On implementing logic programming languages on a dataflow architecture

Patrick Weemeeuw[1]
Maurice Bruynooghe[2]
Marleen De Hondt[3]

K.U.Leuven
Department of Computer Science
Celestijnenlaan 200 A
B-3030 Heverlee
Belgium

Abstract

An implementation scheme for a logic programming language on the Manchester Dataflow Computer is presented. The Manchester Dataflow Computer is a parallel data-driven computer based on the tagged-token model. The logic programming language is derived from PROLOG, with addition of modes and types. The cut operator has been replaced by guards. The implementation scheme supports OR-parallel evaluation of don't-know and don't-care non determinism.

Introduction

Exploiting parallelism is currently one of the major research goals in computer science. At the hardware level, an important approach is the development of dataflow architectures. A well known example is the Manchester Dataflow Computer [Gurd et al. 85]. At the software level, logic programming is a promising approach. In this paper, we explore how a simple logic programming language can be executed on the Manchester Dataflow Computer. We concentrate on aspects of OR-parallelism.

In section 1, we give a brief survey of the Manchester Dataflow Computer. This is followed by a description of our simple logic programming language in section 2. The implementation scheme is presented in section 3, and some experimental results are discussed in section 4. In section 5, we refer to some related work, and we finish with some discussion in section 6.

1. Survey of the Manchester dataflow computer

We briefly summarize the main features of the Manchester Dataflow Computer, for more details, the reader is referred to [Gurd et al. 85] and [Kirkham 84]. A recent survey of dataflow machine architectures is in [Veen 86].

A dataflow graph is a directed graph; the nodes represent functions and the arcs represent data paths between these functions. A state of the computation is represented by a set of tokens on the arcs of the graph. A node is activated as soon as each input arc contains a token. The active node removes the tokens from its input arcs and puts tokens on (some of) its output arcs.

Due to hardware limitations, a node has one or two input arcs, and zero, one or two output arcs. Multiple copies of a token can be obtained by a tree of DUP nodes: the DUP (DUPlicate) operation produces two copies of its only input.

Conditional computation is possible with the BRR (BRanch on boolean - Repeat) node, which copies its left (or first) input token to its left or right output arc, according to the boolean value of its right (or second) input token.

---

1. Currently supported by CEE Biotechnology Action Programme

2. Supported as research associate by the Belgian National Fund For Scientific Research

3. Currently at BIM, Kwikstraat 4, B 3078 Everberg

The graph structure has only an implicit representation in the hardware, all tokens travel on the same ring structure. Besides a (typed) data value, tokens also contain their destination specifying the address and input port of the node which has to process the token. Arcs are usually static and the destinations are known at compile time, however, to allow e.g. returning the result of a procedure to the caller, one has also dynamic arcs, the destination of tokens on such arcs is only known at run-time. Finally, to allow multiple simultaneous activations of the same node (*reentrant code*), tokens also contain a label (*tag*) consisting of an index, an iteration level and an activation name. A node *fires* only when there are tokens with the same label on each input port. A fatal error, called a token clash occurs when there is more than one token with the same label on some edge of the graph (input port of a node). Graphs which are free from token clashes are called *safe*.

The index differentiates between different parts of the same data structure (e.g. an array); the iteration level differentiates between the (possibly parallel) activations of the body of a loop, and the activation name is intended to differentiate between (possibly parallel) activations of a procedure body.

Iteration level and activation name together form the *colour* of a token; the *context* of a token consists of its destination and its colour. Instructions involving labels or destinations will be introduced when needed.

To introduce the notion of matching functions, we first explain the matching unit. The matching unit is a sort of associative memory, where tokens destinated for a node with two input ports wait for the arrival of a token with a corresponding (*matching*) label. This is the standard use of the matching unit, achieved with the matching function EW (Extract/Wait).

The matching function is also included in the destination field. If the destination is a unary operator, the matching function is always BY (BYpass): the token passes the matching store and the operator is activated. The success action prescribes what happens to the already stored *matching* token before activating the operator, the fail action prescribes what happens to the arriving token when a matching token is not available. So, EW means: if the matching token is in the matching unit, then extract it, otherwise wait (i.e. store the arriving token in the matching unit).

Other matching functions are: ED, DD, ID, DD and PG. We only explain here those that are needed for our purposes.

- PD (Preserve/Defer): preserve as success action leaves a copy of the matching token in the matching unit; defer as fail action does not store the token in the matching unit but put it back on the ring structure for another cycle.
- DD (Decrement/Defer): decrement as success action leaves a copy of the matching token in the matching unit with the value decremented; defer as fail action.
- ID (Increment/Defer): analogous to DD.

Note: it is the matching function of the arriving token that determines the action to be performed.

