

**On the impact of argument passing on
the performance of the WAM and
B-Prolog.**

Bart Demoen, Phuong-Lan Nguyen

Report CW 300, September 2000



Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

On the impact of argument passing on the performance of the WAM and B-Prolog.

Bart Demoen, Phuong-Lan Nguyen

Report CW 300, September 2000

Department of Computer Science, K.U.Leuven

Abstract

B-Prolog deviates from WAM mainly by its parameter passing mechanism: parameters are pushed on the control stack instead of being passed through argument registers. Also, the B-Prolog emulator has a quite high performance on the classical benchmark set despite its being ANSI-C compliant. It is therefore tempting to attribute its performance to the parameter passing convention. This issue is investigated here in more detail. Instruction compression was a key issue in this study, so we discuss issues related to it. We next show with a set of artificial deterministic and backtracking programs that the WAM and B-Prolog can outperform each other arbitrarily, indicating that - at least for emulators - neither has a systematic advantage over the other. Moreover, we show that whenever the WAM parameter passing schema is worse, a very simple program transformation can be applied which gives WAM the benefits of B-Prolog's argument passing schema, and this without leaving the WAM framework. Finally, as B-Prolog also offers an efficient implementation of delayed goals, we describe how the same efficiency can be obtained in a WAM environment. All empirical results are obtained with dProlog.

On the impact of argument passing on the performance of the WAM and B-Prolog.

Bart Demoen

Department of Computer Science
Katholieke Universiteit Leuven
B-3001 Heverlee, Belgium
bmd@cs.kuleuven.ac.be

Phuong-Lan Nguyen

Institut de Mathematiques Appliquées
Université Catholique de l'Ouest
49000 Angers, France
nguyen@ima.uco.fr

Abstract

B-Prolog deviates from WAM mainly by its parameter passing mechanism: parameters are pushed on the control stack instead of being passed through argument registers. Also, the B-Prolog emulator has a quite high performance on the classical benchmark set despite its being ANSI-C compliant. It is therefore tempting to attribute its performance to the parameter passing convention. This issue is investigated here in more detail. Instruction compression was a key issue in this study, so we discuss issues related to it. We next show with a set of artificial deterministic and backtracking programs that the WAM and B-Prolog can outperform each other arbitrarily, indicating that - at least for emulators - neither has a systematic advantage over the other. Moreover, we show that whenever the WAM parameter passing schema is worse, a very simple program transformation can be applied which gives WAM the benefits of B-Prolog's argument passing schema, and this without leaving the WAM framework. Finally, as B-Prolog also offers an efficient implementation of delayed goals, we describe how the same efficiency can be obtained in a WAM environment. All empirical results are obtained with dProlog.

1 Introduction

For a good introduction to WAM, see [1, 15]; likewise for B-Prolog, see [17, 18]. It might seem strange that on the one hand the WAM - an abstract machine - and on the other hand B-Prolog - a concrete Prolog implementation - are mentioned and compared, while we are reporting on experience with only one WAM implementation, namely dProlog ([9] - available from <http://www.cs.kuleuven.ac.be/~bmd/dProlog1.0>). The reason is simply that on the one hand the results we obtained for dProlog are applicable many other WAM based implementations, like Yap, SICStus, XSB ... and on the other hand, B-Prolog is the only implementation we know off, that implements TOAM and argument passing through the stack.

The start of this research consisted of two observations: B-Prolog is quite efficient when compared to other emulators, especially considering that B-Prolog does not get any of its performance by relying on non-ANSI-C features. And, B-Prolog deviates from the WAM in certain aspects, the most striking one being that the arguments to a goal are passed through the control stack instead of through argument registers as the WAM does.

When these two facts are put together, it becomes tempting to conclude that B-Prolog is fast because it uses the stack argument passing convention. A closer look is in place however: it is clear that to identify more precisely the impact of a particular difference with the WAM, it would be most useful to have two implementations which only differ in that particular aspect. We have followed

that line of thought earlier in [9] for the term representation and emulator choices. Our original intent was thus to implement the stack argument passing convention also within dProlog. However, adapting the XSB compiler on which dProlog is based for the new argument passing convention while keeping all other things equal turned out to be impractical. Still, while investigating this line of work, we became more and more aware of other differences between B-Prolog and dProlog (as well as other WAM implementations we are aware of) which might be responsible for the perceived speed of B-Prolog.

We consequently changed our attack of the question: why is B-Prolog as fast as it is? We will try to answer this question by having a closer look at the benchmarks for which dProlog is slower than B-Prolog, by inspecting the abstract machine code that B-Prolog generates and by trying to include the underlying idea to dProlog in the hope to make it at least as fast as B-Prolog. We use dProlog [9] for comparison, because it is a good vehicle for us to experiment with emulator options and compiler (abstract machine code generator) variations.

In section 2 we present an initial comparison between B-Prolog and dProlog on a set of classical benchmarks and on some artificial ones used earlier: this shows a gap in performance for some benchmarks, in favour of B-Prolog. In section 3 we introduce a double emulator schema in dProlog: B-Prolog uses this schema and it is responsible for some its speed. In section 4, we take a closer look at the abstract machine code that the B-Prolog compiler generates and apply some of its ideas to dProlog, again narrowing the gap. The remaining three benchmarks are put to further scrutiny in section 5. In section 6 we study in more detail the impact of the argument passing schema of B-Prolog on backtracking predicates. A similar study is performed on deterministic programs in section 7 where in particular the impact on last call optimization is studied. Section 8 looks more closely at argument reuse in both contexts. Section 9 looks a bit closer at instruction compressions while section 10 discusses the connection between the argument passing mechanism and the implementation of delay. Section 11 tries to give some general advice on writing efficient emulators for Prolog.

All the experiments were only performed on a Pentium II, 260MHz, 128 Mb - our findings might not carry over to other architectures, but we are actually convinced they do. Timings are reported in milliseconds. We used B-Prolog 4.0 #3.

2 A first comparison between dProlog and B-Prolog

We use a superset of the benchmark set in [9] - see also [10]. In section 7 we will also use a set of artificial benchmarks that show in a particular the impact of the choice made by B-Prolog for passing parameters on last call optimization.

