Environment Reuse in the WAM

Bart Demoen
Phuong-Lan Nguyen

Report CW 510, February 2008

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

Environment Reuse in the WAM

Bart Demoen
Phuong-Lan Nguyen

Report CW 510, February 2008

Department of Computer Science, K.U.Leuven

Abstract

The WAM passes arguments to predicate calls in registers. The TOAM pushes arguments to predicate calls onto the execution stack. The WAM compiles clauses at a time. The TOAM treats a predicate as the compilation unit. These differences boil down to reuse of argument registers versus reuse of stack frames. In the TOAM, the environment reuse comes together with eager environment creation, while the WAM creates environments lazily. This warrants further investigation. The tak/4 benchmark is used as an inital case study for better understanding what one can expect from environment reuse for deterministic predicates. Additionally, artificial programs are used to amplify the findings. The experiment shows the relative merit of reusing an environment versus avoiding to create it: the latter seems a superior technique. The expected performance gain is related to the form of the program and the determinism of its predicates.

Environment Reuse in the WAM

Bart Demoen[&] and Phuong-Lan Nguyen[@]

& Department of Computer Science, K.U.Leuven, Belgium
@ Institut de Mathématiques Appliquées, UCO, Angers, France
bmd@cs.kuleuven.be, nguyen@ima.uco.fr

Abstract. The WAM passes arguments to predicate calls in registers. The TOAM pushes arguments to predicate calls onto the execution stack. The WAM compiles clauses at a time. The TOAM treats a predicate as the compilation unit. These differences boil down to reuse of argument registers versus reuse of stack frames. In the TOAM, the environment reuse comes together with eager environment creation, while the WAM creates environments lazily. This warrants further investigation. The tak/4 benchmark is used as an inital case study for better understanding what one can expect from environment reuse for deterministic predicates. Additionally, artificial programs are used to amplify the findings. The experiment shows the relative merit of reusing an environment versus avoiding to create it: the latter seems a superior technique. The expected performance gain is related to the form of the program and the determinism of its predicates.

1 Introduction

We assume familiarity with Prolog [2], the WAM [1, 4] and the TOAM [5]. Some acquaintance with the B-Prolog implementation of the TOAM and with hProlog [3] might also come in handy.

We do not know the reasons why David H. Warren choose to pass the arguments to a predicate in argument registers, rather than on the stack, but we can try to reconstruct them with hindsight. One observation is that the WAM compiler compiles clauses separately: the code for a clause does not depend in the other clauses of the same predicate. This allows clauses to have their own specific optimizations. For instance, some clauses are just facts, so they can be treated as a leaf procedure in a classical compiler, i.e., they do not need a stack frame or in WAM-speech no environment: we refer to this as environment avoidance. Of course, the clauses of one predicate need to be glued together - the WAM does that with indexing instructions and a try-retry-trust chain - but this does not violate the basic principle. One reason for this choice could be that adding a compiled clause to an existing predicate is efficient, as that addition does not need to deal with the already existing clauses.

The second observation is related to the last goal in a clause body with at least two goals: if the other goals in the clause have not left a choice point behind, the last goal can discard the environment, so that the same space can be reused

by a subsequently activated clause. So the code for the last goal must check for this determinacy. If arguments are passed through the stack, the location where they need to go depends on the determinacy, and since the general compilation schema for a goal in the WAM is *generate code for the arguments, then generate the call itself*, the choice is between code as in the next table at the left and at the right:

if (! deterministic) goto nondet;	if (deterministic)
generate arg(1) in detloc(1)	then generate arg(1) in detloc(1)
...	else generate arg(1) in nondetloc(1)
generate arg(n) in detloc(n)	...
optimized tail call	if (deterministic)
nondet:	then generate arg(n) in detloc(n)
generate arg(1) in nondetloc(1)	else generate arg(n) in nondetloc(n)
...	if (deterministic)
generate arg(n) in nondetloc(n)	then optimized tail call
non-optimized tail call	else non-optimized tail call

Neither is particularly attractive, and one can see why choosing for a fixed location to put the arguments into, makes the abstract machine simpler. On top of that, both schemes have particular problems when the optimized tail call is to a predicate with a different arity than the head of the clause. This is reflected in B-Prolog code which following the TOAM has choosen for a form of the above alternatives.

