

**Interleaving or separating
environments and choice points in the
WAM**

Phuong-Lan Nguyen Bart Demoen

Report CW 364, August 2003



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

Interleaving or separating environments and choice points in the WAM

Phuong-Lan Nguyen *Bart Demoen*

Report CW 364, August 2003

Department of Computer Science, K.U.Leuven

Abstract

The original WAM uses one stack on which the choice points and the environments are allocated in an interleaved way. Some WAM implementations currently use a separate stack for these two data structures. We evaluate experimentally the difference between these two choices within hProlog and without changing any other feature of the system. The experiments concern time, space and cache performance.

Interleaving or separating environments and choice points in the WAM

Phuong-Lan Nguyen ^{*} Bart Demoen [†]

August 25, 2003

Abstract

The original WAM uses one stack on which the choice points and the environments are allocated in an interleaved way. Some WAM implementations currently use a separate stack for these two data structures. We evaluate experimentally the difference between these two choices within hProlog and without changing any other feature of the system. The experiments concern time, space and cache performance.

1 Introduction

We assume knowledge of Prolog and its implementation. For a good introduction to the WAM [6], see [1]. We have used hProlog 1.8 ¹ in this paper: it is a WAM based system and a successor of dProlog [4]; it is available from the second author. hProlog is meant to become a back end to HAL [3]. hProlog 1.8 is quite complete as a Prolog implementation: it merely lacks dynamic predicates; it supports freeze/2 (as in [2]). Measurements were reported in [5] and they show that hProlog has an excellent performance (as an emulator), so that we can be reasonably confident that our experiments are meaningful.

We will use the following terminology: the *heap* is the area in which the WAM allocates structured terms – elsewhere the term *global stack* is often used for this

^{*}Inst. de Mathématiques Appliquées, UCO, Angers, France, nguyen@ima.uco.fr

[†]Dept. of Computer Science, K.U. Leuven, Belgium, bmd@cs.kuleuven.ac.be

¹hProlog has evolved further in the mean time: the current version is 2.3 and includes attributed variables, the string and character datatype and bigints

area; the *environment stack* is the stack with environments or stack frames in more traditional terminology; the *choice point stack* contains the choice points. The original WAM uses a merged environment/choice point stack which is named the *local stack*.

Several Prolog implementations do not use the original local stack anymore: SICStus Prolog was perhaps the first one, but also neither XSB nor hProlog use a single stack for environments and choice points. Apparently, the motivation has been that locality of reference is better in a split stack model, but also that the split stack model offers easier implementation of parallel Prolog systems or tabling based on suspension/resumption of consumers as in XSB. Yap however implements the original single local stack model and also, Yap is very fast, if not the fastest emulator around. The natural question is of course whether this is a coincidence or not and indeed one of the reasons for the current work is to investigate to what extent Yap's performance could be attributed to this traditional approach to the local stack. We have consequently performed the following experiment: the implementation hProlog 1.8 was adapted in such a way that a compile (of the C-code) time option generates a system that uses either the original WAM local stack or the (default) split stack. We will refer to these versions of hProlog as *one_stack* and *two_stack*. We will describe the adaptations in more detail in Section 2. We then report on the time and space performance of *_stack on a set of benchmark programs in Section 3. Next we run a subset of these benchmarks under a cache simulator, so that we can check whether there is a significant difference in cache behaviour between the two systems: see Section 4. The paper ends with a conclusion in Section 5.

2 Changes to the hProlog machine

The text below can be understood without knowing the particularities of the hProlog variant of the WAM, but some of the quoted code only makes sense when the following is understood: in hProlog, environments are always *upside-down* (and no environment trimming is performed).

2.1 The data structures

hProlog is largely² re-entrant and one record - together with what it points to - captures one incarnation of the WAM as implemented by hProlog. This record - a struct in C, named *machine* - contains all WAM (and extended WAM) registers, pointers to all stacks and stack limits, the open files and the information for statistics. In *two_stack* mode, the machine contains a TOS³ register and the delimitations of the environment and the choice point stack. In *single_stack* mode, it does not contain the TOS register and instead of the afore mentioned delimitations, the begin, the end, and the overflow limit of the single stack for choice points and environments.

One more data structure is affected by the stack decision: the choice point. In *two_stack* mode it contains the top of environment stack; in *single_stack* it does not.