We first compare dProlog 1.0 as used in [9]. In this paper, we always use dProlog (any version) configured in ANSI-C mode. This means that the emulator consists of an ordinary switch statement and that the read-write propagation is done as in the WAM (except when we introduce later the double emulator loop). Furthermore, we always use the *heap_vars* representation (see [9]) for more info.

The benchmark results are shown in table 1. At this point, the relevant columns in table 1 are the ones for dProlog1.0 and B-Prolog. On average, dProlog1.0 and B-Prolog seem to have similar performance already indicating that there is no huge performance edge in passing arguments through the stack - at least not in this benchmark suite. For some benchmarks however, dProlog1.0 lags suspiciously behind and we will try to find out why. There are also some benchmarks for which B-Prolog performs significantly worse than dProlog1.0, but we are of course less interested in why B-Prolog is sometimes slower than in finding out why it is sometimes faster. Still, we will give some

indications for performance loss in B-Prolog later in sections 6 and 7.

For the sake of clarity at this point: the column marked dProlog1.1¹ is a version of dProlog with some more optimizations that are described in section 4. rwdProlog1.1 is a version of dProlog1.1 which has a double emulator for read-write mode propagation which is explained in more detail in section 3. Of course, for fair comparison, any of the dProlog systems used for this paper is in full ANSI-C mode: using GNU C specifics results in gains of the order of 50%.

	dProlog1.0	dProlog1.1	rwdProlog1.1	B-Prolog
boyer	550	500	480	430
browse	810	800	770	790
cal	870	860	840	950
chat	550	540	500	670
crypt	720	720	690	710
ham	990	960	930	1310
meta_qsort	740	730	700	790
nrev	1330	1330	1170	1610
poly_10	460	440	410	410
queens	1800	1750	1690	2240
queens_16	1200	1140	1110	1160
reducer	260	260	250	220
sdda	520	520	500	500
send	710	660	650	480
tak	1000	910	870	720
zebra	1050	1050	1060	1320
indexa	370	370	380	930
indexa2	1370	1370	1360	2290
indexf	610	600	590	1130
list	1180	1200	1160	2040
metacall	970	980	940	1110
plus1f	1920	1940	1880	5330
plus1i	1050	810	790	1990
snrev	1640	1470	1320	2880
snrevswap	1630	1460	1300	2840
struct	1590	1270	1210	3330
comp	11340/7190	10520/6540	10230/6380	13960/8890

Table 1: Comparing B-Prolog with different versions of dProlog

Comp is a benchmark consisting of the compiler compiling itself. The times give are respectively total time/time of first pass which includes the reading.

There are two arithmetic benchmarks (tak and send) for which dProlog1.0 is much slower than B-Prolog and two non-arithmetic benchmarks: boyer and reducer. We will investigate later in more detail why this is so. Our method will consist in looking at the abstract machine code that B-Prolog generates and adding features to dProlog.

However, version 1.0 of dProlog is frozen, so any extensions and improvements we make, will

¹soon available at <http://www.cs.kuleuven.ac.be/~bmd/dProlog1.1>

be in the more recent version 1.1.

3 Read-write mode propagation

There are several ways to propagate the read-write mode of the WAM in an emulator. One of them consists in having two separate switch statements, one for the read mode, one for the write mode. Such is a mixed blessing, but we considered it worth trying it in the context of dProlog1.1, especially since also B-Prolog has it. The results for this version of dProlog1.1 - named rwdProlog1.1 - can be seen in all the tables: it is globally better to have the two switches. It is not a priori clear that it must be like this: the compiler will generate two jumptables for the two emulator switches and if they are both large, a performance degradation can result because they are together too big to be in the cache simultaneously. On the other hand, for programs that work mainly in one mode (e.g. tak is only in read mode) the read-emulator might fit completely in the instruction cache and this can be beneficial.

There are other systems that use the double emulator for read-write propagation. However, we have never seen it described in detail, so we feel it is worthwhile doing it here. There are three classes of WAM instructions:

1. instructions that **set** the RW mode flag so that subsequent instructions can test it - ex. `getlist`
2. instructions that **test** RW - ex. `unicon`
3. instructions that do not deal with RW - ex. `putcon`, `call`

First note that the RW mode flag does not need to exist as a separate variable: one can set `S` to its required value for the read mode and to a non-pointer value (e.g. 0) in write mode. The test (`RW == read`) now becomes (`S != 0`).

The double emulator consists of a read-emulator and a write-emulator; they both have a case for each WAM instruction and deal with the above three classes as shown in table 2

	read-emulator	write-emulator
class 1	perform normal WAM action, but instead of setting RW, transfer to read- or write emulator	goto same instruction in read-emulator
class 2	do read mode of instruction transfer to write emulator	do write mode of instruction transfer to write emulator
class 3	perform normal WAM action	goto same instruction in read-emulator

Table 2: Instructions in a double emulator

Here is some code from the read- and write-emulator for the instructions `unicon`, `call` and `getlist`. The following are worth noting explicitly:

- When transfer to the `write_loop` is made, `S` is not set.
- In the `write_loop`, one could also duplicate the code for instructions of classes 1 and 3 instead of the `goto` and only at the end of such an instruction transfer to the `read_loop`. However, the C compiler (with suitable optimization switched on) short circuits the `goto` from write to read mode, so that no extra overhead is involved. This means that with the above code, the locality

<pre> /* part of write-emulator */ write_loop: switch (*pc) { case unicon: push_atom(atom_from_unicon(pc++)); goto write_loop; case call: goto call_label; case getlist: goto getlist_label; } /* end write-emulator */ </pre>	<pre> /* part of read-emulator */ read_loop: switch (*pc) { case unicon: if (unify((*S++),atom_from_unicon(pc++))) goto read_loop; failure; case call: call_label: set_contpc; pc = entrypoint(pred_from_call(pc)); goto read_loop; case getlist: getlist_label: p = deref(Areg[areg_from_getlist(pc++)]); if (is_undef(p)) { make_list(p); trail(p); goto write_loop; } if (is_list(p)) { S = get_pointer(p); goto read_loop; } failure; } /* end read-emulator */ </pre>
---	---

Table 3: Some C code from both double emulator

of code is probably better. Also, just considering the unification instructions (like unicon) the locality seems better, since code for read and write mode is not intertwined. Even benchmarks like tak which have no such uni instructions, benefit from the double emulator and this seems to support the locality thesis. However, such locality should be proven empirically perhaps by studying the cache behaviour on some benchmarks.