A dataflow graph is *well formed* if no tokens are left in the matching unit after execution. This is an important property, because the capacity of the matching unit is finite and the performance decreases quickly when overflow occurs.

It is often necessary, for reasons of efficiency, to store some data structures (see [Bowen 81],[Veen 86]). For the time being, this is done in the matching unit by means of special matching functions. A special unit, called the *structure store*, is announced to cope with this needs without burdening the matching unit.

2. Description of the source language

Our source language is derived from PROLOG. The language is different from PROLOG in three major points:

- guards as a means of control instead of cuts;
- modes;
- types.

We do not allow cuts in our source language because the effect of the cut-operator depends on the sequential execution of the program. Instead of this, we introduce don't-care non determinism by means of guards. We distinguish between don't-know and don't-care predicates.

Don't-know procedures do not contain any guards; all clauses can be initiated in parallel, and several clauses may succeed.

A don't-care procedure has in each clause one (possibly empty) guard. The guards of the different clauses can be started in parallel; only one of the clauses with succeeding guard commits and is selected for further execution. A procedure call fails if either all guards fail or the remainder of the body of the commited clause fails. Procedures occurring in a guard are not allowed to produce side-effects; but may bind variables.

To simplify compilation in this first attempt, we have also used type and mode declarations; this avoids real unification. We distinguish between modes in and out: an argument on an in-position must be ground at run-time before executing the call; an argument on an out-position must be free at-run time before executing the call, ground after. This mode restriction is often made and a lot of practical programs obey it. [Drabent 87] has a thorough discussion on this topic. Both types and modes can often be inferred with sufficient precision by abstract interpretation, see a.o. [Bruynooghe et al 87].

Currently we handle only types built from the primitive "integer" and the structure "list".

As an illustration, we show some well known examples:

A. **append**: concatenates two (generic) lists to form a new list.

modes: in/in/out
types: List(T)/List(T)/List(T)

```
append ( L1, L2, L3 ) :-
        L1 = nil | L3 <== L2 ;
        L1 <> nil | L1 ==> ( X . L1' ),
                append( L1', L2, L3' ),
                L3 <== ( X . L3' ) .
```

Notes: - we use the infix notation for lists.
- '<==' means construction, '==>' means selection.
- '=' is a test for equality.
- '|' is the commit sign.

B. **delete**: takes one element (the first argument) out of a list of integers (the second argument) and returns also the remainder of the list(the third argument).

modes: out/in/out
types: Integer/List(Integer)/List(Integer)

```
delete ( E, L, R ) :-
        L <> nil, L ==> ( E . R );
        L <> nil, L ==> ( X . L' ),
                        delete ( E, L', R' ),
                        R <== ( X . R' ).
```

C. **perm**: generate a permutation (second argument) of the list given in the first argument.

modes: in/out
types: List(T)/List(T)

```
perm ( L, PL ) :-
        L = nil | PL <== nil;
        L <> nil | delete ( E, L, R ),
                   perm ( R, PL' ),
                   PL <==  ( E . PL' ).
```

In the remainder of the text, we mean by 'in line code' all code associated with (explicitly written) unification, i.e. tests, selections and constructions.

3. Implementation scheme

In this section we present our implementation scheme. This is done in an incremental way: we start with introducing the chosen data representation. This is followed by a discussion of the implementation of the basic building blocks and an example. Then we discuss the extensions needed to include garbage collection and to abort the guards.

The figures below are drawn according to the following conventions:

— nodes are indicated by boxes; macro-nodes are drawn with double lines at the sides of the box.

— continuous arrows represent static arcs; dashed arrows represent dynamic arcs.

— matching functions, if different from the default values, are written close to the arc they refer to.

— a literal on an input arc is represented between double quotes at a T symbol; a ⊥ represents a cutted output arc (no tokens are produced).

— tokens are indicated with their symbolic name or their type and value followed (if necessary for good understanding) by (a part of) their label between '<' and '>'. E.g. L <col(c)> means the token representing the value of L with the colour c in its label; cxt(dest(d),col(c0)) stands for a context token consisting of the destination d and the colour c0, with concealed label.

— when an input port of a node is explicitly labeled in a figure (because there is a destination token referring to it), then that label is in italics.

— DUP nodes are usually indicated by the splitting of an arc.

3.1 Implementation of data structures

In [Bowen 81] a profound justification of possible implementation schemes can be found.

For objects of a scalar type, the implementation is straightforward: they are represented by a token with an appropriate data value.

