Delay in the WAM and the TOAM: redoing a ten year old experiment

Bart Demoen, Phuong-Lan Nguyen

Report CW 454, April 2006

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

Delay in the WAM and the TOAM: redoing a ten year old experiment

Bart Demoen, Phuong-Lan Nguyen

Report CW 454, April 2006

Department of Computer Science, K.U.Leuven

Abstract

A 10 year old experiment related to *freeze/2* and its implementation in B-Prolog and SICStus Prolog is redone. B-Prolog uses the TOAM abstract machine and its implementation of delayed goals is by means of suspension frames on the execution stack. The traditional WAM approach - partly established by SICStus Prolog implementors - is to put suspension terms on the heap. Earlier experiments comparing the two approaches/systems indicated a clear edge for the B-Prolog approach. We first redo these experiments and show to what extent the old conclusions still hold after 10 years. Subsequently, we use hProlog as an environment for showing to what extent the traditional WAM approach to freeze/2 can be made competitive to the B-Prolog approach. We do the same for the event mechanism of B-Prolog.

Delay in the WAM and the TOAM: redoing a ten year old experiment

Bart Demoen* and Phuong-Lan Nguyen†

* Dept. of Computer Science, K.U.Leuven, Belgium
bmd@cs.kuleuven.be

† Inst. de Mathématiques Appliquées, UCO, Angers, France
nguyen@uco.fr

Abstract. A 10 year old experiment related to *freeze/2* and its implementation in B-Prolog and SICStus Prolog is redone. B-Prolog uses the TOAM abstract machine and its implementation of delayed goals is by means of suspension frames on the execution stack. The traditional WAM approach - partly established by SICStus Prolog implementors - is to put suspension terms on the heap. Earlier experiments comparing the two approaches/systems indicated a clear edge for the B-Prolog approach. We first redo these experiments and show to what extent the old conclusions still hold after 10 years. Subsequently, we use hProlog as an environment for showing to what extent the traditional WAM approach to freeze/2 can be made competitive to the B-Prolog approach. We do the same for the event mechanism of B-Prolog.

1 Introduction

We assume working knowledge of Prolog and its implementation without further explanation. For a good introduction to Prolog see [4]; to the WAM, see [1, 9]; for B-Prolog and the TOAM, see [10]; the SICStus Prolog implementation is described in [3]. hProlog is the successor of dProlog [6]; it is available from the first author. The ultimate documentation of these systems is of course their source code. We have used the following releases of these systems: B-Prolog 6.9-2, SICStus Prolog 3.12.0, hProlog 2.6. The benchmarks were all run on Pentium, 1.8GHz, under Debian, and while running the benchmarks, garbage collection was disabled by starting the system with enough initial memory, whenever possible. We also used SWI-Prolog Version 5.5.40 and Yap-4.5.7.

The predicate *freeze/2* has its origin in Prolog II as *geler/2*. [2] is the first publication of its implementation in the context of the WAM. Later implementations use meta-structures [8] or attributed variables [7], and more recently [11] uses delay clauses for the implementation of freeze/2. Delay clauses have since then been replaced by *Action Rules*.

The issue of putting asleep and waking up goals is not restricted to freeze/2: the efficiency of constraint solvers depends crucially on the efficiency of the mechanism to put goals asleep (conditionally) and waking up those goals when

certain events occur. Instantiation is just one such event and it is related to the Herbrand solver. Other solvers have their own events, e.g. *upper bound changed* for a finite domain solver. However, when the other solver is embedded in an untyped context like Prolog, there is a big difference between the Herbrand event of instantiation, and events from the other solver: Herbrand instantiation of any variable (any in the sense of *possibly belonging to a non-Herbrand solver*) is supposed to be supported and the underlying Prolog system must implement freeze/2 for that. The other (non-Herbrand) events can easily be implemented **without** freeze/2: just use *passive* attributed variables to which any information can be attached and which do not react necessarily to Herbrand instantiation. Whether this difference is very important at the implementation level is a different issue, but it should be clear that the wakeup of a goal because an upper bound changed, has nothing to do with Prolog's unification routine.

In the 1996 paper [11], the author explains the use of delay clauses (now named action rules) for implementing delay - as done in B-Prolog. An experimental comparison is presented there as well: there are only a few benchmarks and the comparison is with SICStus Prolog. In other sciences (experimental physics, biology ...) it is quite common to redo experiments. In computer science however, it seems rather uncommon: we certainly didn't find new measurements on the implementation of freeze/2 in the WAM and the TOAM. So we have the following plan: (1) redo the experiments of [11] and check whether the conclusions put forward there are still valid; (2) use hProlog as a lab environment for studying how a traditional WAM approach to delay can be made more efficient. We will also study the *event* mechanism of B-Prolog which is implemented in the TOAM with the same stack based mechanism as freeze, and see how a heap based approach can be made to perform as good. For this, we will also use hProlog.