- Although the locality of instructions seems better in the double emulator, the locality of data might be worse, since both now two jump tables are accessed instead of just one.
- The double emulator is amenable to indirect threading, but direct threading seems not possible. Note that a double opcode schema as in [5], has similar effects.

Making a version of dProlog1.1 with the double emulator, was a matter of an hours work.

Finally, to have an idea of how much the double emulator gains one should compare the rwd-Prolog1.1 column of table 1 with the dProlog1.1 column, as they differ only in this particular

aspect.

4 A closer look at the B-Prolog abstract machine code

With the goal `?-bpc(< file >)`. B-Prolog produces a file with suffix `.asl` which contains the abstract machine code. A quick inspection of such a code file reveals that B-Prolog has instructions like `para_wy_wy_wy`. Alternatively, one can have a look at `BProlog/Emulator/inst.h`, that is until release 4: release 5 is unfortunately distributed without sources.

The existence of these instructions suggest that a particular instruction compression could be worth while in the WAM context: a succession of `getpvar` instructions, or a succession of `putpva*` instructions. We introduced this instruction compression for sequences of 2 or 3 such instructions. It is default in dProlog1.1.

Another instruction compression that can be learned from B-Prolog, is compressing a sequence of `getstr` and one or two `unitvar` instructions. We actually arrived at it in a different way: we measured the dynamic frequency of pairs of instructions that are executed after each other and both the instruction compressions described in the previous paragraph and the current one followed immediately, as well as a compression that is particular to our use of the XSB compiler (see [16]): the succession of a `try4` instruction² and `gettbreg`.

The next thing one can observe in the B-Prolog implementation is that its builtin calling convention differs from the one used by XSB (and Yap): instead of putting the arguments to a builtin (like `functor/3`) in argument registers 1, 2 and 3, B-Prolog gives as arguments to the `functor` instruction, a description of where (stack or temporary registers) the arguments are and whether they are initialized or not. We have also previously noted (in [9]) that such a convention is better in the context of an emulator. One reason to follow the worse convention, is garbage collection - but there are good solutions for this complication³.

So, we adapted the XSB compiler to generate the better calling convention for `functor/3` and `arg/3`; that is we actually introduced two new predicates `muf/3` and `mua/3` with that functionality - at the moment of writing, they work for the boyer benchmark but not yet for all the other benchmarks.

Finally, B-Prolog has a series of `add` instructions. XSB's arithmetic is particularly cumbersome for the most common arithmetic operation: adding an integer to a variable. So we adapted the XSB compiler to generate a compressed instruction for a sequence of a `putint` and `addreg`: this is still worse than could be done, but is easy and worth its while.

All the above code optimizations were implemented in dProlog1.1. Table 1 give in the column dProlog1.1 the results for the benchmarks and there is on average an improvement in the timings.

Table 4 shows the gain in performance by having the better calling convention for `functor/3` and `arg/3` - just on the boyer benchmark, where `boyer(2)` indicates boyer with `functor` and `arg` replaced by `muf` and `mua`. The additional benefit is that heap consumption is lowered considerably.

	rwdProlog1.1	B-Prolog
boyer(orig)	480	430
boyer(2)	420	—

Table 4: Speedup for boyer by better builtin calling convention

²a specialization of the `try` instruction for arity 4

³better than initializing all stack slots

A first conclusion

As a preliminary conclusion, we can already say at this point that the speed of B-Prolog does NOT derive from its parameter passing mechanism, rather from good instruction compression, good builtin calling convention, good arithmetic and the way it propagates the read-write mode.

Also, B-Prolog does not gain much by its supposedly better indexing: XSB generates suboptimal indexing (double level and with try-retry-trust chains for hash collisions - and dProlog follows that). Still, explicit tests for indexing indicate that B-Prolog even lags behind. We'll investigate this later in section 7.

5 Three more benchmarks

We want to have a closer look at the remaining three benchmarks (reducer, send and tak) for which rwdProlog1.1 is slower - sometimes significantly.

5.1 tak

Basically, tak/4 is written as

```
tak(X,Y,Z,A) :-
    X =< Y,
    ...
tak(X,Y,Z,A) :-
    X > Y,
    ...
```

which B-Prolog (almost) generates code for as if the code were:

```
takdet(X,Y,Z,A) :-
    ( X =< Y ->
      Z = A
    ;
      takdet2(X,Y,Z,A)
    ).

takdet2(X,Y,Z,A) :-
    X > Y,
    X1 is X - 1,
    takdet(X1,Y,Z,A1),
    Y1 is Y - 1,
    takdet(Y1,Z,X,A2),
    Z1 is Z - 1,
    takdet(Z1,X,Y,A3),
    takdet(A1,A2,A3,A).
```

The results for takdet can be found in table 5. Note that B-Prolog does not treat if-then-else well, so it makes no sense to execute takdet in B-Prolog. The rwdProlog1.1 figure for takdet now comes close to the B-Prolog one. We have noted earlier ([10]) that XSB generates several instructions too many for the second clause of tak (in this case takdet2): these, together with a suboptimal instruction set for arithmetic inequality testing, most certainly account for the rest of the difference.

	rwdProlog1.1	B-Prolog
tak (orig)	870	720
takdet	760	—

Table 5: Speedup of tak by deterministic rewriting

5.2 Send more money

The difference in abstract machine code for the send benchmark, is mainly in the code for goals like $R = \setminus = Y$: while B-Prolog generates one instruction *jmp_eql_yy*, XSB generates the sequence *putpval, putpval, subreg, jumpz*. We were convinced that a decent instruction compression could make dProlog competitive, but we actually implemented an *indecent* one: the above sequence of instructions is compressed to one *fail_if_equal* instruction if the label of the jumpz instruction is fail. The results are *indecently* interesting:

	rwdProlog1.1	B-Prolog
send compiled without fail_if_equal	650	480
send compiled with fail_if_equal	360	—

Table 6: Speedup of send by better translation of some arithmetic.

