So many WAM variations, so little time.

Bart Demoen¹ and Phuong-Lan Nguyen²

¹ Department of Computer Science, K.U. Leuven,
Celestijnenlaan 200A, B-3001 Leuven, Belgium
e-mail: `bmd@cs.kuleuven.ac.be`

² Institut de Mathématiques Appliquées, Université Catholique de l'Ouest,
Place André Leroy 3, 49008 Angers, France
e-mail: `nguyen@ima.uco.fr`

Abstract. The WAM allows within its framework many variations e.g. regarding the term representation, the instruction set and the memory organization. Consequently several Prolog systems have implemented successful variants of the WAM. While these variants are effective within their own context, it is difficult to assess the merit of their particular variation. In this work, four term representations that were used by at least one successful system are compared empirically within dProlog, one basic implementation which keeps all other things equal. We also report on different implementation choices in the dProlog emulator itself. dProlog is reasonably efficient, so it makes sense to use it for these experiments.

1 Introduction

The WAM (see [1, 16]) has been the basis for many Prolog systems, also the most successful ones. It leaves open certain issues like the implementation of cut, dynamic code etc. but even when it specifies other issues, there are variations that can still pass for the WAM in the broad sense. Examples are optimizations like instruction compression, new ways to propagate the read-write mode, the organization of the stacks or a different tagging schema. Another variation - the one of interest here - concerns the term representation itself: the WAM initializes permanent variables on the local stack when possible, while other implementations have chosen to globalize permanent variables on their first occurrence (BIM-Prolog always, AQUARIUS [15] under certain conditions) and consequently do not have to deal with *unsafe* variables; PARMA [14] represents the binding between two free variables differently from the WAM and has constant time dereferencing; BinProlog [13] employs a *tag on data* schema instead of the WAM *tag on pointer* schema. Each of these WAM variations was effective within its own context: BIM-Prolog used native code generation (not common in 1985 !); BinProlog binarizes clauses before compiling them [12]; PARMA and AQUARIUS rely on abstract interpretation for their top speed. Because the implementation context of these term representations was different from each other and from the original WAM, it is not at all clear how they really compare, i.e. there has been no empirical study of the impact of changing within a single

efficient system one term representation into another one while *all others things are kept equal*. We intend to do exactly that: evaluate empirically and compare directly within the same basic implementation the four above mentioned term representations.

Such an experiment only makes sense if the basic Prolog implementation is reasonably complete and fast. Yap [4] or SICStus Prolog [3] could have served as a good starting point for the experiment. However, adapting such systems is very time consuming and that was one reason for starting almost from scratch: because of our prior involvement with XSB (see e.g. [7]), we borrowed the XSB compiler (see [17]), for the generation of abstract machine code (XSB is largely WAM based) and we built a new emulator. This permitted us to redo partly the experiment reported on in [4] and at the same time benefit from its advice, investigate the potential for speeding up the Prolog part of XSB and build the basis for a dedicated Prolog system for inductive learning applications [6]. It also gave us the chance to satisfy a private curiosity regarding an unusual term representation for integers and floating point numbers, and a different layout of environments: these are reported on later.

We named the resulting Prolog system **dProlog**¹. dProlog is complete enough to bootstrap itself but hardly more complete than needed for the experiment.

We decided to stay in the emulator business as [4], but since it clearly doesn't make sense to study the impact of changes in a slow system we wanted dProlog to be in the same ball park as SICStus Prolog emulated with all bells and whistles (i.e. using gcc extensions). The SICStus Prolog emulator is not the fastest around: Yap [4] beats SICStus Prolog consistently. So we decided to borrow from Yap techniques applicable at the C-level as well as the abstract machine code level. As a consequence dProlog performs as good as Yap for certain benchmarks - especially smaller ones - but given our choice of XSB for generating the abstract machine code, dProlog has not a chance to get always on par with SICStus Prolog. Still, we felt that the initial goal was met close enough. We report on the aspect of performance compared to other systems in section 4, mainly because it shows that our emulator is of sufficient quality to make the results of the experiments relevant.

We also experimented with implementation choices within the emulator itself. Therefore we have set up dProlog so that it can be installed in 6 different basic emulator modes depending on two parameters: the first parameter sets the number of opcode fields in each instruction; it is either *1* - as is customary - and then the read-write mode in the unify-instructions is explicitly tested by means of the WAM S register; or it is *2*, in which case the read-write mode is propagated by using the first opcode field in read mode and the second in write mode (see [4] for more detail). The second parameter can orthogonally be set to *switch*, *jump-table* and *threaded* which is a similar choice as in SICStus *THREADED* = *0,1* or *2*. The latter two require GNU cc. We report on the effects (time and space) of these six modes in section 5.