We start here by reproducing the ratios of timings reported in [11] and a verbatim quote of the conclusion in [11]: we have left out the absolute times and figures related to SICStus Prolog native code and to memory usage.

	block	freeze
nreverse	8.37	28.67
queens	2.22	13.87
sendmory	1.96	10.71
psort	2.31	13.26

Table 1. Execution time ratio SICStus/B-Prolog in 1996

1. *While using freeze and delay clauses does not cause much difference in execution time in B-Prolog, using block declaration is much faster than using freeze/2 in SICStus Prolog.*

2. *For the four programs, B-Prolog is significantly faster than SP-bc*[1] *[and ...]. The speed-ups are due mostly to the novel implementation method of delay adopted in B-Prolog. For the original nreverse program without delay, B-Prolog is only 45 percent faster than SP-bc.*
3. *It is difficult to tell to what extent delay affects the execution time because to do so we have to get rid of the time taken to run predicates that never delay. However, as more than 90 percent of the predicate calls in nreverse delay in execution, the ratios in the row for nreverse roughly tell us about the difference between the performance of the two systems.*

The first conclusion can very well objectively follow from the figures in that paper - the only criticism could be that only four benchmarks are used. However, [11] fails to investigate reasons for the difference between freeze/2 and block declarations in SICStus Prolog: we will get into this in Section 3.

The second conclusion attributes the speed-up to the novel implementation method of delay. We must put this conclusion to scrutiny. Also, even if the conclusion is justified, it does not need to follow that a traditional WAM based approach cannot be as efficient.

The last sentence in the third conclusion seems to mean that the basic performance of freeze (or block) compared to the performance of delay in B-Prolog, is about 28 (respectively 8); we will come back to this in Section 2.

Our motivation was originally only to redo the freeze/2 experiment of [11]. At some point we became also interested in the performance of the B-Prolog mechanism for other events posted by the user, because we were involved in the design of a compilation schema CHR to action rules that uses heavily this event mechanism: Sections 6 and 6.4 contain our experience with this.

2 The same experiments 10 years later

The benchmarks used in 1996 are available in the distribution of B-Prolog. It is a simple matter to rerun them, but since machines have become faster, it was necessary to run the benchmarks with larger input: for queens, it was 8 and now 10; for nrev, it was 100 and now 500; for psort, it was a list of length 15 and now 19. The results are in Table 2.

	block	freeze
nreverse	2.00	4.00
queens	2.26	14.10
sendmory	1.88	12.87
psort	1.84	18.67

Table 2. Execution time ratio SICStus/B-Prolog in 2006

How well do the conclusions of 1996 hold ?

[1] SICStus Prolog byte code - the emulator

1. Clearly, in SICStus Prolog, block declarations are still much faster than freeze/2.
2. No new data related to conclusion 2 are available at this point.
3. If the reasoning for conclusion 3 was correct in 1996, then in 2006, one must conclude that the TOAM implementation of delay is only 2 (for block) to 4 (freeze) times faster; so SICStus Prolog's implementation of block and freeze has caught up by a factor of 4 to 7 - that is quite amazing, but not impossible !

Table 2 is remarkable in two respects:

- the figures in the block column are all close to two
- the freeze figures for nreverse are very different from the other freeze figures which in turn are rather close

For the second point, it is natural to find in the benchmarks themselves a reason for this phenomenon.

The nreverse benchmark differs from the other three in the following respect: nreverse is deterministic, while the other three benchmarks find their solution(s) by backtracking during a labeling phase (queens, sendmory) or by generating permutations (psort). This means that while in nreverse each delayed goal is activated once, the delayed goals in the other three benchmarks are activated many times. This could account for the qualitative difference in the 2006 figures for nreverse and the other three. However, we will see in Section 3 that the explanation is totally different, but we need to dig deeper to find the truth.

