

# Experiments in WAM emulators and term representations.

Bart Demoen

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

Phuong-Lan Nguyen

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

## Abstract

Successful Prolog implementations are often based on WAM. While WAM is a very good starting point, it leaves certain issues open and also allows within its framework a lot of variations, optimizations and extensions, e.g. regarding the term representation, the instruction set, the memory organization, the represented data etc. It is therefore no surprise that different implementations of Prolog have successfully exploited in different ways the freedom left by WAM. While successful within their particular context, it is difficult to assess the merit of a particular variation. This work describes an attempt to do exactly that: Within one basic implementation - dProlog, based on WAM and using the XSB compiler - four term representations were implemented and compared. Each of these representations has been used by some successful system, so it is worthwhile to have empirical data on their performance while keeping all other things equal. Still within dProlog, we also report on different choices in the emulator for WAM. Since dProlog is a reasonably efficient Prolog system, it is relevant to measure the impact of these choices.

## 1 Introduction

WAM<sup>1</sup> has been the basis for many Prolog systems, also the most successful ones. WAM leaves open certain issues like the implementation of cut, dynamic code etc. but even when it specifies other issues, they can be varied on and one can consider such variations often still as 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 that is of interest to this paper - concerns the term representation itself: e.g. while WAM initializes permanent variables on the local stack when possible, 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 WAM and this makes dereferencing a constant time operation; BinProlog [13] employs a *tag on data* schema instead of the WAM *tag on pointer* schema. Each of these variations on WAM has been successful to a certain extent within its own context: BIM\_Prolog used native code generation (which was new at the time); BinProlog binarizes clauses before compiling them [12]; PARMA [14] and AQUARIUS [15] rely on abstract interpretation for their top speed. Because the context of the implementation of the different term representations has been different, it is not at all clear how these term representations

---

<sup>1</sup>we assume knowledge of WAM; see for instance [1]

really compare, i.e. there has been no empirical study of the impact of changing within one efficient system the representation of e.g. tag-on-pointer to tag-on-data, *all others things kept equal*.

This is exactly what we aimed to do and present in this paper: set up an experiment which compares directly the four above mentioned term representations in the same implementation.

Such an experiment only makes sense in the context of a reasonably complete and fast Prolog implementation. So we could have started from an existing efficient system like Yap [4] or SICStus [3] and do the experiment there. However, because of our prior involvement with XSB (see e.g. [7]), and also because changing such complete systems tends to be very time consuming, we decided to start almost from scratch: we borrowed the XSB compiler <sup>2</sup> for the generation of abstract machine code (XSB is largely WAM based) and we built a new emulator. This had the additional advantage that we could benefit from the experience reported in [4] and partly redo the experiment reported on there. Also, our involvement with inductive learning [6] required us to write a Prolog system from scratch at some point anyway.

We named the resulting Prolog system **dProlog**: it is complete enough to bootstrap itself (compiler, toplevel, reader and a number of builtins written in Prolog) and allow rudimentary box level debugging, but it is not more complete than needed for the experiment; in particular dynamic predicates, the findall family of builtins, exception and interrupt handling, modules and garbage collection have not been implemented: such features can be added without affecting the rest of the system speed wise <sup>3</sup> and therefore their absence does not influence our findings.

So one basic choice was to stick largely with the XSB compiler: section 2 contains more information on this issue. The other choice was to remain in the emulator business: the results of our experience might not immediately be valid for native code generating systems. But it clearly doesn't make sense to study the impact (speed or space) of changes in a system that is below standard: e.g. the difference between two tagging schemas might never show in the slow emulator of BIM\_Prolog.