The last observation is related to the reuse of the head arguments for body goals: clearly, as soon as the head unification has succeeded (and any inlined goals), the first goal is set up for sure. In that case, reuse of head arguments is possible when the arguments go to a fixed location. When arguments go to a variable location, the arguments of the first call cannot be reused. In case the arguments of the last call go to the same place as the arguments in the head - that is: in the TOAM schema and when the last goal is reached deterministically - the last call can reuse the arguments of the head. However, the number of times the last goal in a clause is in a position to reuse in the TOAM is not larger than the first goal in the WAM: the reason is of course that the execution can fail before the last goal is reached. So betting on reuse in the first goal seems good.

One can object to the above reasoning by pointing out that for deterministic recursive predicates - very common in applications - the last goal is often very similar to the head, and calling the same predicate, so with the same arity. In this case, the TOAM approach could be better at reuse that the WAM. On top of that, in such a case, the TOAM reuses all the fixed fields of the environment, while the WAM reuses the environment space, but not the contents. This difference is exactly what we will investigate in this paper: we will start by using tak/4 as a case study in Sections 2 and 3. The experiment indicates that the WAM approach can be improved by adopting an environment reuse schema as in the TOAM, at least for tak/4. Section 4 discusses the dynamics of tak/4 and provides more insight in the experimental data. Section 5 performs basically the same experiment in C, showing similar results and also the direct link with

traditional compiler technology. Section 6 uses artificial benchmarks for showing the relative merit of environment reuse versus environment avoidance. Section 7 concludes.

The experiments were done on a 1.8 GHz Pentium 4 with Linux (hProlog 2.7[1], B-Prolog 7.1b3.2) and on an Intel Mac (hProlog 2.7[2], B-Prolog 7.0). Timings are always in milliseconds.

2 Source code for tak/4

Below left is the source code for tak/4 as in the original benchmark. At the rigth, there is the modified code we have also used for this paper.

```
Original (hProlog and B-Prolog)          Modified (for other systems)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~          ~~~~~~~~~~~~~~~~~~~~~~~~~~~~

tak(X,Y,Z,A):-                           tak(X,Y,Z,A):-
        X =< Y,                                  (X =< Y ->
        Z = A.                                       Z = A
tak(X,Y,Z,A):-                                   ;
        X > Y,                                       X1 is X - 1,
        X1 is X - 1,                                 tak(X1,Y,Z,A1),
        tak(X1,Y,Z,A1),                              Y1 is Y - 1,
        Y1 is Y - 1,                                 tak(Y1,Z,X,A2),
        tak(Y1,Z,X,A2),                              Z1 is Z - 1,
        Z1 is Z - 1,                                 tak(Z1,X,Y,A3),
        tak(Z1,X,Y,A3),                              tak(A1,A2,A3,A)
        tak(A1,A2,A3,A).                         ).
```

The hProlog compiler generates the same code for both versions. B-Prolog does the same. So for hProlog and B-Prolog, it doesn't matter which version of tak is started from, but many other implementations do not take advantage of the fact that the negation of the first test implies the second test. Such implementations construct a choice point for every activation of tak/4. Even if the use of these systems is marginal for the purpose of the paper, it would handicap them unnecessarily to use the original form.

3 Abstract machine code for tak/4

Below is the code as generated by B-Prolog (left) and hProlog (right): we have taken the output from *'bpc'/1* and *print_code/1* respectively, but massaged it a little so that the labels become more readable and we have removed code that is not executed during the benchmark.

[1] compiled with gcc (GCC) 3.3.5 (Debian 1:3.3.5-13)
[2] compiled with i686-apple-darwin8-gcc-4.0.1

```
B-Prolog                                hProlog
~~~~~~~~                                ~~~~~~~
@tak:                                   @tak:
 allocate_det(4,10,@tak)

@afteralloc:
 jmpn_ge_uu(3,4,@else)                   test_smaller_or_equal A1 A2 @else
 unify_value_return_det(2,1)             gettval_proceed A4 A3

