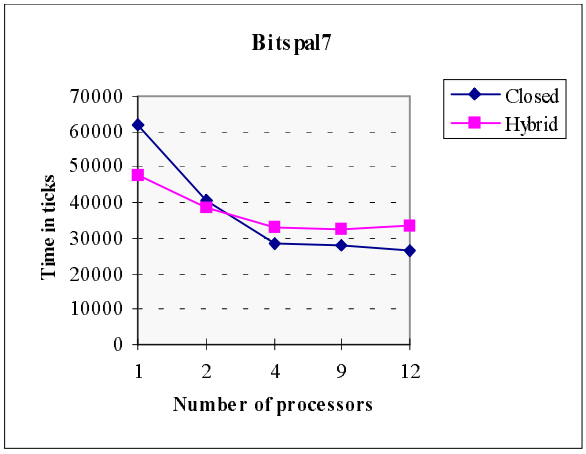


**Figure 9. Execution speed of bitspal7**

Bitspal is a generate-and-test type of Prolog programs. First it generates all the possible bit lists of length 5 or 7, then it tests whether the reverse list coincides with the original one. It achieved a poor speedup in both versions. The reason in case of bitspal5 is that there are at most 32 possible lists to choose from, i.e. the number of tokens in the body of bitspal/1 is relatively small. Although there are more tokens in rev/3 and bits/1, they represent very fine grained tasks and thus, parallelization does not speed up the execution. In case of bitspal7 the number of possible lists is 128, and the tasks in rev/3 and bits/1 have a bit larger granularity (they are proportional to input list length.) These two factors yielded in a slightly better speedup.

The single node execution of Bitspal5 using the hybrid binding method is slower, while that of Bitspal7 is faster than the old version. It proves the anticipated theorem: the bigger the structures are, the more time can be saved by the new binding method. In case of small structures the overhead caused by variable access (handling the hash table and the auxiliary data items) is bigger than the gain of faster unification and back-unification.

The multiple-node tests show, that the cost of token transfer is higher in the new binding scheme than that of the old scheme.

### 5.1.2 *Queens*

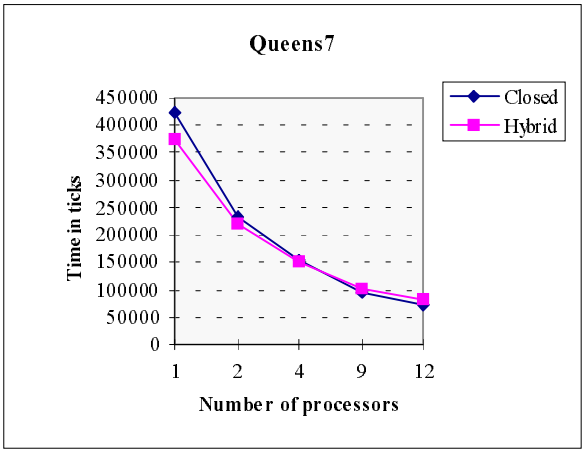


**Figure 10. Execution speed of queen7**

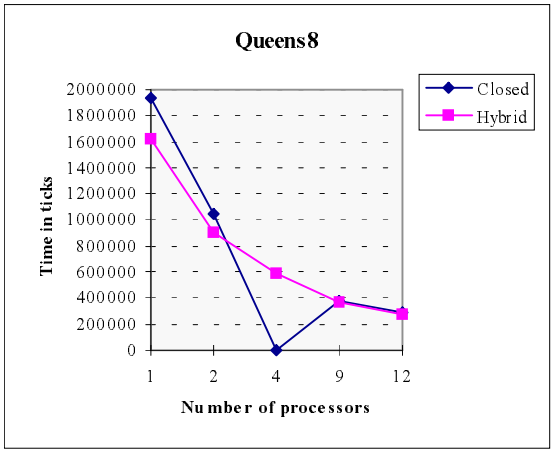


**Figure 11. Execution speed of queen8[5]**

The specimens of the Queen program are the backtracking versions of the many possible implementations of this problem. The single node version of Queen7 repeats the experience of bit-palindromes: a list containing about 7 items is long enough to gain some speedup by faster unification and back-unification. The multiple node versions of Queen8 add even better results: the same gain in case of about 8-long lists is enough to overcome the slower token transfer, although the ratio of the execution speeds of old and new versions is rising as the number of processors is growing.