¹ <http://www.cs.kuleuven.ac.be/~bmd/dProlog> for source code and benchmarks

Section 6 reports on choices in the implementation of the emulator: the assignment of a hardware register to the WAM program counter, binding calls, conditional trailing, trail overflow checking and 28 versus 32 bit integers. In the same section, we also report on the effect of several instruction compressions.

The initial version of dProlog (with all the above variations) always globalizes permanent variables on their first occurrence: we will refer to it as the **heap_vars** version. We then created three more versions: we will refer to them as **wam_vars**, **parma_vars** and **tag_on_data**. They differ from the heap_vars dProlog in only one aspect: wam_vars initializes permanent variables on the local stack whenever possible, exactly as in the WAM, and thus knows *unsafe* variables (see [1]); parma_vars uses the variable representation as in [14]; tag_on_data uses the representation as in BinProlog [13], but without term compression. Parma_vars and tag_on_data also always globalize permanent variables: BinProlog, the only other Prolog system using the tag-on-data representation, doesn't even have a local stack and the intricacies of having PARMA variables also in the local stack reported in [11] scared us off. All these different representations are shortly explained in section 3. We report on the impact of these changes in section 7.

We start with an overview of the XSB compiler and dProlog in section 2.

All the experiments were performed on a Pentium II, 260MHz, 128Mb.

2 The XSB compiler and dProlog

XSB supports HiLog and tabling, both of which we were not interested in for this experiment: consequently, we have removed most of the code in the XSB compiler that deals with these extensions. They do not interfere with the compilation of ordinary Prolog programs. Also *call specialization*² was switched off. The XSB compiler can generate indexing for any argument (depending on declarations); we have disabled the index declaration and specialized all indexing instructions to the first argument. Overall, the XSB compiler generates reasonable to good code, but some basic choices in the compiler are bad for performance, especially within an emulator: (1) the activation of a predicate can cause the creation of two choice points [2]; this slows down some benchmarks; in particular *sdda*, *meta_qsort*, and also to a lesser extent *boyer*; (2) the convention for in-lined built-ins is that their arguments are put in the argument registers 1 up to the arity of the built-in; sometimes up to three *movreg* instructions are generated before and after the call to a built-in; the twin calls to functor/3 in *boyer* are a good example of this inefficiency; (3) register allocation is far from optimal; this results in badly compiled arithmetic (among other things); e.g. the inner loop of *tak* contains at least five instructions that would have been avoided with a better register allocation; (4) XSB does not treat void variables in a special way; this slows down a.o. *zebra*.

The basic structure of the XSB compiler was not changed: variable classification, built-in calling convention, the indexing schema, register allocation etc. were not touched. We made five changes: (1) the generation of the *test_heap*

² a predicate specialization according to (partially instantiated) call patterns that appear in the same module

instruction - the entry point of each predicate in the current implementation of XSB which tests for heap overflow - was suppressed, and its functionality moved to the *call* and *execute* instructions; (2) in order to provide for native size floating point numbers and integers small changes were necessary: see later; (3) XSB uses specialized instructions for the atom `[]`; this specialization was switched off; (4) we have added some peephole optimizations for instruction compression and added a few instruction specializations (section 6.2); (5) we have specialized the XSB *switch_on_term* instruction to lists; this was important for testing an optimization referred to later as *switch* (section 6.2); this does not affect other issues: in our tagging schema, the list and compound test have the same cost.

XSB has only *truncated* integers (28 bits on a 32-bit machine). We implemented both truncated integers and *full* integers, i.e. 32 bits. The latter need changes in the generated code, since a 32-bit integer must always reside on the heap. We give the two instruction streams for the head of the clause *head(f(9, a))*. The overhead of full integers comes from its higher heap consumption and from extra emulator cycles. This will be clear in the benchmarks. Floating point numbers in dProlog are never truncated and always require the above transformation.

truncated ints	full ints
getstr f/2, A1	getstr f/2, A1
uninumcon 9	unitvar A2
unicon a	unicon a
	getint A2, 9

All versions of dProlog have the following characteristics: (1) separate stacks for environments and choice points; (2) no environment trimming; the XSB compiler does not generate the information to do it; the local stack grows towards the heap and environments are put upside-down (see [9]); this can result in less testing during trailing, but in an extra comparison when binding two free variables; (3) no tidying of the trail during cut; (4) no trail overflow testing; see section 6 for explanation and the effect on speed of trail overflow testing; all other overflow testing is done in software; (5) dProlog deals with hash collisions in the indexing code when using hashing by a try-retry-trust chain, just like XSB does; (6) floating point operations are not IEEE compliant.

