A new Scheme for Unification in WAM.

André Mariën

BIM Kwikstraat 4 B-3078 Everberg Belgium bimandre@cs.kuleuven.ac.be

Bart Demoen

K.U.Leuven Celestijnenlaan 200A B-3001 Belgium bimbart@cs.kuleuven.ac.be
BIM Kwikstraat 4 B-3078 Everberg Belgium

Abstract

Prolog spends about 50% of its execution time in unification. It is therefore worthwhile to investigate improvements to unification. The propagation of **read-write** mode in WAM is a candidate for improvement. We present a new translation scheme to deal with it. The scheme has the following advantages: there is no code explosion; for right sided structures - the usual form lists take in Prolog programs - there is no overhead in the propagation of the **read-write** mode; for left sided structures, the overhead is made very small by rearranging the order in which the arguments of the structure are treated; it uses the optimal structure creation code in the **write** mode. The scheme is particularly good for native code compilers. It is also shown how the method can take into account particularities of the underlying hardware. Test results show good speedup, making mode declarations for efficiency reasons almost unnecessary. In addition, we include an important improvement to temporary register allocation schemes.

1. Introduction

We assume some knowledge of the Warren Abstract Machine (WAM) [War]: we will use H for the top of the heap (or global stack), Ai and Xi for the overlapping argument and temporary registers, S for the structure pointer during unification. In WAM the translation of the unification A1 = f([8,9],a,g(b)) is decomposed as follows:

A1 = f(X1,a,X2) , X1 = [8|X3] , X3 = [9|[]] , X2 = g(b)

These elementary unification chunks contain no nested structures. If the left hand side is free, the WAM must create the structure on the right. During this construction, the WAM is said to operate in **write** mode. If the left side is instantiated, its functor and arguments are matched pairwise with the right side. In this case, the WAM is said to operate in **read** mode. A straightforward implementation of the elementary unification chunks is given in figure 1. The **get_*** instructions dispatch between a **write** and a **read** mode stream, depending on whether the X (or A) register it refers to, is free or not.