### 5.1.3 *Hamiltonian path*

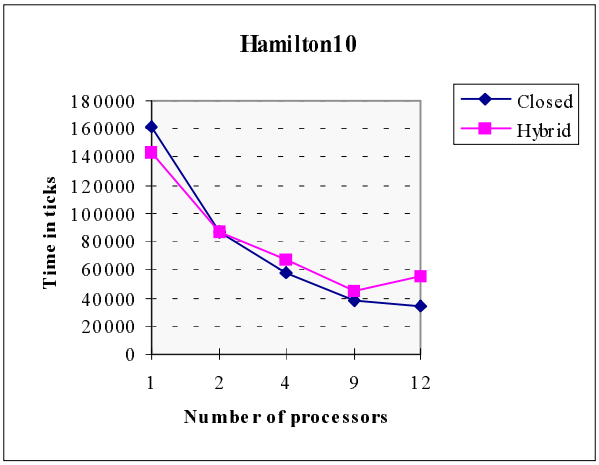


**Figure 12. Execution speed of ham10**

The Hamilton10 is another program, which achieves quite fair speedup, mainly on 2 and 4 processing elements. However, the difference between the old and new binding environments is not as good as one may expect. Although the average length of lists is enough to expect a better performance, most of the structures are ground terms, which are handled uniquely in both binding schemes, and thus, there is not a big difference between them. Hence the slower token transfer determines the execution speed and jeopardizes the small gain in speedup achieved on a single node.

[5] The measured value for closed environment, 4 processors is invalid.

### 5.1.4 List permutation

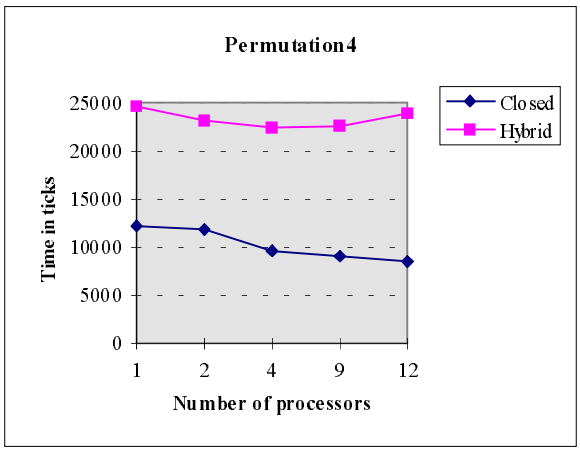


**Figure 13. Execution speed of perm4**

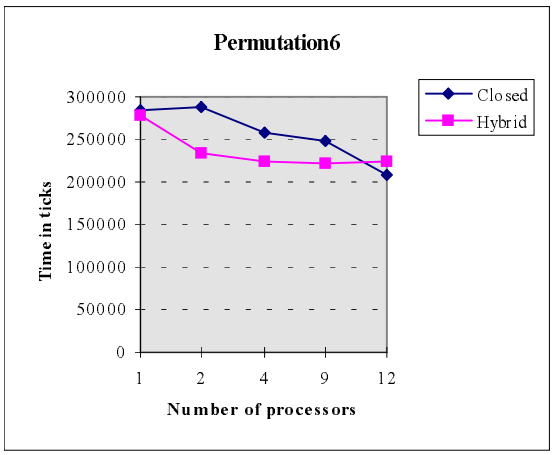


**Figure 14. Execution speed of perm6**

The behaviour of Perm4 and Perm6 shows many similarities with Bitspal5 and Bitspal7. In case of Perm4 the number of solutions is 24, and so, the number of tokens is very small. The maximum 4 items long lists perform very badly in the new binding environment. However, in case of Perm6 the 1-node execution of the hybrid binding method is better than that of the old one, i.e. there are long enough lists to speed up the unifications and back-unifications. Although Perm6 gives better execution speed with the new binding scheme, this is mainly due to the bad performance of the old scheme (see the slow-down at 2-node execution) and not the good behaviour of the new one. This issue is related entirely to scheduling and load balancing.

## 5.2 3DPAM Memory utilization

The other important issue is how the new binding scheme reduces the heap usage. Lists and structures are the natural data representations in declarative languages. They are placed onto the heap. The closed binding scheme requires the copying of these structures quite frequently. The measured values of the heap usage are summarized in the charts below.