@else:                                  @else:
                                         allocategetpvar Y9 Y2 A1

 sub_u1v(4,-4)                           add_integerp A1 Y2 -1
                                         getpvarputpvar A4 Y3 A4 Y4
                                         getpvar2 A3 A2 Y5 Y6
 call_uuuv_d(@tak,-4,3,2,-5)             call @tak

 sub_u1v(3,-6)                           add_integerp A1 Y6 -1
 call_uuuv_d(@tak,-6,2,4,-7)             putpvarvalvalcall A4 A2 A3
                                                           Y7 Y5 Y2 @tak

 sub_u1v(2,-8)                           add_integerp A1 Y5 -1
 call_uuuv_d(@tak,-8,4,3,-9)             putpvarvalvalcall A4 A2 A3
                                                           Y8 Y2 Y6 @tak

                                         putpvalvalvalval A1 A3 A2 A4
                                                          Y4 Y8 Y7 Y3
 tr_det_call_au(@afteralloc,             deallex @tak
                801,4,-5,-7,-9,1)
```

TOAM code is less well known than WAM code, so a short explanation follows:

- TOAM variables are addressed as an offset from the frame pointer; incoming arguments have positive offsets, local variables have negative offsets; for example `sub_u1v(3,-6)` is the code for the goal `Y1 is Y - 1`: 3 is the offset of the incoming argument Y, while -6 is the offset of Y1
- the code for the goal `tak(Z1,X,Y,A3)` is `call_uuuv_d(@tak,-8,4,3,-9)`: -8 stands for Z1, 4 for X, 3 for Y and -9 for A3
- in this `call_uuuv_d` instruction, the u is the equivalent of val in the WAM, and the v is the equivalent of var, i.e., it indicates whether the variable has been seen before or not

We ignore the differences in instruction merging and specialization at the moment and focus on the more important difference in how the two systems deal with arguments and with environments. Note that we use often *environment* for both the WAM and the TOAM stack frame, even though the TOAM frame

contains choice point ingredients when the predicate is not (detected to be) deterministic.

- hProlog passes parameters through argument registers; these registers do not reside in the stack; B-Prolog pushes arguments on the stack just before the stack frame that the callee will make
- hProlog postpones allocating an environment until it enters a basic block that needs it, while B-Prolog allocates a stack frame on entering the predicate
- hProlog deallocates the environment on the tail call, while B-Prolog reuses the environment for the tail call by jumping to an entry-point after the allocating instruction

Even though hProlog executes more instructions (because of less instruction compression, and because of the argument registers), and never reuses an environment, hProlog is faster by 33% on the Linux machine, and about 9% on the Mac. The goal tak(18,12,6,_) is executed 100 times to obtain those figures: the first two columns of Table 1 show them. The next section contains an explanation for the fact that the WAM approach works so well compared to the TOAM approach.

4 The dynamics of tak/4

During the benchmark, the then-branch is taken 4.770.700 times, while the else-branch is taken 1.590.200 times: that is (close to) 3 times less. The factor 3 is of course not a surprise: 3 out of 4 calls in the body are non-tail calls. So, in total, B-Prolog allocates an enviroment 4.770.700 times, while hProlog does the same 3 times less. This can be visualized by the execution tree for tak/4 in which each call-node has outgoing degree equal to 4: the leaves correspond to calls of the form tak(X,Y,Z,A) in which $X \leq Y$, for which the WAM does not allocate an environment. Since the number of nodes N relates to the number of I internal nodes by the simple formula $N - 1 = 4 * I$, the conclusion is easy to make. It is easy to generalize these findings, at least to deterministic programs.

Clearly, the eager allocation of a stack frame (for deterministic programs) seems counterproductive, and it would be a nice experiment to modify B-Prolog to do lazy stack frame allocation, as the WAM does. Since the source code of B-Prolog is not available to us, we have taken the other path, namely modifying hProlog to *reuse* its environments, at least in the tak/4 benchmark - and later in some artificial benchmarks.

hProlog had already enough instructions to do environment reuse, and as an initial experiment we produced abstract machine code for tak/4 to perform environment reuse by hand. We wrote the following code:

```
@entry_no_alloc:
 test_smaller_or_equal A1 A2 @else
 gettval_proceed A4 A3

@entry_alloc:
 test_smaller_or_equalpp Y2 Y3 @afterallocate
 getpvalv Y5 Y4
 dealloc_proceed