This meant we set the standard quite high given the sometimes bad code that the XSB compiler generates: we wanted dProlog to be in the same ball park as SICStus Prolog emulated with all bells and whistles (i.e. using gcc extensions). SICStus Prolog does not have the fastest emulator around these days: Yap [4] beats SICStus Prolog consistently. So we decided to use similar ideas as Yap in the implementation of the emulator. As a consequence dProlog performs as good as Yap for certain benchmarks - especially the smaller ones - but given our choice of XSB for generating the code to be emulated, dProlog has no chance to get on par with SICStus Prolog. Still, we felt that the initial goal was met, at least on the hardware we were experimenting on: Pentium II, 260 MHz, 128Mb. We report on the aspect of performance compared to other systems in section 5, mainly because it shows that our emulator is of sufficient quality to make the results of the further experiments relevant.

Apart from the other motivations for doing this effort of implementing a new system from scratch, we wanted to satisfy some private curiosity regarding an unusual term representation we used for integers and floating point numbers, and a different layout of environments: see further for a description of these. Also the fact that a new emulator based on the XSB compiler would give insight in to what extent the XSB emulator can be improved played some role.

The basic version of dProlog has the following characteristics:

- separate stack for local and choice point stack

---

<sup>2</sup>including the reader

<sup>3</sup>this might not be clear to some readers, but we have no time to dwell on this now

- no environment trimming: the XSB compiler does not generate the information to do it; moreover, the local stack grows towards the heap (as in XSB) but environments are put upside-down, so that after a new environment is pushed, the top of local stack equals E (see [9]); this also results in less testing during trailing, but in an extra comparison when binding two free variables
- no tidying of the trail during cut <sup>4</sup>
- no trail overflow testing: see section 7 for more explanation and the effect on speed of trail overflow testing; all other overflow testing (heap, local stack, choice point stack) is done in software
- like XSB, dProlog always generates a file with the abstract machine code when it compiles a program; this saves time on later consults - for ease of development, this file is human readable, which slows down consulting it, but has been of great help during the experiment
- like XSB, in indexing code with a hash table, dProlog deals with hash collisions by a try-retry-trust chain: this explains some of the (relatively) bad benchmarks

The basic structure of the XSB compiler was not changed: variable classification <sup>5</sup>, builtin calling convention, indexing (which is suboptimal as it is a two level indexing schema - see [2]), register allocation <sup>6</sup> etc. was not touched. We made three kinds of changes: (1) the functionality of the *test\_heap* instruction - the entry point of each predicate in the current implementation of XSB - **testing for heap overflow** was shifted to the *call* (and *execute*) and consequently the generation of *test\_heap* was suppressed; (2) to provide native size integers (and floating point numbers) small changes were necessary: see section 2; (3) we have added a few peephole optimizations for instruction compression and added a few instruction specializations: we will come back to these later and report on their impact in section 7.

We also wanted to experiment with and report on 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 **plain\_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 will also in detail report on the effects (time and space) of these six modes in section 6.

As in [4] we have assigned the program counter to a hardware register. We will show the effect of assigning it to different hardware registers and of making it local or global - see section 7. We report there also on instruction compression, fast call, conditional trailing, trail overflow checking, 28 versus 32 bit integers and instruction specialization.

Our first version of dProlog always globalizes permanent variables on their first occurrence: we will later refer to this version as the **heap\_vars** system.

After the first version of dProlog (**heap\_vars**) was finished with all the above variations, we created three more versions: we will refer to these versions as **wam\_vars**, **parma\_vars** and

---

<sup>4</sup>tidying the trail during cut can affect the complexity of a program in a bad way - see the appendix

<sup>5</sup>XSB no notion if void variables

<sup>6</sup>which is also far from optimal

**tag\_on\_data.** They differ from dProlog in only one aspect: `wam_vars` initializes variables as undefs on the local stack whenever possible, exactly as in WAM, and thus knows **unsafe** variables (see [16, 1]; `parma_vars` uses the variable representation as in [14]; `tag_on_data` uses the representation as in BinProlog [13]. `Parma_vars` and `tag_on_data` also always globalize permanent variables. The reason for this choice in `parma_vars` is that the report in [11] contains several pages on the intricacies of having variables also in the local stack and they scared us. For `tag_on_data`, we made the always globalizing choice because BinProlog, the only other Prolog system using the tag-on-data representation, doesn't even have a local stack. These different representations are shortly explained in section 3. We will report on the impact of these changes in section 8. To keep the code free from too many conditionals, we have chosen to put the variations in different copies of the source code of dProlog: this was a good choice because often a different sequence of dereferencing and testing is optimal for the variations. And in practice, there are only 3 files that are not shared by all versions.