3 The three term representations on the heap in dProlog

The `heap_vars` and `wam_vars` term representation on the heap is the same: the difference is in *where* permanent variables are initialized. `Heap_vars` does that always on the heap; `wam_vars` in the environment when possible. `Tag_on_data` and `parma_vars` are really different. So, we can speak of *three* heap term representations. The next figure shows the representation of the list `[a, b]` in the `tag_on_data` schema to the right and in the other schemas to the left. Each heap cell consists of a tag (P, S, L, A, I or F) and a pointer or value. An empty tag or value field means it is never inspected.

`Tag_on_data` uses one heap cell more per cons than the other representations. In the `tag_on_data` schema as implemented in BinProlog, the list constructor

is treated as any other constructor. We have instead chosen to specialize the representation of lists within the tag_on_data philosophy, because that results in exactly the same abstract machine instructions being executed in all variants of dProlog. The next figure shows compound terms, variables and numbers in $f(X, g(X), 666, 3.14)$. Note how the PARMA representation of two bound free variables creates a cycle instead of a chain of references.

Pointers (to the heap and local stack) are always aligned on a 4 byte boundary, so the tags on pointers (P, S and L) can be 2 bits without restricting the range of pointers. The dProlog tags are as follows (only the significant lower bits are given):

P (ref) = 00	S (struct) = 01	L (list) = 11
A (atom) = 0110	I (integer) = 1010	F (float) = 1110

The representation of floating point numbers is a bit unusual: the number is put on the heap, and its (F-tagged) offset from the start of the heap is used as its entry; the dashed pointer symbolizes the offset. This representation is also used for full integers. We have since long wanted to use such a representation (offsets instead of pointers) and now understand better the consequences. The main issues are: (1) some routines (like general unify) need to know the start of the heap; if the implementation needs to be re-entrant, it means that this start of heap must be passed as an argument; (2) copying terms (with numbers) becomes a bit more involved if the region one copies to (or from) is not contiguous (like for instance in findall buffers); (3) a sliding garbage collection algorithm needs an extra mark bit since numbers on the heap are not preceded by a header; (4) retaining sharing during copying is nearly impossible. We recommend this representation to nobody.

Dereferencing in the three representations is different. Without an attempt to show the best code in a particular context, here is basically the dereferencing of an object p :

parma_vars	if (ref(p)) p = *p;
wam_vars and heap_vars	while (ref(p)) { if (p == *p) break; p = *p; }
tag_on_data	while (ref(p)) { if (p == *p) break; q = p; p = *p; }

In `tag_on_data`, the value of q might be needed after `deref`: if after `deref`, p contains an S-tagged value, $q + 1$ points to the first argument of the structure.

In `parma_vars` dereferencing is a constant time operation. When two free variables are bound to each other, they must be tested for equality. [11] reports this as a drawback of `parma_vars`, but the truth is that – as was noted during the HAL project in which also the PARMA representation is used (see [5]) – testing for equality of two free variables in `parma_vars` might require less steps than in the other representations. Also, there exist programs for which the WAM is quadratic and PARMA linear in the input.

4 Comparing dProlog with other Prolog systems

We compare dProlog with other well-known and/or relevant systems on a set of benchmarks taken from [15]. The compiler of dProlog compiling itself (with output suppressed - input performed with *get0/1* and a reader in Prolog) is added as a medium sized more realistic benchmark: it consists of about 5000 lines of code. Times are always in 1/100s of a second. Sizes in tables are always in units of 4 bytes and represent the maximal stack usage during the running of all the benchmarks above the size figure in the same table; the code size includes also the compiler and toplevel. We repeated each benchmark in a failure driven way; the repetition factor is mentioned in the table: *sdda(1200)* means that a timing is shown for repeating *sdda* 1200 times. Each such repetition was performed four times; the first timing is always ignored (it is often an out-lier either way) and the smallest timing of the remaining three is reported. For dProlog, we show here only the mode *threaded with two opcodes + all other defaults* (see section 6) for `heap_vars`. We use an older SICStus version because it was our initial target and the most recent release (3.8.1) is significantly slower.

SICStus Prolog, Yap and XSB all implement `wam_vars`, while BinProlog obviously implements `tag_on_data`. SICStus and Yap (as dProlog) use gcc specific features, while XSB and BinProlog use only ANSI-C. Yap uses a single stack for choice points and environments, like the WAM.