5.3 Reducer

In the reduce benchmark, there is a lot of (shallow) backtracking (on the `t_redex` predicate). Since backtracking seems the only mechanism where B-Prolog can perform better than WAM (see section 6), it seems reasonable to put the blame there. Still, zebra is the backtracking benchmark par excellence (through the `member/2` predicate) and it executes faster in dProlog than in B-Prolog. So, we'll first try to find the reason elsewhere.

As dProlog relies on the XSB compiler for generating abstract machine code, it suffers from the problem that nested terms in the head of a clause are always constructed (from a certain depth on) even if the argument is instantiated. As an example, the fact

```
foo([6,7,8]).
```

Is translated as shown in the first column of the next table:

generated by XSB	plain WAM	compressed WAM
getlist Areg1	getlist Areg1	getlist Areg1
uninumcon 6	uninumcon 6	uninumcon 6
unitvar Areg1	unitvar Areg1	unilist
putlist Areg2	getlist Areg1	
bldnumcon 8	uninumcon 7	uninumcon 7
bldcon []	unitvar Areg1	unilist
getlist Areg1	getlist Areg1	
uninumcon 7	uninumcon 8	uninumcon 8
unitval Areg2	unicon []	unicon []

The second column shows plain WAM code - but with the XSB naming of instructions. The last column shows how a straightforward compression can be applied to the WAM code. Similar compressed code is used in B-Prolog (and many other systems).

This occurs often in the reducer benchmark (in particular in the `t_redex/2` predicate). B-Prolog generates more decent code for this; in particular, it does not construct subterms unnecessarily and it executes 2 instructions less than dProlog because it uses the obvious instruction compression shown above. We have measured the impact of this (mal)feature of the XSB compiler by rewriting the `t_redex/2` predicate according to the principle exemplified on the above fact as

`foo([6|A]) :- A = [7|B], B = [8].`

This result in an expected performance increase: see table 7 where the transformed reducer benchmark is indicated as `reducer(2)`. Note that this transformation still results in suboptimal code for dProlog1.1, as no similar instruction compression as in B-Prolog is performed.

	rwdProlog1.1	B-Prolog
reducer (orig)	250	220
reducer(2)	230	—

Table 7: Avoiding term construction during unification for reducer

In the second place, we suspected unification factoring to play some role in the loss of performance by dProlog: quite a bit of unification factoring could be done for instance on the `t_redex` predicate (but also on others) and we assumed that TOAM does this (based on [17]). Apparently, this is not implemented in B-Prolog.

6 Backtracking and B-Prolog

In section 7 we will discuss deterministic programs. Here we deal with programs - or at least predicates - which create choicepoints and do backtrack.

Within the B-Prolog paradigm of passing parameters through the stack, there are actually three issues that are different from WAM⁴:

1. parameters are pushed on the control stack *once* for a predicate call and reused by each clause of the predicate
2. an environment is pushed on the control stack *once* for a predicate call and reused by each clause of the predicate
3. depending on the determinacy of a predicate, there can be a different choicepoint - which partly overlaps with the environment

Note that we could have added a similarity between B-Prolog and WAM: both create (at most) one choicepoint for a predicate and share it amongst all clauses.⁵

The overlapping part of point 3 resembles [11], but we'll ignore point 3 completely, mostly because it is clearly something that can be carried over to WAM as well and because shallow backtracking [4] was also based on different types of choicepoints.

As for points 1 and 2, the WAM moves arguments between a choicepoint and the argument registers for each clause, and environment allocation happens on a per clause basis. These differences can be shown quite easily by the following predicate `p/n` with its associated dProlog code:

<code>p(→,→,→,→) :- a,b.</code>	<code>try @1</code>	<code>@1:</code>	allocate, call a, deallocate, execute b
<code>p(→,→,→,→) :- a,b.</code>	<code>retry @2</code>	<code>@2:</code>	allocate, call a, deallocate, execute b
...	
<code>p(→,→,→,→) :- a,b.</code>	<code>retry @(m-1)</code>	<code>@(m-1):</code>	allocate, call a, deallocate, execute b
<code>p(→,→,→,→) :- a,b.</code>	<code>trust @m</code>	<code>@m:</code>	allocate, call a, deallocate, execute b

⁴we ignore indexing issues because they are orthogonal

⁵double level indexing is not considered an option

Assuming a call to p/n and that the whole search tree is explored, and restricting the attention to the p/n predicate, the number of moves between choicepoint and argument registers is $n * m$ and the number of allocates is m . One should add to that the initial n moves for filling the argument registers. In B-Prolog, there is just the pushing on the stack of the arguments - n moves - and one allocate for the whole procedure a/n . It is clear that by making m and n large enough, a plain WAM implementation will be arbitrarily worse than B-Prolog. Note that this is only possible when the arguments are largely ignored in the clauses, something that is not frequent in hand-written code, but might be in generated code.

Let's first discuss the one combined environment-choicepoint of B-Prolog: sharing this frame is to some extent unavoidable in the context of B-Prolog; one particular clause could be activated after indexing has decided that this is the only clause, or after a full choicepoint is created - e.g. because indexing found a free incoming argument. It follows that either there must be specialized code for each way of selecting the clause, or that all clauses must have the same frame, because the offsets of the arguments from the current TOS (or whatever B-Prolog uses) must be the same. However, this is only true within the single stack paradigm, as the original WAM follows. In a double stack paradigm, the arguments would be pushed on the environment stack - not on the choicepoint stack. Within this paradigm, it would now be possible to keep the usual flexibility on whether to create an environment on a per clause basis.