3 The difference between freeze/2 and block declarations

In SICStus Prolog, freeze/2 is implemented basically as:

```
:- block freeze(-, ?).
freeze(_, Goal) :- call(Goal).
```

This means that the goal `freeze(X,foo(A,B,X,D,E)).` could have been specialized to the goal `foo_blocked(A,B,X,D,E).` with a definition of a new predicate

```
:- block foo_blocked(?,?,-,?,?).
foo_blocked(A,B,C,D,E) :- foo(A,B,C,D,E).
```

and since a block declaration is compiled to specialized instructions, this leads to great performance gains.

We name this specialization *block introduction*. The fact that SICStus Prolog doesn't do this specialization is part of the reason why there is such a big difference between block and freeze. It is in fact the whole reason for nreverse.

B-Prolog applies the above specialisation (mutatis mutandis of course) for calls to freeze/2. This explains why there is virtually no difference between the use of action rules, or freeze/2 in B-Prolog.

So why is the ration freeze/block so much bigger for the three backtracking benchmarks than for nreverse ?

The reason is *nested freezes*. The psort benchmark contains the following goal: *freeze(X,freeze(Y,X=< Y))*. The inner nested freeze/2 goal is executed at the moment that X becomes instantiated. This means that the query

```
?- L = [1,2,3,4,5],
   freeze(X,freeze(Y,X=<Y)), member(X,L), fail.
```

executes freeze/2 6 times.

A nesting of two freezes can be transformed away as follows: the goal *freeze(X,freeze(Y,goal(X,Y))* is transformed to the conjunction *NewGoal = newgoal(X,Y,_), freeze(X,NewGoal), freeze(Y,NewGoal)* with newgoal/2 defined as:

```
newgoal(X,Y,Z) :-
        (nonvar(X), nonvar(Y), var(Z) ->
              Z = 1,
              goal(X,Y)
        ;
              true
        ).
```

We name this transformation *unnesting*. the above query executes only two freeze goals when the nested freezes are unnested. The effect of unnesting the psort benchmark is very pronounced: the number of calls to freeze/2 goes down from 2621858 to 38. The sendmory and queens benchmarks contain similarly nested freezes and unnesting has a huge effect, as shown in Table 3 which shows execution times in milliseconds for SICStus Prolog. However, note that unnesting also naturally folds the manifest calls (see Section 4), so some performance gain comes from folding, albeit little for SICStus Prolog.

	block	unnest	ratio unnest/block	freeze
queens	480	980	2.04	2990
sendmory	480	1000	2.08	3270
psort	2200	4450	2.02	22280

Table 3. Block, freeze and unnested freeze compared in SICStus Prolog

It is now clear that the bad performance of freeze/2 in SICStus Prolog on these three benchmarks is due in the first place to not applying unnesting. Note that the ratio between the unnesting and block performance is now almost constant and equal to two.

B-Prolog does unnesting of nested freeze goals. So we can apply a small transformation to B-Prolog programs that hide the manifest nesting as follows: transform the goal *freeze(X,freeze(Y,goal(X,Y))* to *freeze(X,myfreeze(Y,goal(X,Y))* and define

```
myfreeze(X,Y) :- freeze(X,Y).
```

This reduces the performance of psort in B-Prolog by a factor of 1.9. This shows that unnesting is important for performance.

Of course, in an system with block declarations, unnesting should not be performed. It is much better to apply *block introduction* for nested freezes: the goal *freeze(X,freeze(Y,foo(Z))* is transformed to *foo(X,Y,Z)* and foo/3 is defined as

```
:- block foo(-,?,?), foo(?,-,?).
foo(_,_,Z) :- foo(Z).
```

with some obvious optimizations when X and/or Y is in the term Z syntactically.

3.1 A draft new conclusion

Since calls to freeze/2 are almost always manifest - i.e. at the source level, the second argument is not a variable - block introduction is almost always applicable and it does away easily with the performance difference for the benchmarks. When block is not available, unnesting smoothens out the differences between a WAM freeze implementation and a TOAM one.

The ratio B-Prolog/block is close to two for all four benchmarks. Given that B-Prolog is about 50% faster than SICStus Prolog for plain Prolog code, this could indicate that there is a performance edge in action rules: we tend to be cautious about this conclusion.

4 Digging even deeper ...

Wiser because of the experience of block versus freeze, we have looked more closely at the effect of the compilation and execution process itself on the performance. From now on, we will use hProlog for comparison with B-Prolog because it is easier for us to perform implementation experiments with hProlog than with SICStus Prolog. But we believe that the results we will obtain in the hProlog context are of value in any WAM implementation. We first show in Table 4 the benchmark results for B-Prolog and hProlog without any changes to the benchmarks or the hProlog implementation. hProlog has only freeze/2 and no block declarations: freeze/2 is build on top of (dynamic) attributed variables [5].