5 Comparing the 6 basic modes of dProlog

Table 2 shows the results of running dProlog in its six basic modes on the standard set of benchmarks. The modes are indicated as *switch* (for a C switch), *jump* (for a jump table) and *threaded* (for the threaded implementation) and the suffix *1* or *2* indicates whether read-write propagation is done as in the WAM (1) or with two operation codes (2) as described in [4].

The stack sizes are the same for all modes and not included in the table. The code sizes for `switch1` and `jump1` must be equal, and also the code sizes for `switch2` and `jump2`. The fact that the code sizes for `switch1` and `switch2` are exactly the same is a coincidence: the instruction layout is such that each instruction has a spare byte.

The positive effect of using two opcodes (or two addresses in the case of threading) for read-write propagation is not as good as one might hope. We think the main reason is that the WAM non-read-write propagation with the

	dProlog 1	SICStus 3#5	Yap4.2.1	XSB 2.1	BinProlog 6.84
boyer(1)	40	34	32	95	106
browse(1)	48	47	30	95	147
cal(10)	77	69	68	140	105
chat(5)	45	55	43	77	98
crypt(200)	53	57	41	108	79
ham(2)	59	74	54	137	146
meta_qsort(125)	51	42	34	118	150
nrev(5000)	43	66	45	227	88
poly_10(10)	32	27	23	64	58
queens_16(2)	88	88	44	180	148
queens(10)	106	141	91	260	241
reducer(20)	19	13	10	37	35
sdda(1200)	40	33	29	70	75
send(10)	49	48	28	85	124
tak(10)	73	77	53	159	208
zebra(30)	82	82	63	139	173
compile(1)	776	914	739	1774	2074

Table 1. Comparing dProlog with other systems

S register, performs quite well on modern architectures with pre-fetching and branch prediction. The increased code size probably does not matter: this was confirmed by other experiments that are not reported on here.

It might seem strange that switch2 performs worse than switch1: we have tried three variants of switch2 and the results were always similar. Compared to switch1, switch2 requires extra jumps and/or increases register pressure even on instructions that have no read-write variant. This is not true when comparing jump1 with jump2 or threaded1 with threaded2.

6 The effect of features and optimizations in dProlog

The next tables only contain figures that relate to the heap_vars version of dProlog. The general setup is that the default settings are compared with a version in which one (or sometimes more) features are introduced or disabled. We first show variations in the implementation of the emulator itself (section 6.1) and variations in the generated abstract machine code (section 6.2).

6.1 Variations in the implementation of the emulator

Table 3 shows the variation of features in the emulator itself. An absent size figure means it is the same as for the default version. The explanation of the columns is:

- **default**: the default for dProlog is threaded, two opcodes, bind call, truncated integers, local pc = bx, with conditional trailing, no trail overflow check and with all instruction specializations and compressions
- **no bind call**: usually the call instruction refers to a predicate table to find the entry point of the predicate to be called; this makes the implementation of debugging and reconsult easier; in its default mode however, dProlog puts the address

	switch1	switch2	jump1	jump2	threaded1	threaded2
boyer	54	67	44	44	40	40
browse	81	95	51	50	48	48
cal	87	105	87	87	78	77
chat	54	64	48	47	46	45
crypt	70	84	57	56	52	53
ham	99	115	66	65	60	59
meta_qsort	76	89	56	55	52	51
nrev	130	157	45	43	46	43
poly_10	46	56	36	34	33	32
queens_16	118	141	100	98	89	88
queens	176	211	113	110	107	106
reducer	26	30	20	20	19	19
sdda	51	62	41	42	41	40
send	70	85	59	60	49	49
tak	101	125	78	78	73	73
zebra	103	113	85	83	83	82
code	48251	48251	48251	48251	73104	77038
compile	1094	1324	827	808	786	776
code	50202	50202	50202	50202	77826	81692

Table 2. The six basic modes of dProlog - heap_vars

of the entry point directly in the call (and execute) instruction; we name this *bind call*; supporting debugging and reconsult under this schema adds a little implementation complexity and is not fully done in dProlog; this column shows the effect of not binding calls: although binding calls looks a priori like an optimization, its effect varies and is not large; this is in part caused by the fact that in dProlog the lookup of the entry of a predicate, is very simple: only one indirection is needed