The representation of structured objects (only lists in our case) is more complex. An empty list is a scalar object whose data value type is *null*, nonempty lists are represented by a token whose data value type is a *context*-type; the data value itself consists of a context: a colour-destination pair *c-d.r*; the destination *d.r* is always the right input port of a SCD node (Set Colour and Destination, see below). The information stored in the list is represented by three distinct tokens, which have colour *c*, destination *d.l* and which have as index respectively 1, 2 and 3. The first token is a reference count, the data value is an integer, it is used for garbage collection (see below). The second token represents the head, the data value represents either a scalar or another structured object. The third token represents the tail, which is either the empty list or again a structured object. These three tokens reside on the left input port (*d.l*) of a SCD node, which means that they occupy the matching unit, waiting for a matching token to arrive on the other port of the SCD node. (Thus a n-element list occupies 3 n entries in the matching unit, these are the tokens to be moved to the announced "structure store".) One of these tokens can be selected by sending an appropriate token to the other port of the SCD node, this is illustrated in fig. 3.1.

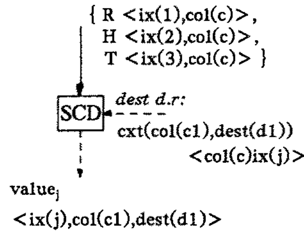


Figure 3.1: Selection of a token residing on a SCD node
SCD node with address **d**; *three tokens representing a list are on the left input port. Arrival of a context token with the same color* **c** *selects the token with index* **j**, *gives it the color* **c1** *and forwards it to destination* **d1** (*value$_j$ is either* R, H *or* T). *If the matching function of the context token is PD, the selection is non destructive.*

3.2 List operators

The macros we use, resemble very much the macros given in [Bowen 81]. The only difference is that we use the information that there are exactly 3 fields needed for a list cell; this allows to optimise the COLLECT-macro described there. For details, we refer to [Bowen 81], we only show a specification of the macro's we will use.

3.2.1 Construction

As a list construction always succeeds, we can use the (slightly modified) STORE-macro of [Bowen 81]. Consequently, ..., L <== (X.L'), ... is represented as shown in fig. 3.2.

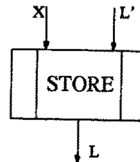


Figure 3.2: List construction X *and* L' *have the same color c; a token* L *with color c is created whose data value is cxt(col(c),dest(d.r)), with d.r the address of the right input port of a SCD node. Also the three list tokens (reference count, head and tail) with colour c and destination d.l are created by the STORE macro.*

3.2.2 Testing for an empty list

Fig. 3.3 shows the graph for this test. Failure of the test must absorb all tokens related to the clause under consideration, so, all these tokens are inputs for the macro BRR. The test is done with the CET (Compare Equal Type) node which compares the type of the data value with the type *null*. The result of the test is fed into a BRR node and decides whether the inputs are forwarded on the F branch or absorbed on the T branch.

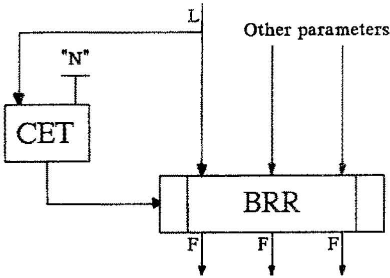


Figure 3.3: Test L <> nil
(type of nil is "N")

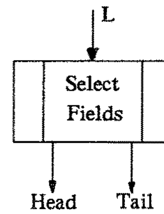


Figure 3.4: Selection L ==> (Head.Tail)

3.2.3 Selection

Selection of the head or the tail of a list is accomplished by sending the index of the corresponding field of the first list cell, together with the context-token representing the list to the SELECT macro as given in [Bowen 81]. For the sake of simplicity, we will always select both the head and the tail, and we represent this by the single macro 'select-fields' as shown in figure 3.4. If the list is possibly empty, a test on nil has to be inserted before the selection.

3.3 Procedure calls

3.3.1 Non tail-recursive calls

To execute a procedure call, all tokens representing input parameters are sent to the body of the procedure; for each output parameter, the destination for that parameter is sent. Some other values are also passed to the procedure; they are explained in fig. 3.5. All tokens passed to the procedure have the same unique colour.

As a procedure may be non deterministic and may produce multiple solutions, all solutions have a distinct colour to distinguish between them. This colour is an extra output parameter of the procedure (the "colour token"). For each solution, a copy in the correct colour has to be produced of all values still alive after the procedure call, because together with the result of the procedure they activate the remainder of the body in that branch. As a result, all this values have to be stored, as can be seen in fig. 3.5. This is the general scheme without garbage collection for non tail-recursive calls.

3.3.2 Tail-recursive calls

Problems arise if, ignoring in-line instructions, the last call of a procedure is a recursive call as in the second alternative of delete. The results of a call delete (E,L',R') at recursion level n are E, R' and a result color ci. These results are fed into the in-line instruction R <== (X.R') of delete at level n-1. This in-line instruction does not change the color, thus we obtain a result at level n-1 having also the color ci. This cannot yield clashes for R' tokens in the instruction R <== X.R' because the level n result with color ci is consumed before the level n-1 result with the same color is produced. However, the result colour ci is produced independently from R and is used to colour the X token required by the instruction R <== X.R'. Thus the result color ci at level n-1 and the X token with color ci at level n-1 can be produced before the X token with color ci is consumed at level n. In such a case we have a token clash.