	B-Prolog	hProlog
nreverse	40	102
queens	212	311
sendmory	254	769
psort	1193	4447

Table 4. B-Prolog and hProlog initially

From these figures, one would perhaps conclude (once more) that the TOAM way superior to the WAM way. However ...

- The hProlog implementation of attributed variables (see [5]) uses one slot (a heap word) for all attributes: in general this slot contains a list pointer to a list with pairs module-attribute value. Freeze/2 is not treated in any special way and consequently, freeze/2 suffers because of this generality: (1) when two attributed variables are bound, the underlying C implementation doesn't know about the special meaning of the freeze attribute and the freeze handler written in Prolog takes care of all actions; (2) when a frozen variable is instantiated, the general handler is called. These general actions result in performance loss. If the implementation has a special freeze slot in the attributed variable, this overhead can be avoided to a large extent. We will refer to this as *spec_slot*. In the context of hProlog, it means that we use the unique attribute slot only for freeze/2. Low-level built-ins deal with this slot, but no further knowledge about freeze/2 is pushed deeper in the implementation. These low-level built-ins existed prior to working on this paper. In order to use the spec_slot freeze/2 version, the user simple does *:- use_module(library(freeze)).* which overrides the general implementation.
- Without special treatment by the compiler, the goal *freeze(X,Z is X+Y)* is treated similarly to the conjunction *Goal = Z is X+Y, freeze(X,Goal)*: the goal is constructed as a heap term and when X becomes instantiated, a meta-call of Goal is executed. So no advantage is taken of the fact that the Goal is manifest. In systems with a slow meta-call[2], this is a serious drawback. And when arithmetic is meta-called, it is also executed slower than a compiled version would be. To be more specific, the following is often more efficient: rewrite the manifest goal above as *freeze(X,newp(X,Y,Z))* and define the new predicate *newp(X,Y,Z) :- Z is X+Y*. We will refer to this as *fold*. Neither queens nor nreverse contain fold-able freezes.
- We have discussed previously already unnesting: nreverse does not benefit from it.

Table 5 shows the benchmarks with unnesting, folding and special slot. The table does not show unnest and fold figures for B-Prolog: that would not be relevant to the point we want to make. Also, B-Prolog performs unnesting and fold already[3] and applying these transformations at the source level only introduces overhead for B-Prolog.

[2] Not hProlog !

[3] Perhaps the word *inlining* would be more appropriate instead of fold in the case of B-Prolog.

		B-Prolog	hProlog	hProlog spec_slot
nreverse	orig	40	102	31
send	orig	254	796	548
	fold		453	222
	unnest		439	228
psort	orig	1193	4447	2206
	fold		3366	1331
	unnest		2352	1105
queens	orig	212	311	175
	unnest		224	176

Table 5.

4.1 An intermediate conclusion

The biggest gain clearly comes from using a dedicated slot for freeze/2. Big gains can also come from folding: the extent of the effect might be hProlog specific, as we have measured it to be much smaller in SICStus Prolog.

Our conclusion is that the TOAM way to delay goals and the WAM way, can achieve similar performance: the edge that hProlog has (for unnest), is partly explained by the fact that B-Prolog is a bit slower on ordinary Prolog code. In any case, from these benchmarks we can't conclude too much about the basic mechanism. A clearer picture will emerge after new and artificial experiments in the next section.

5 New artificial experiments

The new set of artificial benchmarks is aiming at measuring the cost of freeze/2 in three orthogonal ways:

1. there is a cost in freezing and a cost in waking up: Tables 6 and 7 report on freezing, Tables 8 and 9 report on wakeup
2. the same variable can be frozen several times, and later become instantiated, or one can freeze several variables and then instantiate them during one atomic unification; the former is reported on in Tables 6 and 8, while the latter is reported on in Tables 7 and 9
3. the arity of the goal to be frozen and later woken up is important, because both in the WAM and in the TOAM, there is code to be executed for each argument; the tables report timings for a goal with arity 1 and arity 4, separated by a /

We have always subtracted the time of dummy loops. The figures for Yap, SICStus Prolog and SWI-Prolog are mainly given to be able to place the B-Prolog and hProlog figures in a wider performance context.