To get rid of the multiple allocation of environments for different clauses, a simple optimization in WAM would consist in putting one allocate in front of the try-retry-trust sequence and removing the allocate from the clauses. Doing this is straightforward and results in a nice speedup. However, it is not an entirely satisfactory solution as some WAM invariants are broken and in particular garbage collection can get in trouble. Also, this pushing allocate before the try is not readily possible in a classical WAM which has one stack for both environments and choicepoints. Although in WAM variants that have separate stacks for environments and choicepoints, it is possible, it is complicated in dProlog by the fact that environments in dProlog are *upside-down*, so one needs to allocate environment large enough to hold all the permanent variables of the clause which has most.

There exists another solution for WAM implementation that compile body disjunctions *in-line* (see for instance [13]): both XSB and Yap for instance compile disjunctions in-line - and dProlog of course inherits this from XSB. Below we show the transformed predicate p/n and its associated code:

$p(_,_,\dots,_)$	<code>:-</code>		allocate, trymeorelse @1
	<code>a,b</code>		call a, deallocate, execute b
	<code>;</code>	<code>@1:</code>	retrymeorelse @2
	<code>a,b</code>		call a, deallocate, execute b
	<code>;</code>	<code>@2:</code>	
	<code>...</code>	<code>...</code>	<code>...</code>
	<code>;</code>	<code>@(m-1):</code>	retrymeorelse @m
	<code>a,b</code>		call a, deallocate, execute b
	<code>;</code>	<code>@m:</code>	trustmeorelse
	<code>a,b.</code>		call a, deallocate, execute b

We name this the disjunction transformation (for lack of a better name). It is clear that we benefit from this transformation in two ways:

1. one predicate activation results in the execution of only one allocate

2. the choicepoint created by the try-me-or-else instruction contains no parameters, so there is no moving between the choicepoint and argument registers at all

There is a price in this transformation of course: arguments must be moved to the environment. But this happens only once and this does not add arbitrarily to the cost of the execution.

So, the good news is that no hybrid schema as hinted at in [17] is necessary, as the benefit of passing arguments through the stack can be obtained largely from a simple program transformation which requires no changes in the execution model of the WAM. The decision to transform a predicate can be taken locally: no global information is needed. Global knowledge can improve the situation because then the caller would avoid filling the argument registers all together.

The bad news is that for this transformation to be effective, the underlying system must compile body disjunctions *in-line*. Some systems (SICStus and B-Prolog for example) don't do that. Also, in order not to lose indexing, one should combine indexing instructions with the compilation of disjunctions: this was already hinted at in a different context in [14].

Finally, let us point at the analogy between BinProlog principle of binarization ([12]) and B-Prolog's principle of pushing only one environment per predicate invocation. Take the clause:

$$p(P) \text{ :- } q(Q), r(R), s(S), t(T).$$

which BinProlog binarizes to

$$p(P, \text{Cont}) \text{ :- } q(Q, r(R, s(S, t(T, \text{Cont}))))).$$

In this clause, the term $r(R, s(S, t(T, \text{Cont})))$ acts like the frame from which each clause for q will read its arguments. BinProlog could have gone one more step on top of binarization and also *unaryze* all predicates, that is transform each predicate to a predicate with just one argument. It would then also have the single frame property as B-Prolog and with the same disadvantage as the above suggested transformation in the WAM context. Also note that by binarization, performance can become arbitrarily better and worse (see [8]), because binarization also performs some actions early which might be good if there is backtracking, or bad if there is early failure.

To end this section, we give some measurements on a set of artificial programs. Basically, a predicate `a5/5` is called repeatedly. Programs `nbt0`, `nbt2` differ from `nbt1`, `nbt3` in that `nbt1` and `nbt3` are obtained by applying the disjunction transformation. `nbt0` differs from `nbt2` in that `nbt0` does not need a WAM environment in `a5/5`. Sample code for `a5/5` is given.

	original program	disjunction transformation applied
no environment in original	nbt0 <code>a5(-,-,-,-,-) :- fail.</code> <code>a5(-,-,-,-,-) :- fail.</code> ... <code>a5(-,-,-,-,-) :- fail.</code> <code>a5(-,-,-,-,-) :- fail.</code>	nbt1 <code>a5(-,-,-,-,-) :- fail</code> ; fail ... ; fail ; fail.
with environment in original	nbt2 <code>a5(-,-,-,-,-) :- fail, dummy, dummy.</code> <code>a5(-,-,-,-,-) :- fail, dummy, dummy.</code> ... <code>a5(-,-,-,-,-) :- fail, dummy, dummy.</code> <code>a5(-,-,-,-,-) :- fail, dummy, dummy.</code>	nbt3 <code>a5(-,-,-,-,-) :- fail, dummy, dummy</code> ; fail, dummy, dummy ... ; fail, dummy, dummy ; fail, dummy, dummy.

	rwdProlog1.1	B-Prolog
nbt0	33230	29230
nbt1	21820	30400
nbt2	44350	29210
nbt3	21770	30410

Table 8: Results on different backtracking programs

The disjunction transformation hardly affects B-Prolog, while it turns rwdProlog1.1 from about 50% slower (in the worst case - nbt2) to 25% faster than B-Prolog.

Not surprisingly, Yap also benefits from the disjunction transformation. Note that one needs to construct slightly more complicated example programs to show that this transformation does not always benefit implementations that have no in-line compilation of disjunction, like SICStus Prolog. On the other hand, the transformation does not harm in such implementations more than in others.

7 The cost of last call optimization in B-Prolog

In deterministic mode, B-Prolog can possibly reuse the arguments passed on the stack for the *last* call in a clause body: this is dual to the WAM which can reuse the arguments in the *first* goal of a body. A more detailed analysis - see section 8 - shows that neither is better than the other by design as far as argument reuse is concerned. Still B-Prolog pays a heavy price for reuse when the tail call and the head do not have the same arity. And even if the tail call has the same arity, B-Prolog seems to perform worse than a priori expected. The following table shows this: there are 4 small programs which do tail calls of the same and of different arity, and which do or do not require a WAM environment. All are deterministic. The timings are shown for dProlog1.1 (single and double emulator) and B-Prolog. Under dProlog, the programs were compiled with the instruction compressions *collapse_deallocates*, *collapse_unifies* and *collapse_getlist_unifies* switched off⁶.