Two solutions are possible, we can delay the creation of the result colour until the in-line instructions have finished, or we can take care that colours become different. We have opted for the last solution: we change the colour. Before entering such problematic calls, we increment the iteration level of the colour, after returning, we decrement the iteration level. This assures that the results of the two levels have a different colour.

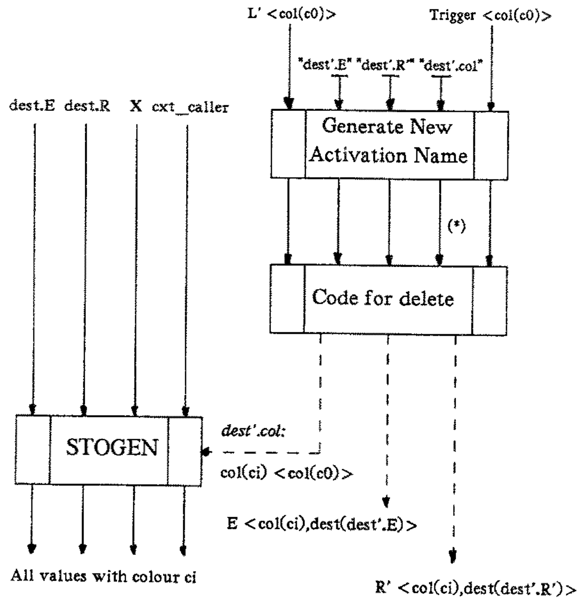
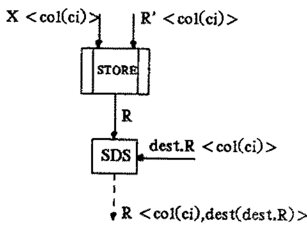


Figure 3.5: Procedure call for delete (E, L', R')
L' is an input parameter. A trigger token Trigger *is also sent to the procedure, to activate parts of the dataflow graph that are not activated automatically by the data. As* E *and* R' *are output parameters, the destination for each of them is sent to the procedure, together with the destination of the colour token* (dest'.col) *and the colour of the caller* (c0)*, both combined in one context token cxt(col(c0)dest(dest'.col)) at (*). All outputs of generate new activation name have a new color c1. For each solution, delete returns the tokens* E *and* R' *as results. They have colour* ci*); also a colour token is returned, it has as value* col(ci) *and as colour the initial colour* c0*. This token is fed into STOGEN, where it matches the other tokens of the clause. STOGEN forwards its other inputs with new colour* ci*. These tokens, combined with the results of delete are used to complete the computation (the statement R <== (X.R')).*

This problem of clashing arises in all cases where, after finishing a call P, an ancestor P' of the same predicate can be completed only by executing in-line code.

3.4 The generate block

At the end of each branch, the results, and the colour of each of them, are returned to the calling procedure by the generate block. For each result we have to return, we use a SDS (Set DeStination) node to send the result to its proper destination (fig. 3.6). The scheme of fig. 3.7 produces the resulting colour token and sends it to its proper destination.



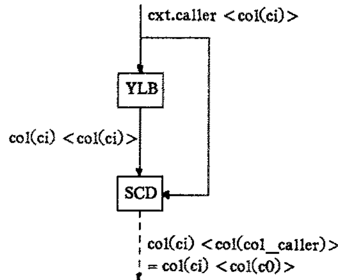


Figure 3.6: R <== (X.R') and returning of R

Figure 3.7: Part of generate block
The YLB (Yield LaBel) node returns the label field of the incoming token.

3.5 Or-parallelism

The branches in a procedure body are activated in parallel. Branches without procedure calls are given a new activation name, except the first one. Because procedure calls start with giving a new activation name (fig. 3.5), this assures that solutions computed in different branches have different activation names.

The duplication of incoming values, and the computation of new activation names for the branches which need it, is done by a split block at the beginning of each branch.

Our scheme for assigning new activation names is as in fig 3.9 (where '&' indicates the generation of a new activation name, and '...' in-line code). This scheme is equivalent with the scheme in figure 3.8, but more economical in activation names, because the third and the fourth branch may fail in the in-line part before generating a new activation name. For procedures which are recursive in more than one branch (as our example), the difference becomes more important, because the difference in use of activation names at the bottom level of recursion is multiplied for each level of recursion. For branches without procedure calls, it would be better to generate new activation names at the end of the branch instead of at the beginning, but this has not been implemented.

```
Query: ..., a, ... .

a :- & ... ;
     & ... ;
     & ..., a, ... ;
     & ..., a, ... .
```

Figure 3.8: A simple scheme for the generation of new activation names.

```
Query: ..., & a, ... .

a :- ... ;
     & ... ;
     ..., & a, ... ;
     ..., & a, ... .
```

Figure 3.9: A more economical scheme for the generation of new activation names.