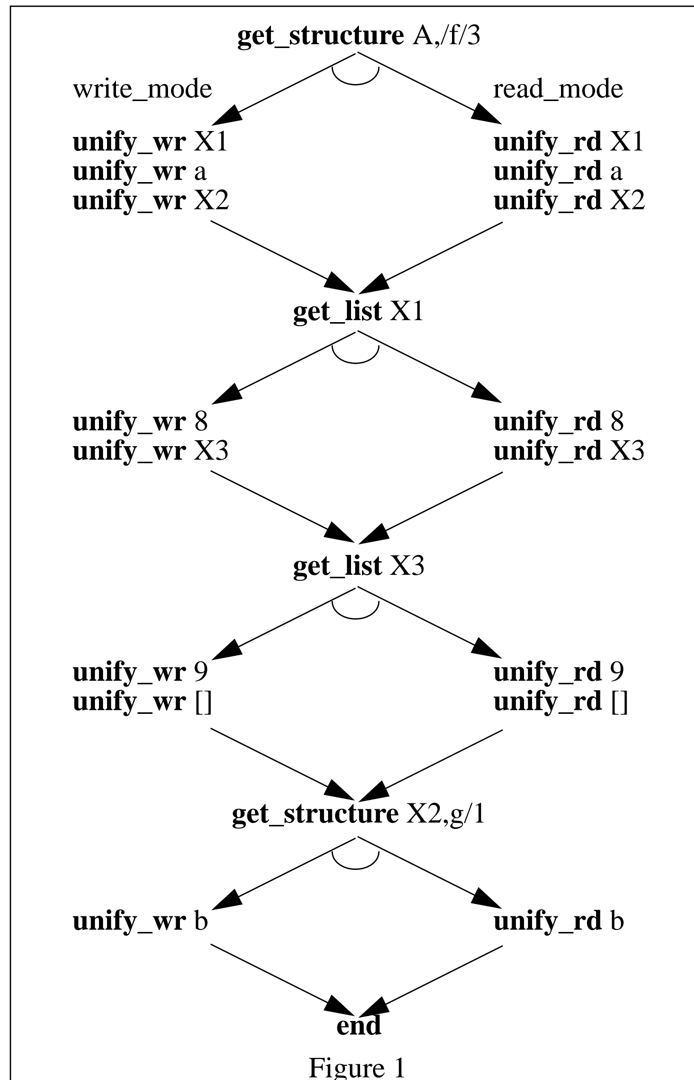

Figure 1

Instead of such a splitting of **read** and **write** stream locally within an elementary unification chunk, WAM has chosen to merge the streams in the following way:

get_structure A1,f/3
unify X1
unify a
unify X2

in which every **unify** instruction must test a **read-write** mode flag (abbreviated by RW), so that the actual code executed in an interpreter takes the following form (leaving out some details of **get_structure**):

```
get_structure A1,f/3     if var(deref(A1)) then RW := WRITE
                                             else RW := READ
unify X1                 if (RW == WRITE) then unify_wr X1
                                             else unify_rd X1
unify a                  if (RW == WRITE) then unify_wr a
                                             else unify_rd a
unify X2                 if (RW == WRITE) then unify_wr X2
                                             else unify_rd X2
```

The code size - in number of abstract machine instructions - is certainly smaller in WAM than when having a separate stream for the **read** and **write** mode for an elementary unification. For an interpreting implementation, the compactness of the code, outweights the slight speed degradation, and it is clear that WAM as such is very much interpreter oriented. In a compiling system, where native code is generated, the size of the abstract code is less important, and in fact, the native code for two separate streams is smaller than for a single stream since there are no tests on RW. On the other hand, even in interpreters, the excessive testing of RW can be avoided. This is discussed in the next section.

2. The propagation of RW in interpreter systems

We describe two methods to avoid the excessive testing of WR. Both methods have been implemented in various systems.

2.1. two different loops in the interpreter instead of one

The first method is suitable for an interpreter: the interpreter is written with one loop for **read** mode unifications and another loop for **write** mode unifications. The **get_*** instructions dispatch to the appropriate loop:

```
get_structure A1,f/3     if var(deref(A1)) then goto write_loop
                                             else goto read_loop
```

and in the write_loop, **unify** means **unify_wr**, while in the read_loop, **unify** means **unify_rd**. Each loop is ended by encountering any non-basic unification instruction.

2.2. two different entry points per unification instruction

A second way to avoid excessive testing, is by giving each unify-instruction two entry points, a read_entry point and a write_entry point, with a fixed relation between the two entry points, so that execution follows the pattern:

```
get_structure A1,f/3            if var(deref(A1)) then goto L1+1
                                                   else goto L1
unify X1              L1:                          goto read_entry1
                              write_entry1:        unify_wr X1
                                                   goto L2+1
                              read_entry1:         unify_rd X1
                                                   goto L2
unify a               L2:                          goto read_entry2
```

		write_entry2:	**unify_wr** a
			goto L3+1
		read_entry2:	**unify_rd** a
			goto L3
unify X2	L3:		goto read_entry3
		write_entry3:	**unify_wr** X2
			goto L4+1
		read_entry3:	**unify_rd** X2
			goto L4
end_unify	L4:		NOP

The 'goto Li' immediately before the label Li, is redundant in this piece of code, but in an interpreter or threaded code system these code fragments need not be consecutive. In a compiled approach they would never be generated.