@else:
 allocategetpvar Y9 Y2 A1
 getpvar3 A2 A4 A3 Y3 Y5 Y4

@afterallocate:
 add_integerp A1 Y2 -1
 putpvarvalvalcall A4 A2 A3 Y6 Y3 Y4 @entry_no_alloc

 add_integerp A1 Y3 -1
 putpvarvalvalcall A4 A2 A3 Y7 Y4 Y2 @entry_no_alloc

 add_integerp A1 Y4 -1
 putpvarvalvalcall A4 A2 A3 Y8 Y2 Y3 @entry_no_alloc

 move_var3 Y6 Y2 Y7 Y3 Y8 Y4
 fast_execute @entry_alloc
```

The *fast_execute* instruction is like a jump, but it also performs a heap-overflow check just like the *execute* instruction, hence the name.

The similarity with B-Prolog's code is of course no surprise: we worked towards it. But note that our new code performs avoids to *allocate* an environment when possible, and at the same time reuses an environment whenever possible. Just as in B-Prolog, the cost for reuse is three moves before the tail call. Note that we have used the fact that tak/4 is deterministic: if an interrupt could push a choice point in between, then our implementation (of fast_execute) would not be correct. Solving this issue requires some runtime tests, and slows down slightly the execution, so it means that the timings we get with the above code, are slightly optimistic.

Our code is not as lean as the one in B-Prolog, because hProlog does not compress the sequence *getpvalv, dealloc_proceed*.

Table 1 shows the timings for B-Prolog, hProlog and the hProlog version that performs environment reuse. There is a clear gain in re-using the environment for hProlog, although it is dependent on the platform and compiler combination. One can only speculate on the gain for B-Prolog if it were to allocate its frames more lazily. The figures for SICStus and Yap are given as evidence that hProlog and B-Prolog are performance wise relevant systems to do these experiments in.

	hProlog	B-Prolog	hProlog + reuse	SICStus	Yap
tak on Linux	315	473	278	810	1372
tak on Mac	375	412	367	740	580

Table 1. Tak on different platforms and systems

5 The same experiment in C

Below is some straithforward C-code for tak, and a form of the C-code that does the environment avoidance by hand:

```
int tak(int x, int y,int z)
{

  if (x <= y) return(z);

  {
    int x1, y1, z1;
    int a1, a2, a3;
    x1 = x - 1; a1 = tak(x1,y,z);
    y1 = y - 1; a2 = tak(y1,z,x);
    z1 = z - 1; a3 = tak(z1,x,y);

    return(tak(a1,a2,a3));

  }
}
```

```
int tak(int x, int y,int z)
{
 noalloc:
  if (x <= y) return(z);

  {
    int x1, y1, z1;
    int a1, a2, a3;
    x1 = x - 1; a1 = tak(x1,y,z);
    y1 = y - 1; a2 = tak(y1,z,x);
    z1 = z - 1; a3 = tak(z1,x,y);

    x = a1; y = a2; z = a3;
    goto noalloc;
  }
}
```

The version that avoids the recursive call, is between 5 and 10% faster than the other version (on both platforms). This was measured with optimization levels O2 and O3 from gcc 3.3.5 and 4.0.1 respectively. Variants of the above versions, showed the same difference rather consistently. This is in line with the observed speedup for hProlog.

6 Artificial benchmarks

In order to amplify the potential advantage of environment reuse and environment avoidance, we have constructed a sequence of benchmarks with a characteristic similar to tak/4, but from which the fluff was removed. In particular, we have benchmarks $taklike_n$ for $n = 1..10$. For $n = 5$ its predicates are defined as:

```
taklike_5(X) :-
        (X =< 1 ->
            true
        ;
            s, s, s, s, s,  % 5 calls to s
            X1 is X - 1,
            taklike_5(X1)
        ).