We will start by briefly introducing certain aspects of the XSB compiler, dProlog, the tagging schema and the term (or variable) representation variants in section 3. We end with a conclusion and description of work that we think is worth doing in this context.

## 2 The XSB compiler

As we have taken the XSB compiler for producing the abstract machine code, it is worth giving some of the characteristics of this compiler.

The naming convention of the XSB compiler for instructions can be mapped easily to the usual WAM terminology: instructions involving variables either refer to a permanent variable (living in an environment) with the letter *p* or to a temporary variable (living in a X register) with the letter *t*. As usual, a first occurrence in the code is indicated by a suffix *var* and all later occurrences by the suffix *val*. So we have instructions like *putpvar, gettval*. The PUT\_UNSAFE instructions is named *putuval*. We'll stick to the XSB naming convention.

We want to sketch briefly here the strengths and weaknesses of the XSB compiler and the changes we made so that it performed for us only what we wanted and not more. 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*<sup>7</sup> was switched off. Overall, the XSB compiler generates reasonable code. In particular we liked the *switch\_on\_term* instruction, which is generated when there are only two alternatives, one being with an argument that is atomic, the other compound (including list) and also the implementation of the Prolog cut is nice. On the other hand, 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. in-lined builtins have the convention that their arguments must be put in the argument registers 1 up to the arity of the predicate; sometimes up to three *movreg* instructions are generated before and after the call to a builtin; the twin calls to `functor/3` in the `boyer` benchmark are a good example of this inefficiency

---

<sup>7</sup>a predicate specialization according to (partially instantiated) call patterns that appear in the module

3. if-then-else is compiled without a choice point only for arithmetic tests; other simple tests like e.g.  $var(X)$  do cause a choice point
4. register allocation is far from optimal; this results in badly compiled arithmetic (among other things); e.g. the inner loop of the tak benchmark contains at least five instructions that would have been avoided with a better register allocation
5. XSB does not treat void variables in a special way; this slows down for instance the zebra benchmark

The XSB compiler can generate indexing for any argument (depending on declarations); in order not to skew the experiment when comparing with other systems, we have disabled the index declaration. At the same time we have specialized all indexing instructions to the first argument.

XSB does a very peculiar instruction compression (see section 7) which one benchmark benefits very much from: `nrev`. We will show how it affects overall speed and we will show the impact of other common compressions.

The XSB system features *truncated* integers (28 bits on a 32-bit machine) and similarly for floating point numbers: we have decided to implement both such truncated integers and also *full* integers, i.e. 32 bits. This necessitates changes in the generated code, since a 32-bit integer must reside on the heap because it cannot have a tag. Below we give the two instruction streams for the head of the clause  $head(f(9, 10))$ . in the case of truncated integers and full integers:

usual XSB code	code for full ints
getstruct f/2, A1	getstruct f/2, A1
uninumcon 9	unitvar A2
uninumcon 10	unitvar A3
	getint A2, 9
	getint A3, 10

So there is not only an overhead because full integers take more heap space, but also in the emulator because full integers require more emulator cycles. This effect will be clear in the benchmarks. Floating point numbers in dProlog are never truncated and consequently always require the above transformation.

### 3 The three term representations on the heap in dProlog

The `heap_vars` and `wam_vars` term representation on the heap is exactly the same: the implementations differ in the exact place of initialization of a permanent variable and then deal with the consequences.