3. Framework for the proposed unification compilation

When compiling one would like to go further and avoid most jumps. This can be achieved by code duplication, as illustrated by figure 3a for the unification A1 = f([8,9],a,g(b)). The execution goes left to the **write** stream, if the var-test succeeds, otherwise right to the **read** stream. As one can see:

X1 = [8|X3] is repeated twice in write mode
X2 = g(b) is repeated twice in write mode
X3 = [9|[]] is repeated three times in write mode

At first sight, it looks as if this strategy leads to unacceptable code explosion: in principle this is the case, but one must bear in mind that only write mode code is duplicated, and that the optimal write mode native code is very dense [Mar2]. It means that for terms with little nesting, the growth of the code size will be modest. Moreover, the above graph can be reduced to figure 3b, in which now only

X1 = [8|X3] is repeated twice in write mode
X3 = [9|[]] is repeated twice in write mode

This optimization can be performed for the last argument of a structure. It is applicable for right-sided structures, which covers most uses of lists in Prolog programs, making the optimization really worthwhile.

One could wish to get rid of all code duplication while still maintaining a separate **read** and **write** stream. In general this means that there is jumping back and forth between the **read** stream and the **write** stream. For the example above, this leads to figure 3c, in which black arrows occur in pairs and indicate choices, while shaded arrows indicate unconditional transitions. This last scheme will be generalized and optimized.