3.6 An example: delete

Here follows the complete scheme for the procedure delete without garbage collection (see fig. 3.10).

3.7 Garbage collection

Up till now, we didn't pay any attention to make our programs well formed. In fact, there are two sources of tokens which remain in the matching store after execution of the program: i.e. lists and the values that have to survive a procedure call.

The garbage collection on lists can be done automatically by introducing a reference count for each cell of the lists, as can be found in [Bowen 81]. This introduces some synchronisation constraints when manipulating lists, in order to guarantee that the counters should not be decremented to soon. This is not further discussed here.

On the other hand, stored values are deleted when the called procedure sends a finish signal to the calling procedure, which means that no more solutions will be generated (see fig 3.11: partial expansion of the modified macro STOGEN). The value to be stored comes on the left input part of the SCD node. Each arriving colour token combines with the destination of the stored value to send a copy of the stored value to the desired destination in the right colour. The finish signal extracts the stored value and sends it to a kill node (this is the reason why the outcoming arc is dynamic). To generate this finish signal, several other synchronisation signals are necessary, but let us first mention the implementation goals for this garbage collection:

- garbage should be removed as soon as possible;
- its effect on performance should be minimal.

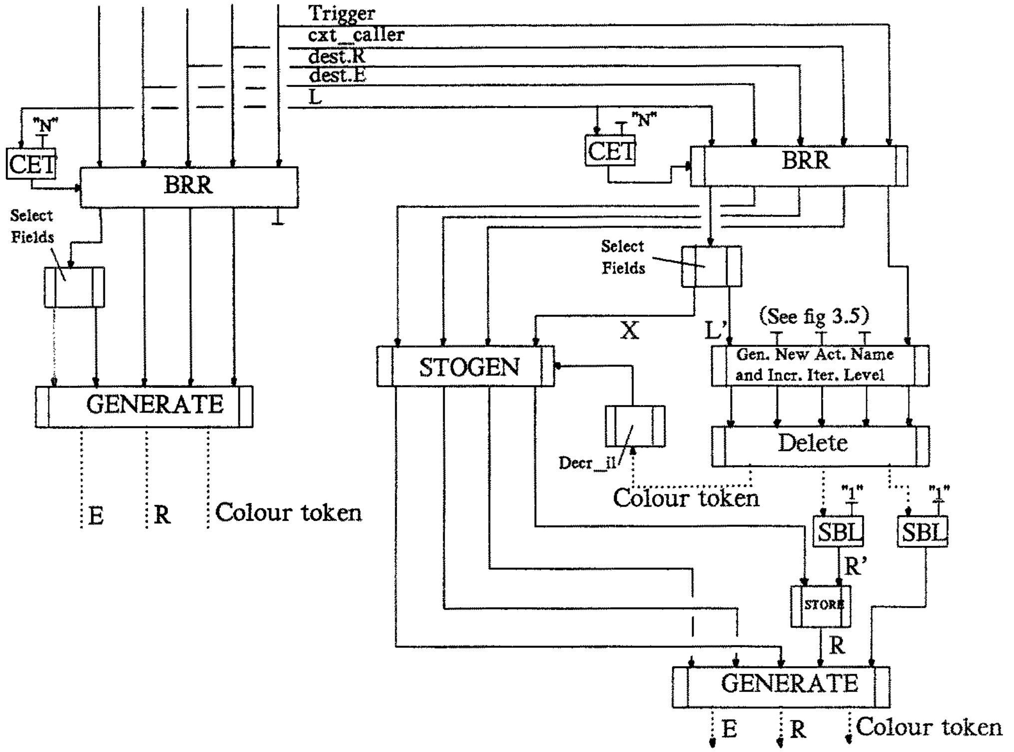
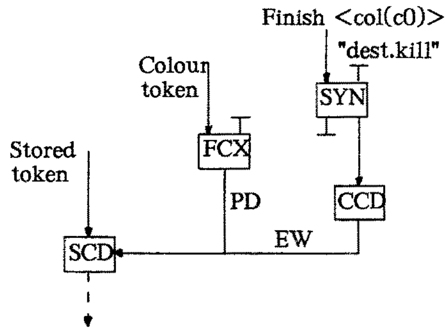


Figure 3.10: Delete without garbage collection
Remark: SBL subtracts 1 from the iteration level in the label, *the "Decr_il" macro subtracts 1 from the iteration level in the* value *of the colour token.*