The three different term representations on the heap can be illustrated well with some pictures: Figure 1 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.

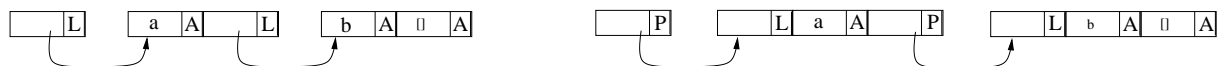


Figure 1: The list  $[a, b]$  on the heap

It can be seen that `tag_on_data` uses one extra heap cell 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 allows the use of exactly the same intermediate instructions executed in all variants of dProlog. See further for more explanation on the specialized list representation for `tag_on_data`.

Figure 2 illustrates compound terms, variables and numbers by means of the term  $f(X, g(X), 17, 3.14)$ .

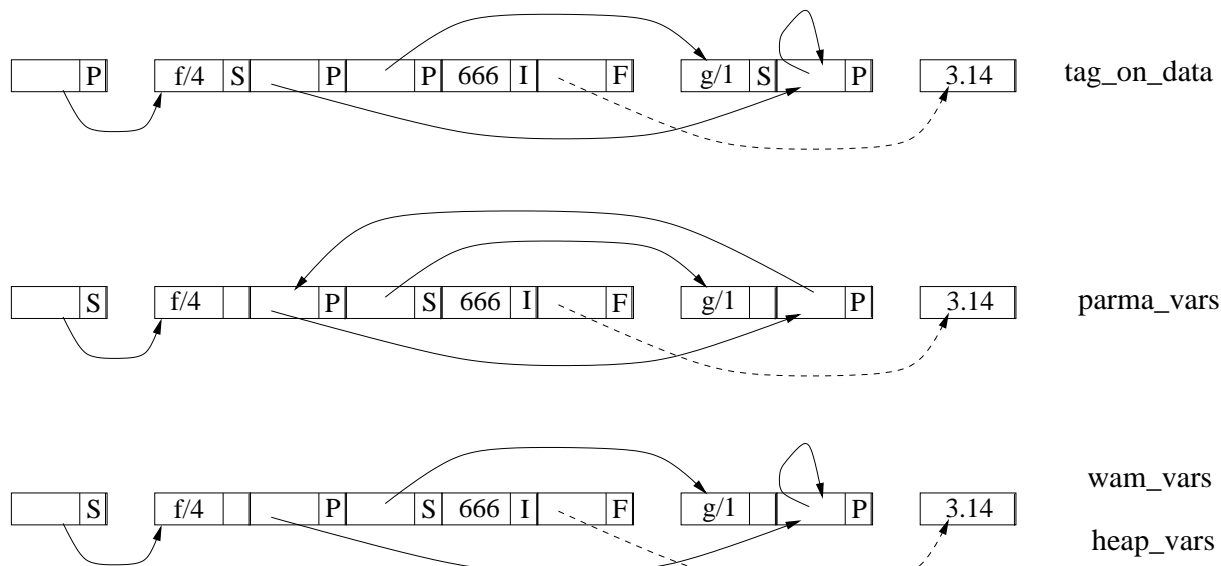


Figure 2: Compound term and numbers on the heap

The tags are as follows (only the significant lower bits are given):

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

Pointers (to the heap) are always aligned on a word (4 byte) boundary, so the tags on pointers (P, S and L) can only be 2 bits (since we don't want to restrict the address range of pointers).

Note how the PARMA representation of two bound free variables creates a cycle instead of a chain of references.