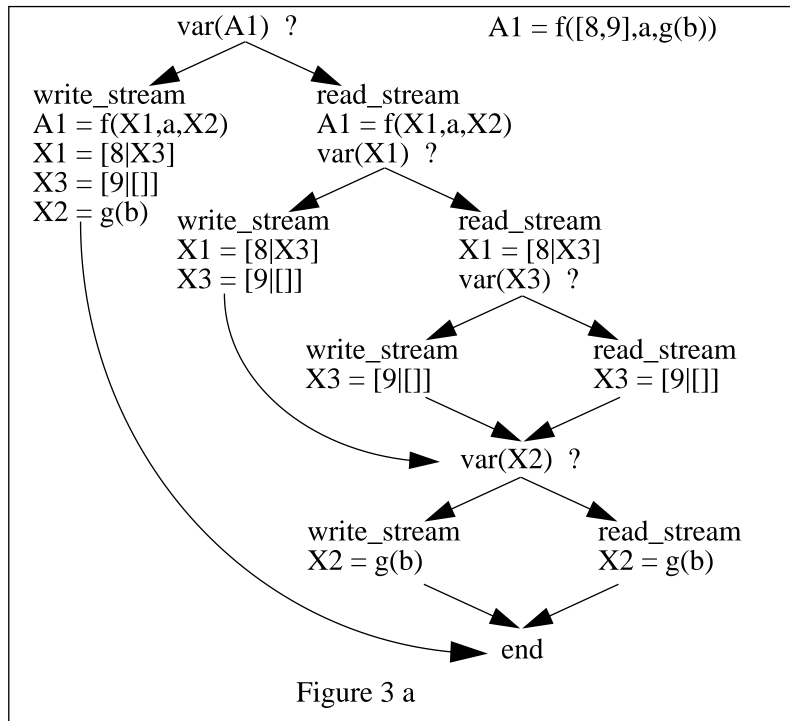

Figure 3 a

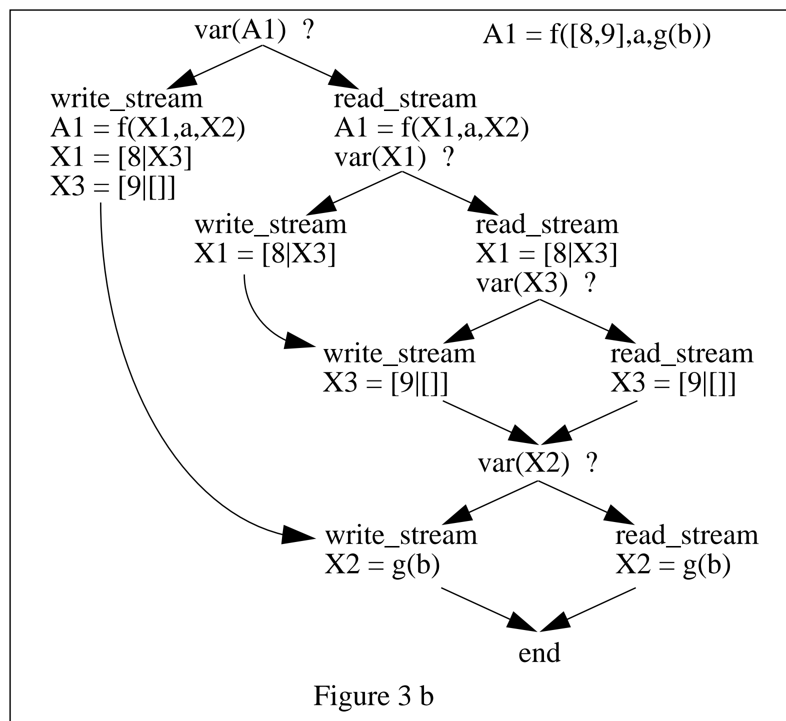

Figure 3 b

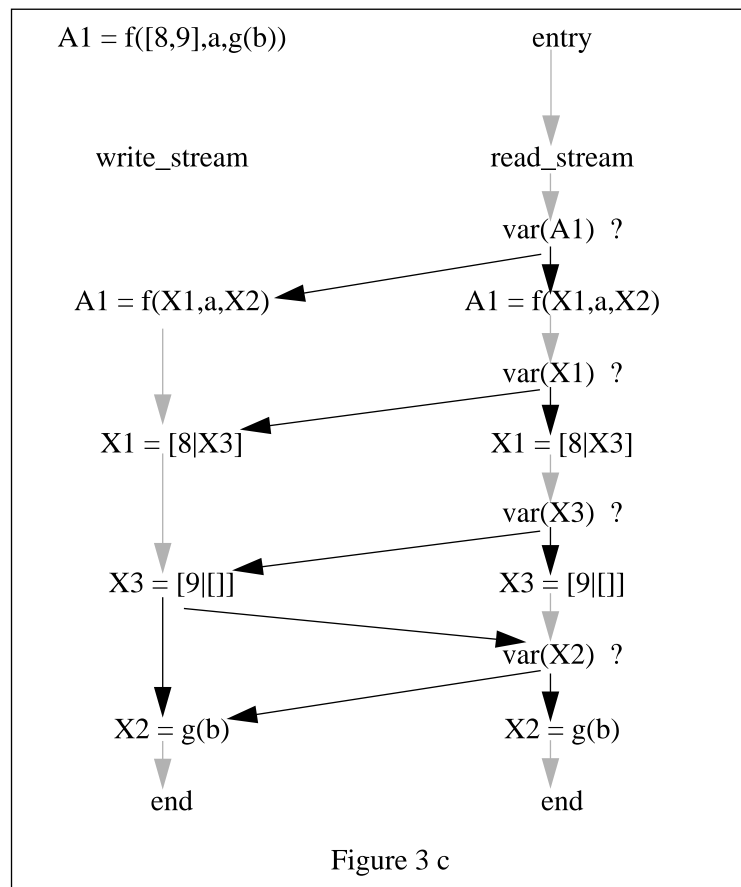

Figure 3 c

4. Relation with other work

[Tur] is one of the first to report on improvements to WAM unification, notably by propagation of the **read-write** mode. He proposes to generate two streams for each unification: the **write** mode and the **read** mode stream. He describes a form of **write** mode propagation, which can be used for the leftmost substructure of a term. This system can propagate **write** mode in the common case of flat lists. For the example above, there are two possibilities for propagation: if A1 = f(X1,a,X2) is in **write** mode, one can step to the **write** mode of X1 = [8|X3] , after which **write** mode can be propagated to X3 = [9|[]] .

The improvements for unification in interpreter based systems described in section 2, have been implemented in various systems, but were not yet published as far as we know. [Van] implemented the optimization for right-sided structures. The most recent and very important work on **read-write** mode propagation, has been reported by [Mei]. He treats structures in a top-down way, so that some unification failures are detected earlier than in ordinary WAM, but also some failures are detected later. The

essence of his method is that there is a separate **read** and **write** stream. His scheme is biased towards right-sided structures, for which almost optimal code is produced by his proposal for many common cases, except that the **write** stream used, is not optimal. Other improvements to WAM unification are in [Mar2] in which optimal compilation of compound term construction is described, and in [Deb],[Jan], [Umr] where temporary register allocation for the WAM was improved.