Figure 3.11: Part of STOGEN with garbage collection
The FCX node (Form ConteXt) combines the colour of the solution of the called procedure with the destination of the node where the stored token is further processed. When the finish signal arrives, it triggers the literal **"dest.kill"**, *which is the address of a KIL node (a KIL node consumes its input tokens without producing any output tokens). This destination token has the same colour as the finish token:* **c0**. *The CCD node (Combine Colour and Destination) transforms the destination token in a context token, which is sent to the SCD node with matching function* EW *to remove the stored token.*



3.7.1 Principle
A procedure call is finished when all branches are finished, so for each branch we need an end-of-branch signal. Further, we do not generate a finish signal for a procedure until we know that all produced solutions are accepted by the calling procedure, because the copies of the stored values at the call may be destroyed as soon as the finish signal arrives. So we need two auxiliary signals: a generate signal that indicates that some branch has produced a solution, and an accept signal that indicates that the calling procedure has processed the solution.

To synchronise this exchange of messages between caller and callee, the callee increments a counter before sending a solution and decrements a counter when the accept signal is received. The finish signal is only forwarded when all solutions are sent and the counter has its initial value.

An end-of-branch signal is derived from the finish signals of the calls or in-line code in the branch. The finish signals travel from right to left in the branch.

When a test fails, a finish signal is generated. When a solution is produced at the end of the branch, a finish signal is released as soon as the generate signal has been registered. Each procedure call in the body of the branch sends a finish signal to its left when

- the called procedure is terminated, i.e. it will produce no more solutions;
- for each solution produced by the called procedure, the part of the branch to the right has been terminated, i.e. it has received a finish signal for each of them.

3.7.2 Algorithm
The behaviour described above is accomplished by extending the function of the already defined blocks. We sum up the functions of each of them here.

The call block

For each activation the call block performs the following functions:

— storage of tokens representing values needed for the activation of the remainder of the branch;

— generation of a copy of those tokens for each solution of the called procedure;

— delivery of an accept signal to the called procedure for each received solution after the generation of the copies of the stored values (i.e.: the activation of the remainder of the branch);

— counting the number of the generated solutions of the called procedure;

— accepting the finish signal of the called procedure;

— accepting a finish signal for each activation of the remainder of the branch;

— sending a finish signal for this activation of the call block to the previous call block (or the split block if there are no previous procedure calls), when the called procedure has finished and the remainder of the branch has finished for each activation initiated by this activation of the call block.

Split block

All the split blocks perform the following functions:

— duplication of all parameters to activate this branch (if not the last branch);

— placing a new activation name on all parameters if necessary;

— accepting an end-of-branch signal from this branch and a finish signal of the next branch; when both are received, a finish signal is transmitted to the previous branch. (The n-th split block doesn't wait of course for the finish signal generated by the next branch).

The first split block performs in addition the following functions:

- counting the generate signals received from the generate blocks of this procedure; and returning a respond signal.
- counting the accept signals of the corresponding call block in the calling procedure;
- generation of a finish signal for this activation of the procedure if 1) the first branch has finished; and 2) a finish signal is received form the second split block (if any); and 3) the number of generate signals equals the number of accept signals.

The generate block

The functions of the generate block are as follows:

- for each solution of this branch, a generate signal is sent to the first split block of this procedure;
- upon receipt of the respond signal of the first split block, the solution of the procedure is sent to the calling procedure and the finish signal is sent to the previous call block (or the split block for this branch if there are no procedure calls in this branch).

The following points are noteworthy in this implementation scheme:

- for each generated solution, the generate signal is guaranteed to arrive before the accept signal: this is the function of the respond signal;
- when a branch fails (because of failing unification), it generates also a finish signal;

3.11 Guards

Up to now, we have ignored the guards. Although a straightforward solution is not difficult to implement (only one partial solution may proceed after the commit-tokens), we aim at aborting all guard evaluations as soon as one partial solution is computed. For the moment, we have an implementation scheme for this, but it has not yet been tested.

There are two kinds of abort signals in a don't-care procedure: internal and external ones. The internal abort signal is generated at a commit-token in the procedure itself, and has to be propagated to all procedures called in the guards (in the form of an external abort signal). The external abort signal is generated at a commit-token of an ancestor procedure and has reached this procedure due to propagation. The processing of the two types of abort signals is the same, but the followed path in the procedure is different. For don't care procedures, we have only external abort signals.

To accomplish this abortion, we associate with each call block a stream of colours. This stream indicates every activation of the procedure associated with the call block.

When a procedure receives an abort signal, it first checks if this is the first abort signal, and if it has not already finished all activity for the activations with the colour of the abort signal (the abort signal and the finish signal may cross each other). If this is the case, then we send recursively one or more abort signals to every directly activated procedure, by using the stream of colours associated with each call block. Further solutions of the procedure are also deleted.