	no WAM environment			with WAM environment		
different arity	a12([_—R]) :- b21(1,R). b21(—,[_—R]) :- a12(R).			da12([_—R]) :- dummy, db21(1,R). db21(—,[_—R]) :- dummy, da12(R).		
	dProlog1.1	rwdProlog1.1	B-Prolog	dProlog1.1	rwdProlog1.1	B-Prolog
	47920	45280	94590	92270	87130	131250
same arity	a22(—,[_—R]) :- b22(1,R). b22(—,[_—R]) :- a22(1,R).			da22(—,[_—R]) :- dummy, db22(1,R). db22(—,[_—R]) :- dummy, da22(1,R).		
	dProlog1.1	rwdProlog1.1	B-Prolog	dProlog1.1	rwdProlog1.1	B-Prolog
	53230	50580	74490	98660	92430	109520

These results show clearly that B-Prolog pays quite a hefty price for its argument passing method. To be fair: not all the performance loss comes from the argument passing convention. B-Prolog checks after all head unifications whether delayed goals must be woken. This accounts for some overhead.

⁶because B-Prolog seemed not to do any compression

Note that we now have a schema for making B-Prolog preform arbitrarily worse than WAM. Consider indeed the following predicates:

$$\begin{aligned} \mathbf{a}([_|\mathbf{R}], A_2, A_3, \dots, A_n) & :- \mathbf{b}(\mathbf{R}, A_2, A_3, \dots, A_{n-1}). \\ \mathbf{b}([_|\mathbf{R}], A_2, A_3, \dots, A_{n-1}) & :- \mathbf{a}(\mathbf{R}, A_2, A_3, \dots, A_{n-1}, \mathbf{x}). \end{aligned}$$

In order to reuse the arguments, B-Prolog needs to perform in the order of n moves for each execute instruction, while the WAM does not. The example also shows that passing arguments through the stack is performance wise highly unstable with respect to unfolding.

It is worth showing for some of the programs above part of the code: the next table shows the code for the predicates a12/1 and da22/2 in both systems.

predicate	dProlog	B-Prolog
a12/1	<pre> getlist A1 unitvar A1 unitvar A2 putint 1 A1 execute b21/2 </pre>	<pre> allocate_flat(1,4) unify_list_y(1) unify_arg_void_one unify_arg_vx(2) jmpn_det_get_ar_cps((a12,1,2)) move_int_y(1,1) move_y_ux(0,2) put_ar_cps(-1) execute((b21,2)) label((a12,1,2)) * para_int(1) * para_ux(2) * call((b21,2)) * return_b </pre>
da22/2	<pre> allocate 3 getlist A2 unitvar A1 unipvar Y2 call dummy/0 putint 1 A1 putdval Y2 A2 deallocate execute db22/2 </pre>	<pre> allocate_flat(2,4) unify_list_y(1) unify_arg_void_one unify_arg_vy(1) call((dummy,0)) jmpn_det((da22,2,2)) move_int_y(2,1) execute((db22,2)) label((da22,2,2)) * para_int(1) * para_uy(1) * call((db22,2)) * return_b </pre>

The instructions marked with (*) are not executed during the benchmark, but they belong to the code for the predicates. The code for da22 shows clearly that the determinacy test which WAM performs in the deallocate instruction is replaced in B-Prolog by the new instruction jmpn_det. Also B-Prolog must generate separate code for the deterministic and non-deterministic case for the arguments of the last goal. This leads to an increase in code size. We have tried to get figures for this increase, but first of all, code size as given by statistics/0, includes non-code related stuff, and secondly, B-Prolog uses words for items for which other systems use bytes or shorts: comparison is therefor difficult.

8 A more detailed analysis of argument reuse

Take a clause like $a(\overline{X}) :- b(\overline{Y}), c(\overline{Z})$. in which \overline{X} , \overline{Y} and \overline{Z} denote possibly a series of arguments - variables or other terms.

First thing to note: WAM can *always* reuse argument registers in the first goal - modulo the concrete form of the arguments \overline{X} and \overline{Y} of course. *Always* means here that this property is independent of the determinacy of the a and b predicates.

In contrast, B-Prolog can reuse in the c-goal parameters from a, only if both the following properties are true:

1. there is no choicepoint for a at the moment c is called
2. the call to b has left no choicepoints

This means that for an activation of a particular clause, the last goal of the clause can reuse parameters on the stack at most once and if the last goal is activated n times, the first (n-1) times such reuse is not possible. Note that the first property must hold even if the goal b would not have been there. This means that the B-Prolog principle for binary clauses is particularly ineffective, as shown by the predicate:

```
a(A,B,C,D,E) :- b(A,B,C,D,E).  
a(A,B,C,D,E) :- b(A,B,C,D,E).  
...  
a(A,B,C,D,E) :- b(A,B,C,D,E).  
a(A,B,C,D,E) :- b(A,B,C,D,E).
```

for which B-Prolog does no argument reuse until the call to b/5 in the last clause, resulting in (almost) as much argument moving as in the WAM. A small benchmark based on the above clauses shows as a result:

	rwdProlog1.1	B-Prolog
morenbt0	49990	110780

Table 9: Binary clauses and argument passing

Applying the transformation as in 6 is bad for dProlog - although it still performs better than B-Prolog. All variables become permanent ones and they are moved repeatedly to the argument registers. This results in the same number of moves as in the original program (between the choicepoint and the argument registers) but moving the permanent variables happens by a putpval instruction (of which there are several) instead of by a tight loop within the a retry or trust, so in an emulator setting, this is clearly bad. As expected, B-Prolog does not suffer in this particular example from the transformation.

One conclusion is that due to its argument passing schema, B-Prolog must execute more abstract instructions: in an emulator, this means more emulator cycles which leads inevitably to a lower performance. This might not be true for native code generation or translation to C.