2.2 The code

The code for resetting the registers on backtracking differs, as in the *single_stack* mode, TOS need not be reset - and when a choice point is created, it need not be saved.

At the moment that a choice point is pushed, the top of the choice point stack must be determined. In *two_stack* mode, this is trivial: the top of the choice point stack is always the current *B*. In *single_stack* mode, we get code like:

```
if (B < E) topofcp = B; else topofcp = E;
```

In *two_stack* mode and assuming that the top of the environment stack - TOS - is set correctly at the start of the execution, it needs updating at the *allocate* and *deallocate* instructions. At *allocate* TOS comes in correct, and is changed by

```
TOS = TOS + nrofpermvars;
```

for allocating an environment with $(nrofpermvars-2)$ permanent variable slots. (the 2 represents the fixed part of the environment: the environment back pointer and the continuation pointer)

At *deallocate*, the computation of TOS must take into account environments blocked by the current top choice point:

²The program and symbol tables are global by choice; the interrupt routines (written in C) need to know which machine is executing.

³top of environment stack

```
if (E > B[tos])
    TOS = B[tos];
else
    TOS = E;
```

In `single_stack` mode we must compute the top of environment stack at *allocate*, since we chose not to save it in the choice points. The code is similar as above for *deallocate* in `double_stack` mode. This is the only place where the top of environment stack is computed and used in `single_stack` mode.

2.3 Other small changes

The code and heap garbage collectors, and the expansions of the runtime stacks were disabled: they are not called during the tests because we start all tests with an initial size that is large enough.

3 Time and space performance

The time and space performance was measured on a set of classical benchmark programs, a compiler compiling itself and a few artificial benchmarks. Timings are reported in milliseconds: benchmarks were repeated a number of times until some reasonable total was obtained. The *%difference* column contains the excess of `one_stack` over `two_stack`, i.e. a negative sign reflects badly on `two_stack`. Timings were made on a Pentium III, 500MHz, 128 Mb.

Table 1 with timings for the classical benchmarks shows differences of about 3% both ways. That hardly seems meaningful, but the *meta_qsort* and *queens* perform lots of backtracking, which seems to indicate that backtracking programs benefit from having a single stack.

benchmark	one	two	% difference
boyer	690	700	-1.5
browse	870	860	+1.1
cal	1140	1160	-1.8
chat	980	970	+1.0
crypt	700	700	0
ham	1400	1390	+0.7
meta_qsort	910	940	-3.2
nrev	1060	1040	+1.9
poly_10	530	530	0
queens_16	1160	1180	-1.7
queens	2220	2290	-3.1
reducer	280	280	0
sdda	690	670	+2.9
send	760	770	-1.3
tak	790	790	0
zebra	1680	1680	0
average			-0.4

Table 1: Timings for some traditional benchmarks

The benchmarks used in Table 2 consist of

- *comp*: an old version of the XSB compiler that was adapted so that it runs under different Prolog systems
- *cpls*: an artificial benchmark which repeatedly creates N environments and then N choice points (see code in Appendix)
- *move1* and *move2*: both are artificial benchmarks (see code in Appendix) that move repeatedly predicate arguments between choice points, argument registers and environments; their structure is slightly different

For the sake of comparison, we have included timings for these benchmarks on SICStus 3.8.5, and Yap-4.3.22: SICStus has a split stack, Yap has single stack.

For the more realistic benchmark *comp*, the split stack model loses narrowly. As expected, with artificial benchmarks one observes larger differences, but we have not been able to find a satisfactory explanation for the difference between *move1* and *move2*, except perhaps the cache behaviour as shown in Section 4.

benchmark	one	two	% difference	SICStus	Yap
comp	12490	12560	-0.6	21270	15130
cpls	2380	2530	-6.0	4580	3800
move1	16510	17160	-3.8	22530	19380
move2	21640	21160	+2.2	30670	22700

Table 2: Timings for some other benchmarks

We report about the difference in space usage only for two benchmarks in Table 3. Since a choice point is smaller in one_stack, one can expect a smaller stack usage for one_stack. On the other hand, in the one_stack model, the cut can free the space of a choice point, but it cannot be reused immediately as in the two_stack model, so in that case the two_stack model has a smaller stack usage. Both are observed but the difference is small.

benchmark	one	two	% difference
boyer	480	510	-0.6
comp	12459	12173	+0.2

Table 3: Space performance of two benchmarks: total of environment and choice point stack

4 Cache performance

Table 4 reports about the number of instructions executed during three benchmarks, the number of reads, the number of L1 read misses, the number of writes and the number of L1 write misses - always in percentages as above, and only for the execution inside the emulator function.