	B-Prolog	hProlog	hProlog-specslot	SICStus	SWI	Yap
1 var 1 goal	109 / 128	59 / 84	44 / 57	188 / 195	380 / 405	528 / 591
2 vars 1 goal	261 / 310	211 / 278	164 / 238	418 / 485	1100 / 1165	1055 / 1198
3 vars 1 goal	346 / 425	267 / 386	196 / 328	588 / 773	1791 / 1960	1568 / 1795
4 vars 1 goal	433 / 524	340 / 526	240 / 436	758 / 970	2500 / 2660	2113 / 2404

Table 6. Freezing: different variables on one goal each

	B-Prolog	hProlog	hProlog-specslot	SICStus	SWI	Yap
1 var 1 goal	109 / 128	59 / 84	44 / 57	188 / 195	380 / 405	528 / 591
1 var 2 goals	235 / 291	281 / 384	139 / 249	368 / 475	985 / 1071	927 / 1101
1 var 3 goals	328 / 418	421 / 545	214 / 323	528 / 668	1610 / 1735	1289 / 1557
1 var 4 goals	444 / 525	567 / 768	230 / 479	678 / 853	2237 / 2365	1601 / 2030

Table 7. Freezing: the same variable on one to four goals

5.1 More intermediate conclusions

The following observations were most striking to us:

- Yap is a high speed Prolog system; however, for these freeze related benchmarks, it performs rather slow between SICStus Prolog and SWI-Prolog; it is relatively better when more goals are woken from the same variable - i.e. a conjunction is woken - but we are inclined to ascribe that to the fact that Yap does not follow ISO for meta-call
- B-Prolog is less sensitive than any other system to the arity of the frozen goal when the freeze actually takes place; when the goal is woken, B-Prolog is amongst the systems most sensitive to the arity of the woken goal
- using the special slot in hProlog again really pays off: the gain is huge and hProlog-spec_slot is systematically the best of all

The most important conclusion seems to be that *yes, the traditional WAM approach can compete favourably with the TOAM approach, if the special freeze slot is used.*

Two more points were important for achieving hProlog's performance:

- when multiple goals are frozen on a variable - say P and Q - one could be inclined to just put them in a conjunction P, Q and later meta-call this conjunction; because of the particular ISO semantics of meta-call, it is more efficient to make a datastructure like $conj(P, Q)$ and add a definition *conj(P,Q) :- call(P), call(Q).*; this also make sure that cuts do not escape their scope[4]
- when one atomic unification instantiates just one frozen variable (resp. two), not the general handler is called, but a specialized version which *knows* there is only one goal (resp. two) to call

[4] This is not standardized, but implementations seem to adhere to that.

	B-Prolog	hProlog	hProlog-specslot	SICStus	SWI	Yap
1 var 1 goal	244 / 253	695 / 687	173 / 172	1098 / 1110	2305 / 2355	1871 / 1883
2 vars 1 goal	539 / 585	1158 / 1165	360 / 361	2095 / 2145	4762 / 4848	3206 / 3243
3 vars 1 goal	806 / 948	1972 / 1982	646 / 655	3103 / 3180	7162 / 7257	4628 / 4683
4 vars 1 goal	1129 / 1332	2532 / 2556	781 / 787	4095 / 4200	9606 / 9700	6077 / 6195

Table 8. Waking up: different variables on one goal each

	B-Prolog	hProlog	hProlog-specslot	SICStus	SWI	Yap
1 var 1 goal	244 / 253	695 / 687	173 / 172	1098 / 1110	2305 / 2355	1871 / 1883
1 var 2 goals	432 / 575	809 / 804	280 / 284	1893 / 1955	3615 / 3687	2323 / 2343
1 var 3 goals	612 / 744	884 / 875	346 / 351	2753 / 2823	4307 / 4413	2645 / 2676
1 var 4 goals	803 / 985	976 / 981	436 / 435	3543 / 3638	4955 / 5087	3064 / 3089

Table 9. Waking up: the same variable with one to four goals

6 Events: an example of multiple wakeups

Some issues of delaying goals are not tested by our benchmark set up to now. In particular, we have only dealt with one solver event: Herbrand instantiation. The most distinguishing fact about Herbrand instantiation is that - in the absence of backtracking - it can happen only once for a particular variable, while other solver events (e.g. bounds changed) can happen many times for the same finite domain variable. Non Herbrand solver events are more important for constraint solvers and a comparison between WAM and TOAM along that line is in order. B-Prolog has another event - not related to solvers in the traditional sense. We give just a small example with one action rule and a query:

```
p(X), {event(X,M)} => write(M).

?- p(X), post(event(X,hello)), post(event(X,world)).
```

This query results in the output *helloworld*.