The representation of floating point numbers is a bit unusual: the number is put on the heap, and its (tagged) offset from the start of the heap is used as its representation; the dashed pointer symbolizes that. Such a representation is also used for full integers. Using two more bits for the tags of floating point numbers (and full integers) restricts the size of the heap to 1 Gb. We have since long wanted to use such a representation (offsets instead of pointers) and now understand better the consequences. The main issues with this representation are:

- 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
- 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)

- a sliding garbage collection algorithm has an extra problem since numbers on the heap are not preceded by a header; this can be solved by an extra mark bit
- retaining sharing of floating point numbers (and full integers) during copying is nearly impossible

Dereferencing in the three representations is different. Without an attempt to show the best code in a particular context, here is basically what needs to happen when dereferencing 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; }

The following is important: `deref` is constant time in `parma_vars`; in `tag_on_data`, the value of  $q$  is needed after `deref`: if after `deref`,  $p$  contains an S-tagged value,  $q + 1$  points to the first argument of the structure.

A final word about dereferencing in the PARMA context: when two free variables are bound to each other, they must be tested for equality. [11] reports this as a drawback, 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` requires possibly less steps than in the other representations. Also, there exist programs for which WAM is quadratic and PARMA linear in the input. Both are explained by explained in the appendix.

## 4 Maintaining dProlog while experimenting

From the start of this project, we knew we wanted to experiment with different issues, while keeping good control over the code and its portability. Probably the main issue here is that it must be easy to modify the layout of instructions and add new instructions. There are certainly several good ways to do this. We found the following very useful.

Information about each instruction is kept as Prolog facts. As an example the fact

```
instr(jumpz, [dlong(R),dlong(Lab)], [opcode,areg(R),label(Lab)]).
```

means that the `jumpz` instruction in the generated abstract machine code file has two operands, both of type `dlong`; and that the internal format (once loaded) has an operation code, a number that represents the argument register (`areg`) and a label in the code; note the use of logical variables to connect the file format with the byte code format.

Additional - and orthogonal - facts specify properties about types like `dlong` (there is also `dfloat`, `byte` and `short`) and the objects like the WAM argument registers. E.g. the facts:

size(byte,1).	alignment(byte,1).	basic_type(areg(-),byte).
size(dlong,4).	alignment(dlong,4).	basic_type(label(-),dlong).

specify that a *byte* is one machine byte, a *dlong* 4 machine bytes, the alignment the implementor wants to give them (or is necessitated by the hardware), and that an argument register is represented by a byte, and a code label by a `dlong`. The `size/2` and `alignment/2` facts are machine dependent (the above work on Pentium and SPARC; there is a different set for e.g. Alpha). The `basic_type/2` facts are related to byte code layout choices and limitations in the abstract machine, e.g. with the facts above, there can be at most 256 temporary variables in a clause.

From these Prolog facts, parts of the dProlog implementation are generated; e.g. the loader (a C function) itself is completely generated, as are all the macros needed for fetching arguments from the byte code. Also an empty emulator is generated, i.e. one that contains local C variable declarations, the instructions to fetch the operands and the transfer of control: such code must be completed with the actions to be taken for the instruction, and usually for an optimized version of the code, the generated code needs changes, but it was a great help in getting the emulator off the ground.

## 5 Comparing dProlog with SICStus, Yap, XSB and BinProlog

Just to show the relative strengths and weaknesses of dProlog, we give a comparison with other well-known and/or relevant systems. The first set of benchmarks is taken from [15]. All times are in milliseconds. Often, the original benchmark produces timings that are too small to be significant; e.g. `sdda` produces 0 or 10 msec. We therefore repeat each benchmark in a failure driven loop; the repetition factor is mentioned in the table: `sdda(1200)` means that the timing is shown for repeating `sdda` 1200 times. Each such repetition was performed 4 times and the smallest timing is reproduced below. For dProlog, we show here only the mode *threaded with two opcodes + all other defaults* - see section 7 for `heap_vars`. All timings were done on a Pentium II, 260MHz, 128 Mb. Timings are given in 1/100 sec<sup>8</sup>, and whenever we give sizes of particular areas, they are in 4-byte words - except code sizes which are in bytes. The benchmarks always run after each other in the same process. When sizes are given, they represent the state after all the benchmarks above the size figure in the same table have run.