4. Discussion of some experiments

We have tested some small programs with the Manchester simulator [Sargeant 85]. We derived the number of executed instructions, the length of the critical path and the average parallelism as a function of the number of elements of the list. Tables with the results for *quicksort* (deterministic) and *delete* (nondeterministic) are shown in figures 4.1.a and 4.1.b. A first conclusion is that garbage collection introduces much overhead: the average parallelism remains the same, but the number of instructions is multiplied roughly by 3 à 3.5. This is not astonishing,

because, to guarantee a correct execution, we had to introduce a lot of (local) synchronization. It is also an indication that this synchronization did not destroy the OR-parallelism of the program.

quicksort						
#elem.	without g.c.			with g.c.		
	#instr.	cr.p.	par.	#instr.	cr.p.	par.
10	7469	1932	3.9	22693	3846	5.9
20	19279	4872	4.0	58193	9669	6.0

Figure 4.1.a: Results for quicksort
number of instructions, length of critical path and average parallelism (with and without garbage collection)

delete						
#elem.	without g.c.			with g.c.		
	#instr.	cr.p.	par.	#instr.	cr.p.	par.
0	33	10	3.3	84	30	2.8
5	792	136	5.8	2656	374	7.1
10	2402	276	8.7	8041	794	10.1
15	4862	416	11.7	16176	1214	13.3
20	8172	556	14.7	27061	1634	16.6

Figure 4.1.b: results for delete

The number of executed instructions compares rather unfavourable with an implementation on a sequential machine with backtracking: we found a ratio of 1 instruction on a sequential architecture for 7 instructions on a dataflow architecture for delete with garbage collection (9 elements). According to [Gurd et al. 85] a dataflow MIPS has the potential to match the power of a conventional sequential MIPS (for an integration program), so this result seems rather bad. A major explanation is that we store a lot of tokens in the matching store, which have to be destroyed when no longer needed. The structure store might provide a partial solution for this problem. As the structure store appears colourless, some instructions manipulating the colours of the stored tokens might be avoided. However, we will still need the finish signal to decide when the stored tokens can be destroyed.

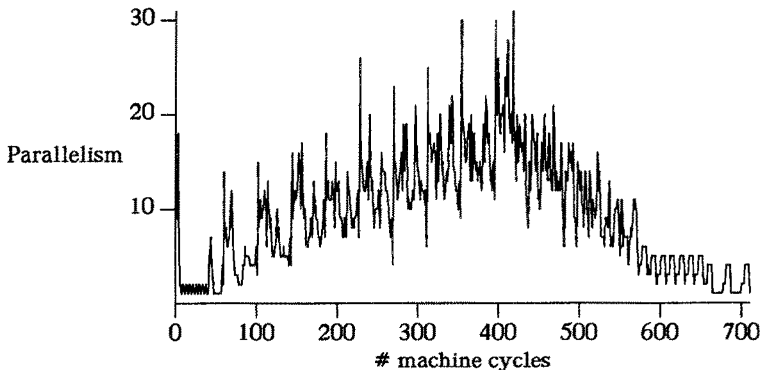


Figure 4.2: parallelism during execution of delete (9 elements)

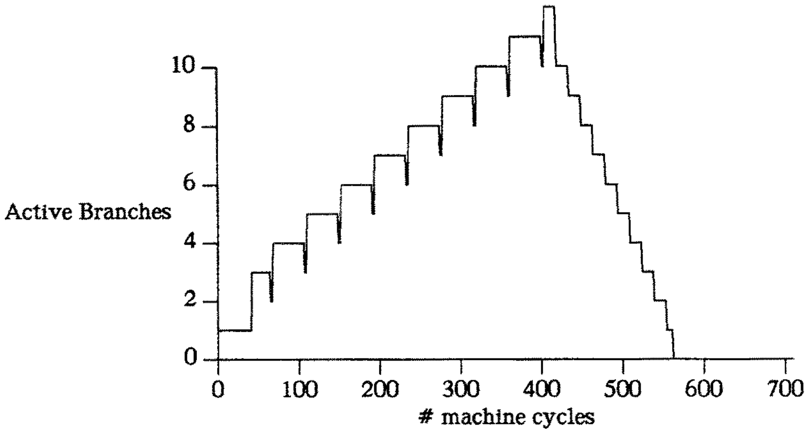


Figure 4.3: active branches during execution of delete (9 elements)

In fig. 4.2 we show the parallelism during the execution of delete with garbage collection for 9 elements. This parallelism is rather unevenly balanced, and may cause for other programs such as 'perm' too high peak values (with the related problems of matching store performance and token queue occupancy).