- **always trail**: this column represents unconditional trailing; its effect depends on the kind of benchmark (a benchmark where each potential trailing is actual, obviously benefits from deleting the test) but is never large except for *nrev*; we also give the maximal trail usage for comparison
- **trail overflow test**: by default, dProlog allocates enough space for the trail (as much as the heap in heap_vars and tag_on_data, six times as much in parma_vars, as much as heap and local stack together in wam_vars), so that trail overflow testing is not necessary³; however, the space penalty can be too high; so it is worth looking at the potential performance gain; the effect is quite high on some benchmarks, but overall small
- **full integers**: many implementations sacrifice the range of integers for the benefit of their compact representation: XSB represents integers with just 28 bits, as in the default mode of dProlog; dProlog can also run in a mode with 32 bit integers: this column shows the performance penalty both space and time wise; the relatively bad performance can be partly explained by the particular representation used for full integers, relying on a heap offset; another reason is the suboptimal abstract machine code for full integers (see section 2)
- **glob bx, glob bp, register, no reg decl**: as advised in [4] we assigned the WAM program counter (pc) to a hardware register; [4] does not report that there

³ this “optimization” came up during working with Paul Tarau but might be folklore

are several choices: one can reserve the hardware register locally to the emulator loop or globally in the whole of the implementation; and different choices as to which hardware register is used are possible; our default choice is to assign the program counter locally to the hardware register *bx*: *loc bx*; the meaning of *glob bx* and *glob bp* follows; we also tried just declaring the program counter locally as a register or giving no directions to the C compiler at all.

	default	no bind call	always trail	check tr overflow	full integers	glob bx	glob bp	register	no reg decl
boyer	40	40	40	41	45	40	40	41	41
browse	48	47	47	50	49	47	47	50	49
cal	77	78	79	79	91	78	78	77	78
chat	45	45	45	45	47	45	45	44	44
crypt	53	52	54	51	62	52	52	52	51
ham	59	61	62	62	60	59	59	64	65
meta_qsort	51	51	52	52	51	51	51	51	51
nrev	43	46	53	44	44	43	43	52	51
poly_10	32	32	33	32	33	31	31	31	31
queens_16	88	90	90	88	108	88	88	90	90
queens	106	105	107	103	108	103	103	107	107
reducer	19	19	19	19	19	18	18	19	19
sdda	40	41	41	40	41	40	40	41	40
send	49	49	50	55	60	49	49	54	54
tak	73	74	72	74	93	75	75	74	74
zebra	82	83	83	86	86	81	81	85	85
heap	448296	-	-	-	674911	-	-	-	-
trail	57677	-	403043	-	-	-	-	-	-
code	77038	-	-	-	77761	-	-	-	-
compile	776	779	787	787	844	769	796	785	785
heap	1369456	-	-	-	5972416	-	-	-	-
trail	227623	-	854611	-	-	-	-	-	-
code	81692	-	-	-	82198	-	-	-	-

Table 3. The effect of emulator properties in dProlog

The table indicates that assigning the WAM program counter to a global *bx* register (the Yap choice is global *bp*) performs best, followed closely by the local *bx* register. One should not take this as a general truth: it will depend on factors beyond control like the C compiler and the particular way of writing C code in the emulator. Rather one should take this as an indication that exploring the alternatives might be worthwhile. We made local *bx* our default because it performs well while being local. Note that also just declaring the program counter as a register without assigning it to any particular hardware one, performs quite well, with the additional advantage that it is fully ANSI.

6.2 Variations on the abstract machine code

We report here on variations in the generated abstract machine code: the columns in table 4 show the timings when one or more instruction compression or specialization is turned *off*. Note that even though some instructions are not generated when some optimizations were turned off, all instructions are present in the emulator at all times.

The XSB compiler performs one particular instruction compression during its peephole optimization: the sequence *getlist,unitvar,unitvar* is compressed to *getlist_tvar_tvar*; *append/3* in particular benefits. Table 4 mentions it as *getlist*. We implemented two more instruction compressions and two specializations:

- **dealloc**: compresses the instruction sequence *deallocate,proceed* into one (new) instruction *dealloc_proceed*, the sequence *deallocate,execute* into *dealex* and *deallocate,builtin,proceed* into *builtin_dealloc_proceed*;
- **uni**: two subsequent *unitva** instructions are compressed to one instruction; this leads to four instructions *uni_tvaX_tvaY* where X and Y can be *r* or *l*; Yap also performs this compression;
- **try**: specialized versions for the *try*, *retry* and *trust* instructions are generated for predicates with arities 2 and 3; Yap does this for 0 up to 4;
- **switch**: this specialization exists in Yap and in some other forms in other implementations; if the list exit of a *switch_on_list* instruction points to the corresponding *getlist* instruction, a specialized *switch* instruction is generated which in the list exit jumps directly to the read-mode of the instruction *following* the *getlist*

The specialization *try* and compression *dealloc* are each independent of all the other transformations; they are performed in the order: *switch, getlist, uni*. A *switch* specialization prevents a *getlist* compression of the corresponding *getlist* instruction, but still allows the *uni* compression of the instructions after the *getlist*. Likewise, a *getlist* compression clearly prevents the application of *uni* compression on the unify instructions after the *getlist*. Since the interaction of these three transformations is tricky, we give all seven possibilities (together with the default, that makes 8).