9 More on instruction compression

At some point we realised that a more systematic approach to instruction compression was needed. We therefore enhanced dProlog1.1 with a (C) compilation option that results in a dump - after

a session - of a file with a frequency count of all executed instructions and also of all pairs of executed instructions. This has been most helpful in deciding which compressions to do. We have been looking only at sequences of length 2 but we have done also 3-compressions, because of the following considerations:

- a 2-compression sometimes suggests a 3-compression, especially when the 3-compression takes the same space as the 2-compression
- B-Prolog does some 3-compressions and even 4-compressions

As our configuration of choice of dProlog is `heap_vars`, we performed our instruction compressions there at first, only later worrying about the possibility of doing them in e.g. `wam_vars`. Also, since we are involved in ilProlog [2], we wanted to perform the same compressions there. However, recently ilProlog became independent of the XSB compiler, i.e. it now uses a *better* compiler designed and implemented by Henk Vandecasteele. It turned out that useful instruction compressions in non-`heap_vars` mode or with the new ilProlog compiler, were less easy to find. The lesson we learned is not really a surprise: if the basic compiler has more instructions, or if the compiler optimizes *too much*, there is less opportunity for effective instruction compression. The reason is in the respective case:

- as an example, take the `putpva*` compression: while `heap_vars` dProlog only distinguishes between `val` and `var`, the `wam_vars` version also must distinguish the unsafe value; this means that the frequency of the combinations will be lower and/or that there will be more new instructions introduced by the compressions (XSB actually also distinguishes between variables that it wants to dereference or not - this distinction is highly questionable; dProlog treats them the same: no dereferencing takes place)
- a better optimizing compiler will not generate long sequences of e.g. `getpvar` instructions, e.g. because it will optimize for *early failure*; so the compressible sequences will have a smaller frequency and consequently a smaller impact and/or there will be more compressible sequences, so that more implementation effort is needed to do them all

To end this section, we take a quick look at some of the statistics that our system spat out for the `comp` benchmark which is an older trimmed down version of the compiler compiling itself.

Before doing some compressions, of the 100M instructions executed about 14M were `putdval` instructions. And 6.8% of the 2-combinations were `getpvar`, `getpvar`. After performing some compressions (and running the compiler on a different input) 2.6M out of 77.5M 2-combinations were `getpbreg`, `trymeorelse`. Such dynamic statistics clearly indicate what can be optimized by instruction compression. Note that static frequency counts only give a rough approximation of the dynamically obtained ones.

One instruction compression that seems particularly effective for B-Prolog and which is used systematically in its implementation is the compression of the `fork` and a `unify` instruction, or in terms of the WAM of a `retry` instruction and a `get` instruction⁷. In B-Prolog, the thought of doing such a compression is quite natural, the `fork` instruction does nothing more than filling out a new alternative in the choicepoint. Such a compression is of course also potentially useful in the WAM: the compressed instruction should first attempt the unification - setting `S` - and if that fails just jump to the alternative instead of going through a heavy general fail operation.

⁷these are not equal, but close enough in spirit

10 The delay mechanism of B-Prolog

Packaged together with its new parameter passing mechanism, B-Prolog also has a new way of implementing delayed goals. While the mechanism of [3] uses heap terms - and thus resembles closures in lazy functional language implementations - B-Prolog uses the frame (arguments+environment+choicepoint) on the control stack to keep the delayed goal. One of the advantages is that reactivating this goal is cheap since no arguments need to be moved to the argument registers, no symbol table lookup is needed⁸ and finally, delaying the same goal immediately again (because some condition for wake-up was not fulfilled) is damn cheap. The latter re-delaying operation could also be made *constant time* in the heap schema, but it requires more low-level support than implementations seem to care for. On the whole, the empirical comparison of both methods seems not to have been made properly⁹: from [18] it was not clear whether the obtained performance gain was due to the overall design of B-Prolog or particularly due to its delaying mechanism. We do not attempt such an evaluation here either, but the disjunction transformation shown in section 6 suggests a way of getting a similar implementation of delayed goals, at almost the same price - both performance and implementation wise. Suppose we have the following code for a predicate with a block declaration:

```
:- block p(-,?).
```

```
p(t1,t2) :- asd.
```

```
p(t3,t4) :- qwe.
```

then first transform it to:

```
p(X,Y) :-
    label(L),
    (var(X) ->
        trigger(inst(X),label(L)),
        freeze_current
    );
    '$p'(X,Y)
).
```

```
'$p'(t1,t2) :- asd.
```

```
'$p'(t3,t4) :- qwe.
```

In this code the meaning of the predicates `label/1`, `trigger/2` and `freeze_current/0` are intuitively clear, but one must take into account that a similar mechanism as in B-Prolog is needed for delaying goals, freezing the stack (XSB terminology) and to chain delayed goals.

One can see an analogy between on the one hand the implementation of delay as in B-Prolog or by a copying-to-the-heap approach, and on the other hand the implementation of suspending consumers in a tabling Prolog by freezing the stacks (as XSB does) or by copying their relevant parts as in CAT [6]. Freezing stacks requires an extra freeze register - the SF register from B-Prolog was introduced for such purpose - unless one is lucky and can apply a trick like in CHAT [7] and can have an overall impact on performance and the implementation, while a copying approach is more orthogonal to the rest of the machinery. Also, freezing the stack might necessitate a control stack compactor because unneeded environments cannot be deallocated easily. E.g. in the code:

⁸this can also be avoided in the setting of [3] of course

⁹this goes back to a conversation with K. Sagonas

```
a :- b, dummy.  
b :- freeze(X,foo(X)).
```

```
?- a,a,a,a,a,a,a, ... , a.
```

The frame for `a/0` cannot be recovered easily and B-Prolog indeed overflows quickly the control stack (for a variant of this example) while in SICStus, frozen goals reside on the heap for which there is a collector already.

Note that the above source transformation is more or less what B-Prolog does at a lower level. Also calls to `freeze/2` are treated similarly.

11 Abstract machine code for WAM emulators

Of course we mean the WAM in the broad sense and our advice concerns performance only.

1. keep the instruction set small - this means for instance that specializing instructions for void is not always good; neither is inventing new instructions for dealing with the cut
2. do not optimize the code (too much) - this is in accordance with [5] which also claims that sophisticated compiler optimizations are not needed to obtain good performance
3. critical builtins should have the convention described earlier and/or be specialized; this includes arithmetic
4. now dynamically profile executions and then
 - specialize
 - compress

instructions sequences

The problem with profiling for instruction compression is that the amount of data might seem overwhelming when looking for 3-compressions: with 256 instructions, there are 16777216 3-combinations¹⁰. That shouldn't be prohibitive: current PC's are quite capable of handling this amount of data.