5. Optimal structure creation

Since our scheme uses the optimal code in the **write** stream, we give here a brief introduction: for a detailed account, see [Mar2]. The idea is that for a term to be constructed, the address - relative to H - of every component of that term can be determined at compile-time. Given this observation, the best possible code (for a machine with auto-increment instructions) for the argument of the goal do([7,8,9]) is:

```
put_list        A1    % A1 = H + listtag
push_integer    7     % *H++ = 7 + inttag
push_list       1     % *H++ = (H+1) + listtag
push_integer    8
push_list       1
push_integer    9
push_constant   []
```

It is fairly easy to adapt this code to architectures without auto-increment, and make use of H-relative addresses.
The code for structure creation has one abstract machine instruction for each Prolog item to be created, and each abstract instruction corresponds to 1 or 2 machine instructions, which means that the overhead of threaded code or an interpreter loop - typically at least 2 instructions- is higher than the actual useful work to be done. From this, it is obvious that the proposed code works best in a system which generates native code.
This approach should be compared to the best WAM approach: in the WAM, it is best to built structures bottom up. This is a well known idea, one of the most recent references being [Piq]. The code above improves this, and can be adapted to create a structure in almost any desired order, e.g. top-down as above. This is an important property, as head unification in WAM can not be done bottom-up in WAM without a serious loss in performance. Therefore, the best construction of structures in the WAM as in [Piq], can not be combined smoothly with **read** mode unification, whereas with our **write** mode code, it can be done.

The optimal structure creation code offers two important opportunities: firstly, in an improved head-unification scheme, one would like to emit code like the above for the **write** stream, but the problem how to link the **read** stream and this optimal **write** stream is still to be solved. This will be done in section 6.
Secondly, the optimal code for the construction of arguments, allows for a nice integration in, and improvement of temporary register allocation schemes: indeed, for clauses like: head(A) :- goal([A]), ..., allocation schemes see little way out but to store

A temporarily in say X2, in order to make the construction of [A] possible in X1. We translate the example to:

```
push_xval           X1
push_constant       []
delayed_put_list    A1,-2
```

where the -2 as argument to **delayed_put_list**, is the offset relative to H of the list pointer to be put in argument register 1. The saving of A into some other register is avoided by a heap-relative instruction instead of a heap-nonrelative instruction. This actually costs nothing on many processors and it greatly reduces the complexity of temporary register allocation schemes as in [Abe], [Deb], [Jan] or [Umr] since it resolves conflicts at no cost.

6. New and modified instructions

In this section, we introduce gradually the modifications to existing instructions, and the new instructions we need for our scheme. The instructions are described in a C-like pseudo code. In the WAM instructions, we use the notation @i for label i.

6.1. examples of a right-nested structure or list

Illustrated feature: no jumps back from **write** mode to **read** mode

Prolog code:

```
        head([a,b]) .
```

modified WAM code:

```
        get_list          A1,@1        % if var(A1) then goto @1
        unify_constant    a,1          % post increment S by 1
        unify_list        @2           % if var(*S) then goto @2
        unify_constant    b,1          % post increment S by 1
        unify_constant    [],0         % NO increment for S
        jump              @end
@1      push_constant     a
        push_list         1            % list points to next cell
@2      push_constant     b
        push_constant     []
@end:
```

Prolog code:

```
        head(f(a,g(b,c))) .
```

modified WAM code:

```
        get_structure     A1,f/2,@1
        unify_constant    a,1          % post increment S by 1
        unify_structure   g/2,@2
        unify_constant    b,1          % post increment S by 1
        unify_constant    c,0          % NO increment for S
        jump              @end
```

```
@1    push_constant    a
      push_structure   1          % struct points to next cell
      push_functor     g/2
@2    push_constant    b
      push_constant    c
@end:
```