7 Comparing four term representation schemas

Table 5 shows the execution times and some stack sizes for the four term representations mentioned in section 3. Every version here has the same default settings as in 6.1. Only heap and trail sizes are mentioned, as the choice point stack, local stack and the code size are the same for all four versions.

A priori, one can reason that *heap_vars* and *parma_vars* must consume the same amount of heap, while *tag_on_data* potentially consumes more (due to the absence of an optimized list representation) and *wam_vars* consumes less because some objects only ever live on the local stack. For the trail, one expects *heap_vars* and *tag_on_data* to trail exactly the same cells, while both *wam_vars* and *parma_vars* trail potentially more. Indeed, when a variable is bound to a non-variable in *parma_vars* all the cells in the chain representing the variable are bound and subject to trailing. Note also that in *parma_vars* trailing one cell

	default	try	dealloc	switch	getlist	uni	switch getlist	switch uni	getlist uni	switch getlist uni
boyer	40	42	40	40	40	40	40	40	40	40
browse	48	49	49	47	49	50	49	48	50	51
cal	77	80	77	77	77	77	77	77	77	77
chat	45	45	45	44	45	45	45	46	46	46
crypt	53	53	53	54	55	52	54	54	55	55
ham	59	66	59	59	66	61	66	61	70	70
meta_qsort	51	54	51	50	51	54	51	54	54	54
nrev	43	43	43	50	43	54	50	62	54	62
poly_10	32	32	32	32	32	33	32	33	33	33
queens_16	88	89	88	92	89	90	92	92	92	94
queens	106	115	105	106	106	107	108	108	113	116
reducer	19	19	19	19	19	19	18	19	20	19
sdda	40	41	41	40	41	40	41	40	41	41
send	49	49	49	46	49	49	49	49	49	49
tak	73	73	73	73	73	73	73	73	73	73
zebra	82	82	82	82	84	83	84	82	87	87
code	77038	77474	77477	77005	77317	78355	77317	78289	78913	78913
compile	776	785	782	776	782	789	784	787	796	796
code	81692	82236	82209	81641	82184	83177	82184	83075	84161	84161

Table 4. Variations of abstract machine code in dProlog

needs two entries on the trail stack. It is clear that `wam_vars` must (conditionally) trail on globalizing a permanent variable. But even for permanent variables that are never globalized, `wam_vars` potentially trails more than `heap_vars`.

The figures in table 5 are in accordance with the above reasoning. However, for the smaller benchmarks, the difference in trail usage between `parma_vars` and the others is very small, which confirms the common knowledge that `var-var` bindings are uncommon ... for small benchmarks. The *compile* benchmark shows a different picture.

The huge difference between the heap usage of `wam_vars` and the others is striking. Even though 99% of the difference is caused by the double calls to `functor/3` and `arg/3` in *boyer*, and a better built-in calling convention would reduce the difference to almost zero, in general this extra heap consumption is a drawback for any schema that always globalizes variables.

In order to reduce the potential bias in common operations towards one particular term representation, we have tried for each version several ways to write (in C) the dereference operation: each version is run with the best (sometimes more than one) we found for it. For the `wam_vars` version, we have specialized the trailing test whenever possible.

`Tag_on_data` is clearly the loser. On the small benchmarks, `wam_vars` comes out as the winner with `heap_vars` a close second; `parma_vars` suffers badly from its disadvantages on some of the small benchmarks. On *compile* `parma_vars` wins, albeit with a small margin: the high number of trail entries (almost triple of

	heap_vars	wam_vars	parma_vars	tag_on_data
boyer	40	38	40	42
browse	48	46	46	53
cal	77	75	77	77
chat	45	44	46	47
crypt	53	52	52	54
ham	59	58	60	64
meta_qsort	51	48	54	53
nrev	43	44	47	56
poly_10	32	31	34	34
queens_16	88	85	90	90
queens	106	103	113	113
reducer	19	18	21	21
sdda	40	42	43	44
send	49	45	52	49
tak	73	71	75	73
zebra	82	82	97	90
heap	448296	144069	448296	448366
trail	57677	58635	115366	57677
compile	776	776	773	840
heap	1369456	862479	1369505	1647279
trail	227623	237689	642356	227633

Table 5. Four schemas for term representation

heap_vars) indicates that more var-var bindings occur here and that is where parma_vars is at its strongest.