### 5.2.1 1 processor execution

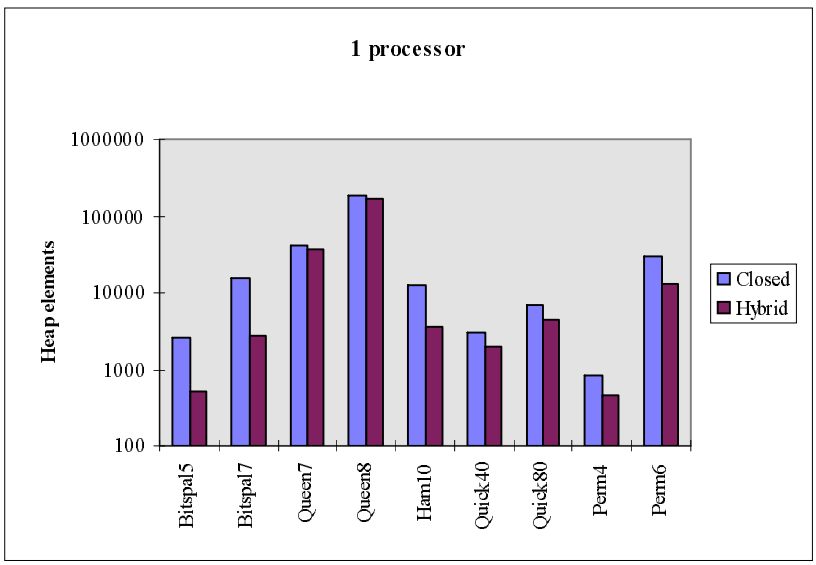


**Figure 15. Heap utilization on 1 processor**

The heap usage drops spectacularly in the hybrid binding scheme in comparison with the closed one. The newly introduced memory areas in the hybrid scheme are: Pointer Array (PA) and Hash Table (HT). The sum of used heap, PA and HT in the hybrid scheme is usually higher than the heap usage in the closed scheme. However,

- the heap copying is optimized and strongly reduced in LOGFLOW
- there are proposed optimizations for efficient use (detecting and reclaiming unused cells) of PA and HT that have not been implemented yet. The DAS (Dynamically Allocated Static [Kacsuk93]) data structure of PA and HT offers an easy way for memory management in these areas.

### *5.2.2 2 processor execution*

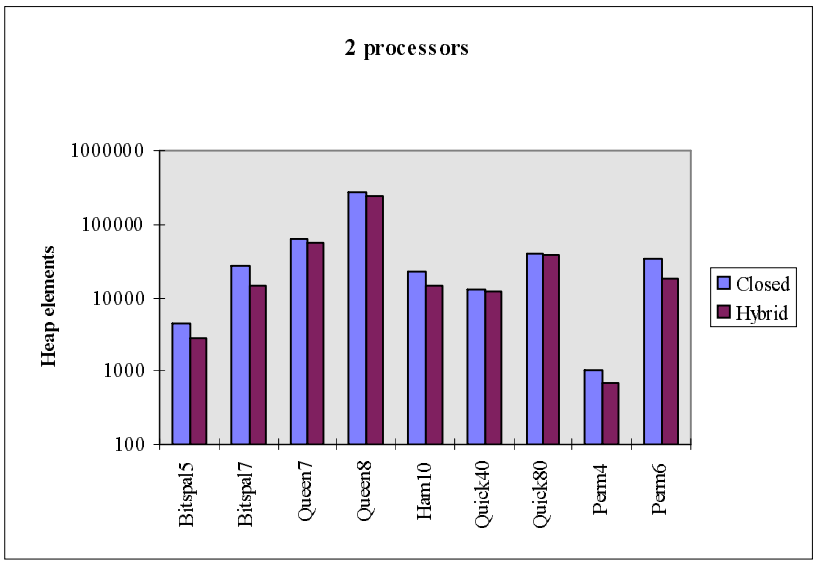


**Figure 16. Heap utilization on 2 processors (the sum of used heap cells is shown)**

Although the heap utilization is still lower, the proportion of the heap usage shows an increasing trend. It means that a large amount of structure copying is related to token transfer and not to environment closing.

Tests on more than 2 processors were not carried out, since the execution is nondeterministic, depends on the scheduling and thus, each running yields a different memory usage pattern, which makes hard to evaluate the binding scheme itself.