In fig. 4.3 we show the number of active branches during execution (an active branch is a branch that has received a trigger signal and not yet generated an end-of-branch signal). The procedure delete is activated at cycle 42 (the main activity before is the building of the lists, which is mainly sequential) and terminates at cycle 554. The query finishes at cycle 561, the remaining activity is to remove the lists.

5. Related work

Parallelism in logic programming has grown into a vast research area and it is outside the scope of this paper to attempt a survey. See for example the recent books [Conery 87] and [Wise 86]. Especially the latter gives a dataflow perspective on the field. Another survey focussing on OR-parallelism can be found in [Warren 87].

The only research effort we are aware of having some points in common with our approach is the implementation of flat PARLOG on the parallel reduction architecture ALICE [Lam & Gregory 87]. Both source languages have mode declarations. Flat PARLOG supports AND-parallelism, and thus suspension when trying to access unavailable variables. Our approach doesn't support full AND-parallelism, but there might be some overlapped execution of successive calls. Suspension is provided automatically when no values are available (the hardware is data driven). Our guards are not restricted to non-recursive procedures.

The ALICE architecture supports priority levels, which are used to perform garbage collection in parallel with other activities, but at a higher priority. This might improve the processing of finish and abort signals.

6. Discussion

We have spent a modest effort in exploring the possibilities of executing a logic programming language on the Manchester Dataflow Computer. The results obtained so far are not very encouraging. They indicate that the gains due to parallel execution of instructions are undone by

the increased number of instructions to be executed.

Of course, there are several areas for improvement. First, the procedure calling scheme is too complex for deterministic procedures that always succeed exactly once. In that case, we do not have to store values in the call block, neither does such a procedure need a finish signal. We also expect that the handling of tail recursion can be improved. For the time being, we failed to find a scheme equivalent to an implementation on a sequential machine. For example, the in-line instruction after the tail recursive call of delete causes a major problem; if not present, it might be possible to skip all recursive levels between the first and the last recursive step for certain tokens.

On the other hand, the language was substantially simplified, the most significant restricion being that we excluded full unification by assuming modes. It is expected that the handling of full unification will increase the overhead.

Based on our effort, we are tempted to conclude that the realisation of an efficient PROLOG system on a dataflow machine is an undertaking at least as challenging as it has been for the conventional Von Neumann architecture. Only the widespread availability of such machines can make it a worthwile undertaking.

Acknowledgement

We are indebted to the dataflow research group at Manchester for providing us with the simulator and documentation.

References

— *[Bowen 81]* Bowen, D. L., Implementation of Data Structures on a Data Flow Computer, Ph. D. thesis, University of Manchester, April 1981

— *[Bruynooghe et al. 87]* Bruynooghe, M., G. Janssens, A. Callebaut and B. Demoen, Abstract interpretation, towards the global optimisation of Prolog programs, *Proc. Fourth IEEE Symposium on Logic Programming, San Francisco, september 1987*

— *[Catto 81]* Catto, A. J., Nondeterministic Programming in a Dataflow Environment, Ph. D. thesis, University of Manchester, june 1981.

— *[Catto & Gurd 80]* Catto, A. J. and J. R. Gurd, Nondeterministic Dataflow Graphs, *IFIP 1980*, p. 251 - 256.

— *[Conery 87]* Conery J. S., Parallel execution of logic programs, Kluwer Academic Publishers, 1987

— *[Drabent 87]* Drabent, W., Do logic programs resemble programs in conventional languages? *Proc. Fourth IEEE Symposium on Logic Programming, San Francisco, september 1987*

— *[Gurd et al. 85]* Gurd, J. R., Kirkham C. C. and Watson I., The Manchester Prototype Dataflow Computer, *Communications of the ACM*, January 1985 Volume 28 Number 1, 34 - 52

— *[Kirkham 84]* Kirkham, C. C., The Manchester Prototype Dataflow System, Basic Programming Manual, November 1984

— *[Lam & Gregory 87]* Lam, Melissa and Steve Gregory, PARLOG and ALICE: a Marriage of Convenience, *Proc. Fourth International Conference on Logic Programming*, Melbourne, p.294 - 310

— *[Sargeant 85]* Sargeant, J., Simulator Users Guide, University of Manchester, january 1985

— *[Veen 86]* Veen, A. H., Dataflow machine architecture, *ACM Computing Surveys*, Vol. 18, No 4 (december 1986), p. 365 - 396

— *[Warren 87]* Warren, D. H. D., Or-Parallel Execution Models of Prolog, *TAPSOFT 87: Proceedings of the international joint conference on theory and practice of software development*, Pisa, Italy, March 87, Lecture Notes in Computer Science 250, p. 243 - 259.

— *[Wise 86]* Wise, M. J., Prolog multiprocessors, Prentice-Hall, 1986