Finally, as we said in [9] also inter-predicate compressions should be looked at: maintaining optimizations that cross predicate boundaries is straightforward even in a dynamic context like Prolog. Inter-predicate compressions are one example of *long distance* compressions: others should also be tried out.

Of course indexing (including unification factoring) must be done right - however, we have not yet investigated this issue yet.

12 Conclusion

Our first conclusion is that passing the arguments to a goal through the stack does not give a clear performance advantage for plain Prolog programs. Even for non-deterministic programs, it is not a clear win, especially since a simple source to source transformation can give most of the benefit of B-Prolog within the WAM. However, it has a clear advantage when the system implements delayed

¹⁰not all of them occur in the WAM; e.g. two try instructions should never follow each other dynamically

goals, but even this must be put to scrutiny, since the method described in [18] can be adapted to the WAM. For deterministic programs, passing the arguments through the stack seems to have only performance penalties: more abstract machine instructions must be executed and in particular environment shifting can be costly.

The second conclusion is that instruction compression is a far more effective way of improving the performance of an emulator than the radical change in the abstract machine which the argument passing mechanism of B-Prolog is.

Finally, the double emulator implementation of READ/WRITE propagation is on average worth its while, but this might depend on the cache of the underlying architecture. More detailed measurements and simulations are needed to confirm this. dProlog1.1 is probably the first system in which both a single emulator and the double emulators are implemented and compared, all other things kept equal. As such, this experience has proven valuable.

Acknowledgements

Part of this work was conducted while the first author was a guest at the Institut de Mathématiques Appliquées of the Université Catholique de l'Ouest of Angers, France. Sincere thanks for this hospitality. We are grateful also to the XSB team for the opportunity to use its compiler and to Neng-Fa Zhou for help in using B-Prolog and for making the sources of B-Prolog 4.0 available.

References

- [1] H. Ait-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. The MIT Press, Cambridge, Massachusetts, 1991. See also: <http://www.isg.sfu.ca/~hak/documents/wam.html>.
- [2] H. Blockeel, L. Dehaspe, B. Demoen, G. Janssens, J. Ramon, H. Vandecasteele. Executing Query Packs in ILP Proceedings of ILP2000 - 10th International Conference on Inductive Logic Programming, July 2000, London
- [3] M. Carlsson. *Freeze, Indexing and Other Implementation Issues in the WAM*. Proceedings of the 4th International Conference on Logic Programming, Melbourne, 1987, pp 40-58
- [4] M. Carlsson. *On the Efficiency of Optimising Shallow Backtracking in Compiled Prolog* Proceedings of the Sixth International Conference on Logic Programming, pp. 3-16, MIT Press, 1989
- [5] V. S. Costa. *Optimising Bytecode Emulation for Prolog*. Proceedings of PPDP'99, LNCS 1702, Springer-Verlag, 261-277, September, 1999. See also <http://www.ncc.up.pt/~vsc/Yap/>.
- [6] B. Demoen, K. Sagonas. *CAT: the Copying Approach to Tabling*. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Principles of Declarative Programming, 10th International Symposium, PLILP'98, Held Jointly with the 6th International Conference, ALP'98*, number 1490 in LNCS, pages 21–35, Pisa, Italy, Sept. 1998. Springer.
- [7] B. Demoen, K. Sagonas. *CHAT: the Copy-Hybrid Approach to Tabling*. Future Generation Computer Systems, Vol 16, Iss 7, May 2000, 0167-739X
- [8] B. Demoen, A. Mariën. Implementation of Prolog as binary definite Programs Proceedings of the second Russian Conference on Logic Programming, Lecture Notes in Artificial Intelligence, 592, p. 165-176, March 1992

- [9] B. Demoen, P.-L. Nguyen. *So many WAM variations, so little time*. Computational Logic - CL2000, First International Conference, London, UK, July 2000, Proceedings (V. John Lloyd, ed.), Lecture Notes in Artificial Intelligence, vol. 1861, Springer, 2000, pp. 1240-1254.
- [10] B. Demoen, P.-L. Nguyen. *Experiments in WAM emulators and term representations*. CW report 283, Januari 2000 <http://www.cs.kuleuven.ac.be/publicaties/rapporten/cw/CW283.abs.html>
- [11] A. Mariën, B. Demoen. *On the management of E and B in WAM*. Proceedings of North American Conference on Logic Programming, Cleveland, Ohio, oct 1989, p. 1030-1047
- [12] P. Tarau. *Program Transformations and WAM-support for the Compilation of Definite Metaprograms*. In A. Voronkov, editor, Logic Programming, RCLP Proceedings, number 592 in Lecture Notes in Artificial Intelligence, pages 462-473, Berlin, Heidelberg, 1992. Springer-Verlag.
- [13] H. Vandecasteele, B. Demoen, G. Janssens. *Compiling large disjunctions* Proceedings of the Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages at CL2000, July 24, 2000, London
- [14] R. Venken, B. Demoen. A partial evaluation system for prolog: theoretical and practical considerations Workshop on Partial Evaluation and Mixed Computation 18-24 Oct 1987 Ebberup, Fyn Denmark - also in New Generation Computing Vol. 6 Nos. 2,3 1988 (279-290)
- [15] D. H. D. Warren. *An Abstract Prolog Instruction Set*. Technical Report 309, SRI International, Menlo Park, U.S.A., Oct. 1983.
- [16] See <http://www.cs.sunysb.edu/~sbprolog/>.
- [17] N.-F. Zhou. "On the Scheme of Passing Arguments in Stack Frames for Prolog" Proc. Eleventh International Conference on Logic Programming, MIT Press, pp.159-174, 1994 See also <http://www.sci.brooklyn.cuny.edu/~zhou/bprolog.html>
- [18] N.-F. Zhou. A Novel Implementation Method for Delay Joint Internatinal Conference and Symposium on Logic Programming, pp.97-111, MIT Press, 1996