## 5.3 3DPAM+WAM execution

The 3DPAM+WAM execution was tested on a single node only, using explicit user notations for parallel/sequential control. The reason is that the scheduling in LOGFLOW is both horizontal (among processing elements) and vertical (between the 3DPAM and the WAM layer.) A scheduling decision is made at run-time, and can change both the processing element and the layer dynamically. In such a way it requires many complex measurements to separate the effect of binding schemes from the effect of scheduling parameters. Therefore, the simplest case was chosen for well tested (regarding the sequential annotations) Prolog programs. In such a way the execution is deterministic and the differences between binding schemes can be observed.

### *5.3.1 Execution speed*

The whole idea of hybrid binding environment tries to treat some weaknesses of closed binding environment (3DPAM) and is not related to WAM [AitKaci91] at all. In such a way, it will not cause great improvement in performance when a big amount of work is done by a WAM engine. Instead, the conversion between the 3DPAM and WAM engines became a slightly more complicated, which can result a small slow-down when many tokens are converted. It can be seen that the Hamiltonian path performs better, where 923 tokens were converted than the Queen7 or Queen8 where many thousand tokens were moved between the two engines. Yet, the effect of hybrid scheme still can be seen when comparing Queen7 and Queen8, as the longer lists behave better yielding a smaller slow-down.

| | WAM tokens | Closed (ticks) | Hybrid (ticks) | |
|---|---|---|---|---|
| Queen7 | 1898 | 108088 | 120108 | 111.12% |
| Queen8 | 7566 | 435536 | 457762 | 105.10% |
| Ham10 | 923 | 80818 | 77461 | 95.85% |

**Figure 17. Execution speed in LOGFLOW (1 PE, static granularity control)**

### *5.3.2 Memory usage*

The heap usage looks similar to the multi-processor 3DPAM execution, since the conversion between the two engines is similar to the token transfer between processing elements, as the structure copying can not be omitted.

| | WAM tokens | Closed (cells) | Hybrid (cells) | | PA used (records) | HT used (records) |
|---|---|---|---|---|---|---|
| Queen7 | 1898 | 12966 | 12126 | 93.52% | 2450 | 2447 |
| Queen8 | 7566 | 53835 | 51627 | 95.90% | 9623 | 9620 |
| Ham10 | 923 | 9852 | 6402 | 64.98% | 7268 | 44 |

**Figure 18. Heap utilization in LOGFLOW (1 PE, static granularity control)**

## 6 Conclusions

The aim of redesigning the binding scheme in LOGFLOW was to reduce the cost of closing procedure and memory consumption. This problem has been addressed by many researchers and there are several solutions. After analysing the overheads, a new binding scheme was designed and implemented for LOGFLOW. This hybrid binding method tries to combine the benefits of closed binding environments and the efficiency of non-closed environment.

The new binding scheme was implemented and tested on a transputer network [NemKa96b]. The model proved to be functionally correct both in single node and multiple node execution.

Performance tests showed that the primary goal, the reduction of closing overhead within a single processing element, was successfully fulfilled. However, the cost of migration between processing elements became higher, thus the overall performance of the system is worse in some cases, depending of the type of problem. The behaviour of the closed and the hybrid binding schemes were tested on several types and sizes of Prolog programs investigating which binding environment fits better one of the types.

When a WAM is added to 3DPAM, most of the work is done by WAM engines, the 3DPAM engines serve as a coordination layer and hence, the binding method has not much influence to the performance. It can be explained, that the connection of the 3DPAM and the WAM is also a combination of a closed and non-closed binding method, and thus, it eliminates some problems of the closed binding scheme. Implementing the hybrid method in 3DPAM and adding a WAM to it is - from one point of view - somewhat two solutions to the same problem.

The test results also suggested that this kind of hybrid binding scheme is an excellent candidate for binding environment on multithreaded architectures like Datarol-II, where the intra-processor overhead associated with closing procedures is reduced by the binding method, whereas the inter-processor copying overhead is eliminated by the facilities of the architecture. This idea is presented in [KaAm97].

It was clearly shown that the new binding scheme achieves better performance when the lists and structures are big enough, e.g. lists of length at least 7 in case of Queen and Bitspal, the Queen8 in case of multiple-node execution. (The difference is even bigger in case of some other test programs not presented in this paper.) It suggests, that in practical applications, where usually bigger structures are used than those in standard benchmarks, a better average performance can be expected.