8 Related work

[4] is worth mentioning because it gives good advice on implementing an emulator for Prolog. We have followed it to a large extent and we did benefit a lot from it. In fact, most of the advice we put to scrutiny turned out to be excellent, even though we were working in the context of a different abstract machine code generator and a different stack layout. So, our work is in line with [4], redoing partly its experiment in a different setting. [4] does not stress the importance of the quality of the generated machine code, probably because its compiler - although not sophisticated - is of sufficient quality. We think that a decent compiler is very essential for speed. Also, [4] does not provide any insight in the importance of the double opcode schema for read-write mode propagation, which we do in section 5. Finally, [4] was not aimed at providing empirical data on alternative term representations. In fact, no such work exists as far as we know. Alternative term representations were always implemented in combination with other features which can obscure in unknown ways the effect of the choice of the term representation.

As far as “paper” comparisons is concerned: [11] describes nicely the issues involved in implementing PARMA variables and relates them to the WAM, but as the authors note themselves, it is impossible to conclude with any confidence

how within the same implementation these two alternatives would compare. [13] compares only minimally tag-on-data with the usual WAM representation and focuses on the issue of term compression and stack usage.

9 Conclusion and future work

We have gathered experimental data on how four term representations that are easily compatible with the WAM compare and about choices in the implementation of the abstract machine compiler and emulator. One conclusion is that the `tag_on_data` representation seems the least attractive from the performance point of view. Its main attraction is in the fact that since pointers are tag-less, the address space is not restricted and that one can have up to 32 tag bits in the data, a luxury that can make any implementor's mouth water. Admittedly, we have not implemented term compression, but since [13] does not report a significant speedup related to term compression and sometimes even a slowdown, we feel that our conclusion holds. The situation with `parma_vars` is more complicated: first of all, one must realize that for admittedly artificial examples, `parma_vars` can perform arbitrarily better than the other version. Moreover, while `parma_vars` performs worse on the smaller benchmarks, it performs better on the one benchmark that makes more use of the full potential of the Prolog variable. The figures give a slight preference to `wam_vars` time wise over `heap_vars`, and space wise `wam_vars` clearly wins, modulo the remarks made in section 7. However, other considerations could make one prefer `heap_vars`. E.g. in the context of SLG-WAM, the trail *must* be tidied on cut as far as the local stack trailed entries goes. But tidying the trail has bad worst case complexity. Then choosing for a `heap_vars` schema seems suddenly quite attractive given the small time penalty. In any case, such tradeoffs should be made by the implementor and might be based on the figures here or after having seen the actual implementation of dProlog.

We have put a lot of effort into avoiding bias towards any one version of dProlog. In particular, we have made sure that in all versions the same abstract machine code is executed instruction by instruction, that the stack layout is the same, that the number of times the general unification is called is the same, that the indexing (in particular the occurrence and treatment of hash collisions) is the same etc. ⁴ When comparing the four term representation each within a different context, it will be next to impossible to keep all the above factors the same. However, one might also see this uniformity as a drawback: each of the term representations might benefit in a different way from a particular action, e.g. dereferencing during a particular instruction might be good in `wam_vars` but not in `parma_vars`. We are aware of the fact that given much more time, we can improve each of the four versions; but we are not aware of having favored any

⁴ the number of times dereferencing is performed is *not* the same: `wam_vars` and `parma_vars` have by necessity more than `heap_vars` (about 6% for the benchmark suite); `tag_on_data` has marginally less because of a different optimal sequence of instructions in the `switchonlist` instruction

particular one. We have even put effort in providing each version with its best dereferencing macro and tag testing sequence.

The figures in section 6 give a good indication on which specializations, compressions and emulator optimizations are interesting and also uncover weaknesses in the XSB compiler. Our experience has been used in a partial redesign of XSB.

The experiments might show different results on other processors. Since dProlog is portable the experiments can be redone for other platforms. Still, some extra work is required, mainly because one must assign hardware registers to WAM registers carefully.