Roughly, every time there is a call to post(event(X,M)) the bodies of the action rules that were called previously with X as the channel variable in the event, are executed.

An extra motivation for investigating this event mechanism is that we have been involved in the compilation of CHR programs to action rules using the event mechanism (see [12]): anticipating that this was going to be successful, we wanted to have an idea about the efficiency of an event mechanism in hProlog.

6.1 Events in the WAM

It takes little work to mimic this event behaviour in a Prolog system with attributed variables. We show this by an hProlog program (also working under SWI-Prolog) equivalent to the above example:

```
p(X) :- put_attr(X,event,p1(X)).

p1(X,M) :- write(M).

post(event(X,M)) :-
        get_attr(X,event,Goal),
        call(Goal,M).
```

A more general translation schema goes as follows:

```
h(Z), {event(X,Message)} => Body.
```

is translated to:

```
h(Z) :-
        (get_attr(X,event,A) ->
             put_attr(X,event,[h1(Z)|A])
        ;
             put_attr(X,event,[h1(Z)])
        ).

h1(Z,Message) :- Body.

post(event(X,Message)) :-
        get_attr(X,event,Goals),
        call_goals(Goals,Message).

call_goals([],_).
call_goals([G|Gs],Message) :-
        call_goals(Gs,Message),
        call(G,Message).
```

Two issues are not addressed by the above code:

- the matching (as opposed to unifying) semantics of a head in an action rule is not taken into account: hProlog does not (yet) have instructions for doing matching; however, this issue does not influence the benchmarks
- extra conditions in the action rule are not catered for: a small adaptation would; this issue does not influence the benchmarks either

6.2 Optimizing events in hProlog

The above code is not good enough to achieve performance comparable to what B-Prolog offers for its events. Here is an account of the extra work involved:

- B-Prolog executes goals in the chronological order in which they where installed; his is achieved above by the non-tail recursion in call_goals/2; in the actual implementation, we simply call reverse/2 first and then call a tail recursive version of call_goals/2; that is more efficient; reverse/2 was implemented at a low level in hProlog
- for hProlog, we have used a new low level attribute built-in, which amounts to using a dedicated attribute slot for the events and which renders the body of h(Z) in the example above to a single call to a built-in predicate
- again referring to the example above, instead of adding the goal h1(Z) to the goal list, we add h1(_,Z): this allows us to replace the goal *call(G,Message)* by a call to a new built-in predicate *event_call(Message,G)* whose working is as follows: it calls a copy of G in which the first argument is replaced by Message[5]. In hProlog, it is relatively expensive to go from the functor N/A to a the code-entry address of the predicate with functor N/(A+1); this new built-in avoids that cost
- finally, the whole of call_goals/2 has been implemented as two new abstract machine instructions

Clearly, we needed some effort in our WAM based implementation to make it compare favourably to the TOAM approach, but not an unreasonable effort. Most importantly, we have not left the heap oriented WAM approach.

6.3 The event related benchmarks

We measured the performance of the following three actions:

- the *setup* of agents
- the *activation* of agents
- the *execution* of agents

This terminology differs a bit from what is used in the B-Prolog documentation, so we clarify what we mean: for an action rule `p(Z), {event(X,M)} => Body.` the goal `p(Z)` sets up the agent. In B-Prolog terminology this means generating the agent and suspending it. The Body is not (yet) executed at this point.

A subsequent goal to `post(event(X,...)` activates all agents waiting for an event on X: the activation does not yet mean execution of the corresponding bodies, but includes the rechecking of the any guards. B-Prolog also uses the word *activate*.

Finally, only when the bodies are really executed, we speak of an executed agent.

Measuring these actions happens with the following action rules defined:

[5] The arguments in h1/2 are also switched: this saves WAM argument register traffic.

```
fail(X), {event(X,_)} => fail.
true(X), {event(X,_)} => true.
```

In order to measure the setup if N agents, we execute

```
?- time(do_n_times(N,true(X))).
```

In order to measure the activation of N agents, we first set up an agent on the failing action rule, and then on the $N-1$ trivially succeeding action rules. Subsequently, we measure the time for performing one event post. More concretely:

```
?- fail(X), do_n_times(N,true(X)), time(post(event(X,_)).
```

This make sure that the bulk of the time goes to activation, because the executed set of goals immediately fails.

It is difficult to measure execution separately from activation, so in the table below the execute figure includes the activation as well. In order to isolate the execution only, the *pure exec* row contains the difference between the execute and activate row.