Furthermore, several possible optimizations were proposed in [NemKa96a] and [NemKa96b] which have not been implemented yet (e.g. hash cache, garbage collection in the auxiliary data storage, etc.) Test results suggest that some of them could further improve the performance of the new binding method.

## 7 Acknowledgments

Authors would gratefully acknowledge the work of Portuguese partners, especially J.C. Cunha, J. Lourenco, P. Medeiros, R. Marques and V. Duarte, in the framework of the project. Tests were realized on a 16-node Parsytec machine made available to us courtesy of Universidad Nova de Lisboa. The WAM compiler and interpreter were made available to us courtesy of A. Gonzalez, J. Tubella and E. Elias at UPC, Barcelona.

## 8 References


[AitKaci91] H. Ait-Kaci: Warren's Abstract Machine. MIT Press, 1991.

[Borgwardt84] P. Borgwardt: Parallel Prolog Using Stack Segments on Shared-Memory Multiprocessors. Proceedings of the 1984 Symposium on Logic Programming.

[Conery87] J.S. Conery: Binding Environments for Parallel Logic Programs in Non-Shared Memory Multiprocessors. Proceedings of the 1987 Symp. on Logic Programming, 1987.

[ChassCod94] J.Chassin de Kergommeaux, P.Codognet: Parallel Logic Programming Systems. ACM Computing Surveys Vol 26. No 3. Sept 1994

[MarCun96] R. Marques and J.C. Cunha, PVM-Prolog: A Prolog interface to PVM, Proc. DAPSYS'96, Miskolc, 1996, 173-181

[GuJa90] G. Gupta, B. Jayarman: On Criteria for OR-Parallel Execution of Logic Programs. Proc. of the 1990 North American Logic Prog. Conf., 1990, pp. 737-756.

[Kacsuk92] P. Kacsuk: Distributed Data Driven Prolog Abstract Machine. In: P. Kacsuk, M.J.Wise: Implementations of Distributed Prolog. Wiley, 1992.

[Kacsuk93] P. Kacsuk: Memory management in LOGFLOW. Microprocessing and Microprogramming 39. (1993) pp. 79-82.

[KacsWise92] P. Kacsuk, M.J.Wise: Implementations of Distributed Prolog. Wiley, 1992.

[Kacsuk94] P. Kacsuk: LOGFLOW: Prolog on Massively Parallel Machines. Invited Paper, 6th International Conference on AI and Information-Control Systems of Robots, Smolenice, 1994. pp. 71-80.

[Kacsuk97] P. Kacsuk: Execution Models for a Massively Parallel Prolog Implementation. Journal of Computers and Artificial Intelligence. Slovak Academy of Sciences, 1997. (accepted for publication)

[KaAm97] P. Kacsuk, M. Amamiya: A Multithreaded Implementation Concept of Prolog for Datarol-II. Proceedings of ISHPC97 (accepted).

[KalRam92] L.V. Kalé, B. Ramkumar: The Reduce-Or Process Model for Parallel Logic Programming on Non-Shared Memory Machines. In: P. Kacsuk, M.J.Wise: Implementations of Distributed Prolog. Wiley, 1992.

[KimGaudiot94] H. Kim, J-L. Gaudiot: A Binding Environment for Processing Logic Programs on Large-Scale Parallel Architectures. Technical Report, University of Southern California 1994.

[NemKa96a] Zs. Németh, P. Kacsuk: Analysis and Improvement of the Variable Binding Scheme in LOGFLOW. Technical report, KFKI-MSZKI 1996.

[NemKa96b] Zs. Németh, P. Kacsuk: Technical Description of Implementing a Hybrid Binding Scheme in LOGFLOW. Technical report, KFKI-MSZKI 1996.

[Podh97] N. Podhorszki: Multi-phase Garbage Collection in LOGFLOW. Accepted for Workshop on Parallelism and Implementation Technologies for (Constraint) Logic Programming Languages, Port Jefferson, 1997.

[SasPat91] A.V.S. Sastry, L.M. Patnaik: OR-Parallel Evaluation of Logic Programs on a Multi-Ring Dataflow Machine. New Generation Computing, 10 (1991), pp. 23-53.