Like [4] we found that a fast emulator is a combination of the following factors: (1) a good discipline for writing C code, (2) selective use of gcc features, (3) a decent basic abstract machine code generator, (4) some instruction compression, (5) some instruction specialization. In dProlog, we were missing mostly item 3: while [4] claims for instance that a sophisticated register allocation (see e.g. [8]) is not needed, we found that bad register allocation as in XSB, is a real drawback. Also the following two points are very important in our opinion: single level indexing is to be preferred over double level indexing (see also [2]); and secondly the basic mechanism for (in-lined) built-ins must suit the emulator: the XSB mechanism leads to more register shuffling and executed emulator cycles.

One can wonder whether redoing the effort described in [4] in a different context is relevant: we think that redoing it in exactly the same context would have been meaningless. Our work shows that the advice of [4] is useful elsewhere.

Directly related to dProlog, there is ample work for speeding up the implementation. Apart from issues mentioned before, we think two more points are worth exploring further: specialization of built-ins (e.g. functor/3 is almost always called in one of two basic modes and the compiler usually knows which because of the var-val distinction) and instruction compression over predicate boundaries: the *call* or *execute* instructions transfer very often to the same instructions. We experimented shortly with both, but gathered until now insufficient systematic data to report on them.

More generally, there are still many things worth investigating within a fixed implementation context: there is for instance no hard comparative data on entirely different tagging schemas, e.g. like the one SICStus Prolog is using. Other issues are the implementation of general unify (with support for cyclic terms or occurs check), optimal abstract machine code for arithmetic, calling of built-ins, the use of type or mode information in an emulator (as in [10]) and many more. Also different allocation schemas within a WAM-like implementation can and should be experimented with, together with different garbage collection strategies and principles. Inspiration here might come from the implementation of other programming languages. The problem with memory management experiments is however that an entirely new type of benchmarks is needed.

We will definitely continue using dProlog for experimenting and gathering information on the implementation of WAM variants and beyond.

Acknowledgements

We thank the XSB team – in particular David S. Warren and Kostis Sagonas – for making their compiler and system available, and for explanations. We are also grateful to Vitor S. Costa and Paul Tarau for making their systems freely available and for help in using them.

References

1. H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. The MIT Press, Cambridge, Massachusetts, 1991.
2. 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
3. M. Carlsson. *Design and Implementation of an Or-Parallel Prolog Engine*. PhD thesis, The Royal Institute of Technology (KTH), Stockholm, Sweden, Mar. 1990.
4. V. S. Costa, *Optimising Bytecode Emulation for Prolog*. Proceedings of PPDP'99, LNCS 1702, Springer-Verlag, 261-277, September, 1999.
5. B. Demoen, M. García de la Banda, W. Harvey, K. Marriott, and P. Stuckey. *Herbrand Constraint Solving in HAL*. Proceedings of the International Conference on Logic Programming 1999, Las Cruces, New Mexico, ed. D. De Schreye, MIT Press, pp. 260-274
6. B. Demoen, G. Janssens, H. Vandecasteele. *Executing Query Flocks for ILP*. Proceedings of BENELOG'99, Maastricht, 5 November 1999
7. B. Demoen, K. Sagonas. *Heap Garbage Collection in XSB: Practice and Experience* Proceedings of the Second Int. Workshop on Practical Aspects of Declarative Languages, Boston, Jan. 2000, pp. 93-108
8. G. Janssens, B. Demoen, A. Mariën. *Improving the register allocation in WAM by reordering unification*. Int. Conference & Symposium on Logic Programming, Seattle, Washington aug 1988, pp. 1388-1402
9. A. Mariën, B. Demoen. *On the management of E and B in WAM*. Proceedings of the North American Conference on Logic Programming, Cleveland, Ohio, oct 1989, pp. 1030-1047
10. Phuong-Lan Nguyen. *Optimisation du Code produit par un Compilateur Prolog*. Rapport de DEA, ENSIMAG, Laboratoire de Génie Informatique, Grenoble, 1988
11. T. Lindgren, P. Mildner, and J. Bevenmyr. *On Taylor's scheme for unbound variables*. Technical report, UPMail, October 1995.
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. P. Tarau and U. Neumerkel. *A Novel Term Compression Scheme and Data Representation in the BinWAM*. Proceedings of Programming Language Implementation and Logic Programming, sept. 1994, Springer, LNCS 844, pp. 73-87
14. A. Taylor. *PARMA-bridging the performance gap between imperative and logic programming*. *Journal of Logic Programming*, 29(1-3), 1996.
15. P. Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming ?* Report 90/600, UCB/CSD, Berkeley, California 94720, Dec 1990.
16. D. H. D. Warren. *An Abstract Prolog Instruction Set*. Technical Report 309, SRI International, Menlo Park, U.S.A., Oct. 1983.
17. See <http://www.cs.sunysb.edu/~sbprolog/>.