s.
```

The search tree for such a predicate has the form as in Figure 1. The black nodes

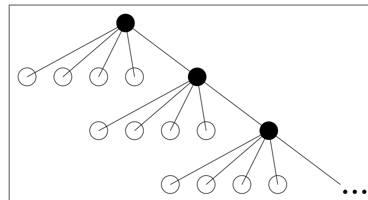

Fig. 1. Search tree of the artificial benchmarks

correspond to calls that can reuse the current environment. The other nodes cannot. For $taklike_n$, the ratio between the calls that can avoid an environment and the calls that can reuse the environment is $(n-1) : 1$.

The goal is always of the form $? - taklike_n(5000000)$.: this number was choosen because it gives reasonably meaningfull timings. Table 2 shows the timings for hProlog and B-Prolog on two platforms: in i/j, the first item refers to a timing on the Linux machine, the seond one is obtained on the Mac.

The first column of Table 2 shows the results of running hProlog unaltered on the benchmarks. The second column shows the effect of making hProlog allocate an environment for the *s.* fact: normally the WAM (and hProlog alike) generates just a *proceed* instruction; in this case, hProlog was made to generate a *allocate, deallocate_proceed* sequence. The third column shows the result for hProlog with reuse of the environment for the tail call to *tak_like*.

The figures for SICStus, Yap and B-Prolog are given merely for comparison, and the first two will not be discussed.

n	hProlog	hProlog +extra env	hProlog +reuse	SICStus	Yap	B-Prolog
1	150/183	208/249	110/123	720/515	378/353	226/254
2	250/199	458/325	208/147	980/610	488/416	492/466
3	418/229	542/496	374/170	1095/790	714/598	596/590
4	450/252	628/572	404/196	1240/950	769/625	692/704
5	478/369	714/670	432/317	1315/1050	842/670	792/826
6	514/398	798/739	462/340	1425/1180	872/732	894/959
7	542/422	888/817	500/364	1535/1280	958/781	994/1154
8	586/454	972/899	524/397	1640/1380	984/844	1100/1191
9	590/484	1056/980	558/417	1745/1590	1052/938	1204/1367
10	630/492	1148/1058	589/444	1870/1705	1086/1014	1304/1456
increment	53/34	104/89	53/35	127/132	78/73	119/133

Table 2. The effect of creating an extra environment and re-using the environment

Table 2 also indicates the average increment between successive values of n. The hProlog columns show that

- the relative gain of environment reuse depends on n: the gain is larger with smaller n; this gain goes from 26% to 6.5% (on Linux) and 27% to 9.7% (on Mac); note that those are overestimates of what can be achieved in practical programs, because the artificial benchmarks contain hardly any fluff
- the relative loss of creating the extra environment is about 27% to 45% (on Linux) and 26% to 53% (on Mac); again, those figures overestimate the relative effect

When one considers the absolute figures (for Linux), one sees that environment avoidance reduces the runtime by 58 msecs up to 518 msecs. Environment reuse gives an almost constant gain between 30 and 40 msecs.

One can conclude that the WAM optimization of not allocating an environment for a fact, is more effective that the TOAM optimization of reusing the stack frame. Both the absolute and relative figures suggest that. Moreover, as expected, environment avoidance becomes better when there are more goals in the body.

7 Conclusion

Tak/4 lends itself easily to environment reuse in the WAM: such reuse is more difficult if a predicate has more than one clause with an allocate, and if one still wants to compile clauses in isolation, as the WAM does. So we cannot claim that we have ultimate answers and solutions. The experiment with tak/4 was rather easy to perform because of hProlog's large instruction set. We observed that environment reuse for tak/4 was quite effective, but depending on the platform-gcc combination. The analysis of the experimental results shows that environment

avoidance is a better optimization than environment reuse. Of course, for performance reasons, one would like to have them both. The next step should be an adaptation of the hProlog compiler to exploit the reuse of environments.

Acknowledgements

Bart Demoen thanks Research Foundation-Flanders (FWO-Vlaanderen) for support. Part of this work was performed during a visit to IMA, UCO, Angers.

References

1. H. Aït-Kaci. The WAM: a (real) tutorial. Technical Report 5, DEC Paris Research Report, 1990.
2. W. Clocksin and C. Mellish. *Programming in Prolog.* Springer-Verlag, 1984.
3. 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, *CL2000: Proceedings of the 1st International Conference on Computational Logic*, volume 1861 of *LNAI*, pages 1240–1254, London, UK, July 2000. ALP, Springer Verlag.
4. D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI, 1983.
5. N.-F. Zhou. Global optimizations in a Prolog compiler for the TOAM. *Journal of Logic Programming*, 15(4):275–294, 1993.