Pseudo code definitions for the instructions:

get_list Ai,@j

```
S := deref(Ai) ;
if var(S) { *S := H + listtag ;          % var becomes tagged ptr to heaptop
            trail(S) ; goto @j ; }
if not list(S) goto failure ;
S := S - listtag ;                       % prepare for read mode unification
```

get_structure Ai,F/A,@j

```
S := deref(Ai) ;
if var(S) { *S := H + structtag ;        % var becomes tagged ptr to heaptop
            trail(S) ; goto @j ; }
if not struct(S) goto failure ;
S := S - structtag ;
if(*S != functor(F/A)) goto failure ;    % check if correct functor
S++ ;                                    % prepare for read mode unification
```

unify_list @j

```
S := deref(S) ;                          % 'last' element of previous struct
if var(S) { *S := H + listtag ;          % var becomes tagged ptr to heaptop
            trail(S) ; goto @j ; }
if not list(S) goto failure ;
S := S - listtag ;                       % prepare for read mode unification
```

unify_structure F/A,@j

```
S := deref(S) ;                          % 'last' element of previous struct
if var(S) { *S := H + structtag          % var becomes tagged ptr to heaptop
            trail(S) ; goto @j ; }
if not struct(S) goto failure ;
S := S - structtag ;
if(*S != functor(F/A)) goto failure ;    % check if correct functor
S++ ;                                    % prepare for read mode unification
```

unify_constant C,i

```
if var(deref(S)){ *deref(S) = C ;
                  trail(deref(S)) };     % var becomes constant
if(deref(S) != C) goto failure ;         % check if correct constant
S += i ;                                 % adapt S
```

push_list i

```
*H := (H+i) + listtag ;          % push tagged ptr to cell at distance i
H++ ;
```

push_constant C

```
*H = constant(C) ;               % push tagged ptr to cell at distance i
H++ ;
```

push_structure i

```
*H := (H+i) + structtag ;        % push tagged ptr to cell at distance i
H++ ;
```

push_functor F/A

```
*H := functor(F/A) ;             % push functor descriptor
H++ ;
```

6.2. example of a left-nested structure

Illustrated feature: manipulation of S to avoid jump backs

Prolog code:

```
head(g(h(a,b),c)) .
```

modified WAM code:

```
     get_structure    A1,f/2,@1
     increment_S      1          % increment S by 1
     unify_constant   c,-1       % post decrement S by 1
     unify_structure  g/2,@2
     unify_constant   a,1
     unify_constant   b,0
     jump             @end
@1   push_structure   2          % points to two cells ahead of this cell (g/2)
     push_constant    c
     push_functor     g/2
@2   push_constant    a
     push_constant    b
@end:
```

Pseudo code definition of the new instruction:

increment_S i

```
S += i ;
```

6.3. example of a nested structure, with the need for jump back.

Prolog code:

```
head(f(g(a),h(foo))) .
```

modified WAM code:

```
      get_structure   A1,f/2,level=1,@1  % to write mode till the end
                                         % test_level ahead
      move_star_S     X1,1               % save argument for later use
      unify_structure h/1,level=2,@2     % goto write mode temporarily
      unify_constant  foo,0
@4    get_structure   A1,g/1,level=0,@3  % to write mode till the end
                                         % no test_level ahead
      unify_constant  a,0
      jump            @end
@1    push_structure  4
      push_structure  1
      push_functor    h/1
@2    push_constant   foo
      test_level      2,@4
      push_functor    g/1
@3    push_constant   a
@end:
```

Pseudo code definition of the instructions:

move_star_S Xi,j

Xi := *S ; S += j ;

test_level i,@j

if L >= i goto @j ;

The ‘level=i’ argument in the **get_structure** and **unify_structure** instructions, indicates that a new global register, called L, is set to the value i. The **get_list** and **unify_list** are augmented with this extra argument as well. This register L is used in the **test_level** instruction. We have adopted the convention to number the levels upwards starting from 0, where 0 means: do not set the level, because no **test_level** instruction will be reached after the transition to the **write** stream has been made. There is no **test_level** instruction referring to level 1, because 1 is the level set if the execution has to stay in the **write** stream until the end of the **write** stream, but has to pass by at least one **test_level** instruction.