	B-Prolog	hProlog
setup	695	220
activate	413	93
execute	641	318
pure exec	228	225

Table 10. Benchmarking the different phases of an event

The figures in Table 10 speak for themselves: setup and activation take much less time in the hProlog heap based approach, while pure execution is basically the same in both systems.

The conclusion is clear: the TOAM stack based implementation doesn't offer essentially better performance than the heap based implementation.

6.4 Vanishing agents

In the terminology of B-Prolog, an agent vanishes after it is activated and commits to a matching clause. E.g. for the following code and query

```
h(X,Y), var(Y), {event(X,_)} => true.
h(X,_) => write(bar).

?- h(X,Y), Y = 1, post(event(X,_)), post(event(X,_)).
```

the output is *bar* once: since the second clause is a matching clause, the agent set up by the goal h(X,Y) vanishes and the last goal post(event(X,_) effectively turns into true without further effects.

The action rule way to kill an agent looks at first sight as a cheap way to deactivate a delayed goal and one that might be difficult to mimic in the WAM approach. However, a small experiment shows that even though the agent has vanished semantically, it remains alive. The experiment makes use of the following code and query:

```
h(X,Y), var(Y), {event(X,_M)} => true.
h(_,_) => true.

test(N) :-
       h(X,Y), ... , h(X,Y), % N times
       Y = 1,
       post(event(X,_)),     % all agents have vanished now
       time(post(event(X,_))).
```

The timings for `?- test(N).` show a linear behaviour in N. This means that the agents are still connected to the variable X even after they have vanished.

Our WAM version of agents and events needs to be adapted slightly to cater for vanishing agents. However, it seems not feasible to overcome the linear effect described just above, except when a large constant factor is tolerated in setup and execution. Within the attributed variable approach, when all agents have vanished, a simple reset of the attribute of X can make it disappear effectively and cheaply. Presumably a similar action is also possible in B-Prolog.

7 Sitting ducks and moving targets

Usually when one benchmarks ones own implementation against another one, the other implementation is a sitting duck. This was not the case this time: when we started working on this, B-Prolog had release 6.8. While working on the paper, several new releases 6.9* came out. After we finished, there was 6.9-b3.2 with new functionality related to events. Here is what the author of B-Prolog described as the changes:

1. An action rule can specify multiple event patterns of event(X,O) as long as the second arguments of the patterns share the same variable.
2. The post_event(Channels,O) can post the event object O to conjunctive and disjunctive channels. For a conjunctive channel (C1 /
   C2), only agents attached to both channel C1 and channel C2 receive the event. The call post_event(C1
   /C2,O) posts the event to agents on both channels.
3. The primitive post_event_dfs(Channels,O) is similar to post_event(Channels,O) but agents on the channels are activated one at a time. While post_event does breadth-first posting, post_event_dfs does depth-first posting.

We will describe in short which changes are needed in our implementation approach to events.

7.1 Multiple event patterns

It is clear that this new feature poses no difficulty for the WAM approach: a slight adaptation of the example code in Section 6.1 takes care of it.

7.2 Conjunctive and disjunctive channels

In our approach, the channel variables carry their suspended agents as lists of Prolog terms, so it is quite easy to augment the definition of post/2 given in Section 6.1 with extra clauses. For disjunction we need something like:

```
post(X \/ Y, Message) :-
        get_attr(X,event,GoalsX),
        get_attr(X,event,GoalsY),
        union(GoalsX,GoalsY,Goals),
        call_goals(Goals,Message).
```

We cannot expect this to compare favourably in speed with B-Prolog, but lower-level support almost certainly will. Note that the clause above does not support disjunction of more than two variables, but it is easy to adapt it.

7.3 Breadth-first versus depth-first posting

The difference between df-posting and bf-posting is shown by the following example:

```
h(X,Y), {event(X,M)} => writeln(1-M), post(event(Y,foo)).
g(X), {event(X,M)} => writeln(2-M).
k(X), {event(X,M)} => writeln(3-M).

| ?- h(X,Y), g(X), k(Y), post_event(X,bla).

1-bla
3-foo
2-bla

| ?- h(X,Y), g(X), k(Y), post_event_dfs(X,bla).

2-bla
1-bla
3-foo
```

The main issue is that the post(event(Y,foo)) goal is postponed in the df version until after all previous posts are finished. There is also a different order of the output 1-bla and 2-bla: this seems to be intentional, but maybe it is not desirable.