All three show that `one_stack` executes less instructions, and has less reads and writes.

The most striking difference is between the write misses for `move1` and `move2`: this might be entirely due to the relative position of environment and choice point stack. We do not attach much importance to it.

benchmark	instructions	reads	read misses	writes	write misses
comp	-1.5	-2.0	+1.1	-7.2	-0.7
move1	-0.8	-1.0	+0.4	-5.8	-3.9
move2	-0	-0.8	-0.8	-4.5	+4.4

Table 4: Excess percentage of `one_stack` over `two_stack`

The cache statistics were obtained with *cachegrind* written by Julian Seward (jseward@acm.org).

5 Conclusion

The power of this contribution is in the fact that within basically the same system the two alternative stack layouts are implemented and that all other aspects of the abstract machine were kept the same: this means that any observed performance differences can be attributed to the difference between one and two stacks. The fact that hProlog is reasonably performant compared to other state-of-the-art WAM emulators, adds to the credibility of the experiment.

Folklore tells that the locality of reference is better in the single stack model and that consequently the performance should be better. The experiment confirms this, but in a weak sense: the differences are so small that they seem hardly significant. The main conclusion is that there is no good performance reason to chose one model over the other and that certainly the excellent speed of Yap cannot be attributed to its conservative stack layout.

Acknowledgements

This work was conducted while the second 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 also thank Henk Vandecasteele for his work on the ilProlog compiler used within hProlog.

References

- [1] H. Ait-Kaci. The WAM: a (real) tutorial. Technical Report 5, DEC Paris Research Report, 1990. See also: <http://www.isg.sfu.ca/~hak/documents/wam.html>.
- [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] B. Demoen, M. García de la Banda, W. Harvey, K. Mariott, and P. Stuckey. An overview of HAL. In J. Jaffar, editor, *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, volume 1713 of *LNCS*, pages 174–188. Springer, 1999.
- [4] 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.
- [5] B. Demoen and P.-L. Nguyen. About unnecessary performance differences between Prolog implementations. In *Proceedings of CICLOPS - Colloquium on Implementation of Constraint and Logic Programming Systems*, 2001. accepted.
- [6] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI, 1983.

Appendix

All benchmarks to be activated with the query ? – *go*.

cpls

```
go :- time((t(10000,100,100);true)).

%% N for how often the thing repeats
%% E for how many envs before how many C choicepoints

t(N,E,C) :- N > 0,
            envs(E,C),
            M is N - 1,
            t(M,E,C).

envs(E,C) :- (E > 0 ->
             E1 is E - 1,
             envs(E1,C),
             d
             ;
             choices(C)
             ).

choices(C) :- C > 0, C1 is C - 1, choices(C1).
choices(C) :- d, C == 0.

d.
```

The definition of time/1

```
time(X) :-
    statistics(runtime,[T1|_]),
    call(X),
    statistics(runtime,[T2|_]),
    T is T2 - T1,
    write(T),nl.
```

move1

```
go :- time(move(1000000)).

%% idea is to create a choicepoint with 10 arguments
%% and repeatedly backtrack to it, so that these args must
%% be moved to a (new) environment

move(_) :- a(1,2,3,4,5,6,7,8,9,0).
move(N) :-
    (N > 0 ->
        M is N - 1,
        move(M)
    );
    true
).

% repeat the following a/10 clause 10 times ...
a(Y0,Y1,Y2,Y3,Y4,Y5,Y6,Y7,Y8,Y9) :-
    d,
    b(Y0,Y1,Y2,Y3,Y4,Y5,Y6,Y7,Y8,Y9),
    fail.

b(_,_,_,_,_,_,_,_,_,_).

d :- fail.
```

move2

```
go :- time((a(1,2,3,4,5,10000000);true)).

a(A1,A2,A3,A4,A5,N) :- a, fail, b(A1,A2,A3,A4,A5,N), fail.
a(A1,A2,A3,A4,A5,N) :-
    N > 0,
    M is N - 1,
    a(A1,A2,A3,A4,A5,M).

a.
b(_,_,_,_,_,_).
```