6.4. example combining the reordering of subterms with jump backs

Prolog code:

```
head(f(h([foo]),g(a,b))) .
```

modified WAM code:

```
get_structure   A1,f/2,level=1,@1  % fall through test
move_star_S     X1,1               % postpone h([foo])
unify_structure g/2,level=2,@2     % come back after
                                   % construction of g(a,b)
unify_constant  a,0
```

```
      unify_constant  b,0
@4    get_structure   A1,h/1,level=0,@3
      unify_list      level=0,@5
      unify_constant  foo,1
      unify_constant  [],0
      jump            @end
@1    push_structure  5              % ptr to h([foo])
      push_structure  1              % ptr to g(a,b)
      push_functor    g/2
@2    push_constant   a
      push_constant   b
      test_level      2,@4           % back to read mode if
                                     % only g(a,b) in write mode
      push_functor    h/1
@3    push_list       1
@5    push_constant   foo
      push_constant   []
@end:
```

The idea is to treat the deeper nested structures later than the less deeply nested ones. A similar idea can be found in [Umr], where it is used to diminish the number of temporary registers used. Here, reordering has the same effect, but it also reduces the number of **test_level** instructions, as well as the number of taken jumps from the **write** stream to the **read** stream, the actual number being dependent on the instantiation of the argument of course.
This reordering of the arguments of a term, complicates the generation of the **write** stream: indeed: the order in the **write** stream must be the same as the order in the **read** stream. In our implementation, the **read** stream is generated first, starting from the parse tree of the clause, and the **write** stream is generated from the **read** stream.

7. Measurements

The above scheme was implemented in August 1989 on several architectures (SPARC,68020,80386) and within the native code generation scheme adopted in BIM_Prolog release 2.4.3. We give here a small table of results, split in two parts: the first three tests show the effect on unification alone, the other four tests are somewhat larger: the boyer program is a theorem prover, simplify is an expression simplifier, queen is a moderately fast 8-queens program, triang is a solver for the triangle version of the solitair game. Per processor, the ratio of the running times of specific tests with and without the new unification scheme are shown.

	68020	80386	sparc
put	2.48	3.1	2.41
get(readmode)	1.35	1.24	1.25
get(writemode)	3.68	5.4	2.72

boyer	1.01	1.05	1.02
simplify	1.04	1.02	1.01
queen	1.06	1.07	1.03
triang	1.03	1.04	1.17

Good speedup of unification was noticed in all cases. As modes give an improvement in the order of 10% with the normal WAM, the new code reduces their effect dramatically. The main effect of modes then remains the reduction in code size.

8. Comparison with related work

The closest related work is in [Mei]. We point out the most important differences with our work:

> [Mei] treats arguments in a top-down fashion, rather than in a breadth-first manner as is usual in WAM. Our scheme can be adapted to do the same: it is a matter of storing and restoring S. It will not make the scheme essentially different. As [Mei] points out, by treating arguments in a top-down fashion, the number of necessary temporary registers is reduced and some unification failures are detected earlier. Of course, other unification failures are detected later by [Mei] as well.
>
> [Mei] does not consider reordering of the arguments inside a compound term: by not doing so, more jumps from **read** stream to **write** stream and vice-versa are performed, and - although this is less important - more temporary registers are needed. Also, it biases the method in [Mei] towards structures nested to the right.
>
> In [Mei], the **write** stream is not optimal: the optimal **write** stream (which uses no temporary registers at all) is described in [Mar2] and is incorporated here.
>
> [Mei] could not report real speedups, because the scheme was only implemented in an emulator: we expect that also our method yields little performance improvement in emulators or even threaded code implementations. Moreover, the code size in WAM-like instructions, is considerably larger than in WAM. Still, since we implemented our scheme in a native code compiler, we have been able to prove that the scheme yields an important performance gain.
>
> Our method is better than the one in [Mei].