Depth-first posting can be done in the WAM approach by the following definition (not taking the other new features into account - and relying on hProlog/SWI specific global variables):

```
post_event_dfs(X,M) :-
        b_getval(insidepostdfs,A),
        (A == [] ->
             b_setval(insidepostdfs,[true]),
             post_event(X,M),
             b_getval(insidepostdfs,ToDo),
             b_setval(insidepostdfs,[]),
             treat_posts(ToDo)
        ;
             b_setval(insidepostdfs,[post(X,M)|A])
        ).
```

The above describes only one possible meaning: the intended meaning is at the moment of writing not completely clear; in particular it needs clarification on the interaction between df and bf. Competitive efficiency can only be obtained by low-level support.

7.4 Evaluation of the new constructs

Ad-hoc constructs can certainly solve an urgent problem. However, it seems to us that a more fundamental approach to the design of the primitives related to events is necessary. For instance: conjunctive and disjunctive channels seem like a good idea, but why not allow them together in one channel expression like $A/\backslash(B\backslash/C$? And why not decouple the reversal of the activated agents from the postponing of new events ? Clearly, the WAM approach can implement such new constructs and because in the WAM approach, the agents on a channel are first order language objects (just lists of Prolog terms) it is a nicer context to experiment with such new constructs.

8 Conclusion

The implementation of delay in B-Prolog fits in very well with the TOAM principle of passing arguments in the execution stack. It even might have an advantage in principle over the WAM way. However, as our experiments have shown, on itself it is not systematically nor significantly better than a well tuned WAM implementation of delay: very important to its performance are specialization (in the form of abstract machine instructions) and inlining (what we named fold

in the experiments). These techniques can be applied in a WAM context, and together with some low level support (a couple of built-ins) and a dedicated slot for freeze (or for other events) which also B-Prolog uses, we get similar performance in WAM and TOAM. The other lesson we learned from going through the experience reported on in this paper is that we must redo old experiments, and most of all: we must look deeper into the reasons for performance differences. Finally, an important advantage of the WAM approach is that it lends itself more easily to experimentation because an agent is just a Prolog term waiting to be metacalled.

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 in Angers, France. Sincere thanks for this hospitality. We also thank Henk Vandecasteele for maintaining the hipP compiler which we use within hProlog.

References

1. H. Aït-Kaci. The WAM: a (real) tutorial. Technical Report 5, DEC Paris Research Report, 1990.
2. M. Carlsson. Freeze, Indexing, and Other Implementation Issues in the WAM. In J.-L. Lassez, editor, *Logic Programming: Proc. of the Fourth International Conference (Volume 1)*, pages 40–58. MIT Press, Cambridge, MA, 1987.
3. M. Carlsson. *Design and Implementation of an Or-Parallel Prolog Engine*. PhD thesis, The Royal Institute of Technology (KTH), Stokholm, Sweden, Mar. 1990 See also: http://www.sics.se/isl/sicstus.html.
4. W. Clocksin and C. Mellish. *Programming in Prolog*. Springer-Verlag, 1984.
5. B. Demoen. Dynamic attributes, their hProlog implementation, and a first evaluation. Report CW 350, Dept. of Computer Science, K.U.Leuven, Belgium, Oct. 2002.
6. B. Demoen and P.-L. Nguyen. So many WAM variations, so little time. In J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey, editors, *Computational Logic - CL2000, First International Conference, London, UK, July 2000, Proceedings*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 1240–1254. ALP, Springer, 2000.
7. C. Holzbaur. Meta-structures vs. Attributed Variables in the Context of Extensible Unification. In M. Bruynooghe and M. Wising, editors, *Proceedings of the Fourth International Symposium on Programming Language Implementation and Logic Programming*, number 631 in Lecture Notes in Computer Science, pages 260–268. Springer-Verlag, Aug. 1992.
8. U. Neumerkel. Extensible unification by metastructures. In *Proceedings of the second workshop on Metaprogramming in Logic (META'90)*, pages 352–364, Apr. 1990.
9. D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI, 1983.

10. N.-F. Zhou. On the Scheme of Passing Arguments in Stack Frames for Prolog. In *Proceedings of The International Conference on Logic Programming*, pages 159–174. MIT Press, 1994.
11. N.-F. Zhou. A Novel Implementation Method for Delay. In *Joint Internatinal Conference and Symposium on Logic Programming*, pages 97–111. MIT Press, 1996.
12. N.-F. Zhou, T. Schrijvers, and B. Demoen. Translating Constraint Handling Rules into Action Rules. Third Workshop on Constraint Handling Rules, Report CW 452, K.U.Leuven, Department of Computer Science, July 2006.