[Mei] also points out some optimizations - like collapsing consecutive instructions for void variables in **write** mode or omitting them in **read** mode - which are all compatible with our approach and were in fact implemented.

9. Concluding remarks

The essence of the above presented scheme, is that the optimal **write** stream is integrated in head unification. The cost of jumping between the streams is kept low, on one hand by the very mechanism itself, i.e. the use of the level number, but also by the reordering of arguments, which reduces the need for jumping between the streams. The optimal **write** stream is useful also for resolving conflicts in temporary register allocation.

Our scheme makes mode declarations for most programs almost unnecessary for efficiency reasons, i.e. mode declarations have the effect of reducing the code size because only the **read** or the **write** stream has to be generated.

Special hacks to make the frequently occurring cases of flat lists work optimally, have always been around. Our method is no such hack: it produces optimal code for frequently occurring cases as well as for rare cases - deeply nested structures to the left and right - because of the generality of our approach.

The described scheme, has been adapted to architectures without auto-increment: it then uses H-relative addressing instead. More details can be found in [Mar1]

Finally, since the code size does not increase dramatically when duplicating code, as described in section 3, it is in many cases a viable alternative to the described scheme.

10. Acknowledgment

B.D. wishes to thank the ‘Diensten voor de Programmatie van Wetenschapsbeleid’ of the Belgian Government for sponsoring part of this research, by project RFO/AI/02.

11. References

[Abe] S.Abe et al., ‘A New Optimization technique for a Prolog Compiler’, Proceedings of CompCon 86 Spring, San Fransisco, March 1986, pp 241-245

[Bow] K.A. Bowen, K.A. Buettner, I. Cicekli, A.K. Turk, ‘The design of a high-speed incremental portable Prolog compiler’ 3ICLP pp 650-656

[Car] M. Carlsson, ‘Design and Implementation of an OR-Parallel Prolog Engine’ PhD thesis SICS Dissertation Series 02, ISSN 1101-1335 1990

[Deb] S. Debray, ‘Register Allocation in a Prolog Machine’ 3SLP 1986

[Jan] G. Janssens, B. Demoen, A. Marien. ‘Improving the register allocation in WAM by reordering unification’ International Conference & Symposium on Logic Programming, Seattle, Washington aug 1988

[Mar1] A. Mariën, B. Demoen, ‘A new Scheme for Unification in WAM’ KUL-CW report 125, March 1991

[Mar2] A. Mariën, ‘An optimal intermediate code for structure creation in a WAM-based Prolog implementation’ Proceedings of the International Computer Science Conference ‘88, Hong Kong, Dec 1988, pp. 229-236

[Mei] M. Meier, ‘Compilation of Compound Terms in Prolog’, North American Conference on Logic Programming, Oct 1990, pp. 63-79

[Piq] F. Pique, ‘Compilation d’un Prolog II Modulaire’, Thesis, Université AIX-MARSEILLE II, Februari 1990

[Tur] A. Turk, ‘Compiler optimizations for the WAM’ Third International Conference on Logic Programming, pp. 657-662, London, July 1986

[Umr] Z. Umrigar, ‘Finding advantageous Orders for Argument Unification for the Prolog WAM’ North American Conference on Logic Programming, Oct 1990, pp. 80-96

[Van] P. Van Roy, ‘Can Logic Programming Execute as Fast as Imperative Programming ?’ Report No. UCB/CSD 90/600 Dec 1990, Berkeley, California 94720

[War] D.H.D. Warren, ‘An Abstract Prolog Instruction Set’ Technical Report, SRI International, Artificial Intelligence Center, August 1983