	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

Table 1: Comparing dProlog with other systems

We also show the results of a set of very artificial benchmarks: we used these to find weaknesses in dProlog. In the table, each benchmark has as comment what it actually tests. BinProlog performs *super compilation* on certain types of predicates. In particular, predicates that look like

---

<sup>8</sup>the precision of `getrusage`

append/3 are not executed by the regular emulator in their deterministic mode, but by a C function. Swapping the last two arguments of append/3 prevents this super-compilation: this was done in snrevswap below.

	dProlog	SICStus	Yap	XSB	BinProlog	testing
indexa	10	19	9	75	67	indexing on atoms
indexa2	71	80	74	205	932	double level indexing on atoms
indexf	25	24	26	113	97	indexing on compound terms
list	34	61	45	193	68	list traversal
struct	51	80	58	251	67	structure traversal
metacall	39	399	85	359	86	meta call of atomic goal
plusli	44	114	125	150	392	integer + 1
pluslf	133	387	321	319	867	float + 1.0
snrev	58	83	61	269	89	nrev on non-list
snrevswap	60	82	62	269	157	snrev with args2-3 swapped

Table 2: Artificial benchmarks

Under XSB, the benchmarks indexa, indexa2 and indexf were compiled with the goal specialization turned off, because it interfered very much with what we wanted to measure. It is striking that the good results of dProlog on the artificial benchmarks, does not scale to the larger ones: this is at least partly caused by the XSB compiler which for small clauses often produces perfect code, but for the larger ones degrades.

It is perhaps worth mentioning that 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.

## 6 Comparing the 6 basic modes of dProlog

The next table 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 jumptable) and **threaded** (for the threaded implementation) and the suffix **1** or **2** indicates whether read-write propagation is not done as in WAM (1) or with two operation codes (2) as described in [4]. We also give the code size (in bytes) because the code space usage for the different modes can be quite different. However, a priori one can say that 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 alignment and operands of every instruction are such that each has a spare byte.

It can be seen that 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 there are two reasons for that: first of all, the code size grows because of the second address. This results in more memory accesses and decreases speed. The second reason is that the WAM non-read-write propagation with the S register, performs quite well on modern architectures with pre-fetching and branch prediction. 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.

	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	193004	193004	193004	193004	292416	308152
indexa	35	46	11	11	11	10
indexa2	133	165	79	79	75	71
indexf	60	73	28	25	27	25
list	109	138	35	34	35	34
struct	158	199	60	55	52	51
metacall	97	117	45	46	41	39
plusli	104	128	48	48	45	44
plus1f	189	233	138	136	129	133
snrev	166	197	65	64	63	58
snrevswap	164	197	66	64	65	60
code	202168	202168	202168	202168	305576	321808

Table 3: The six basic modes of dProlog - heap\_vars

## 7 The effect of features and optimizations in dProlog

The next tables only contains 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 disabled or introduced. We first show variations in the implementation of the emulator itself, then variations in the generated abstract machine code and finally variations in the size of certain parts of the loaded abstract machine code.

### 7.1 Variations in the implementation of the emulator

Table 4 shows the variation of features inside the emulator itself. The explanation of the columns in table 4 is as follows:

- **default:** the default options for dProlog are threaded, two opcodes, bind call, truncated integers, local pc = bx, conditional trailing, no trail overflow check, all instruction specialization and compression active
- **no bind call:** usually the call instruction (and execute) needs to refer to a global predicate table to find the entry point of the predicate to be called; this gives more flexibility as far as debugging and reconsulting is concerned; in its default mode of operation however, dProlog will put the address of the entry point directly <sup>9</sup> in the call (and execute) instruction; we named this **bind call**; note that maintaining usual debugging and reconsulting can be obtained even when binding calls is active with little implementation overhead; however as currently implemented, the bind call optimization is not compatible with Prolog level debugging; column one shows the effect of not binding calls: although this optimization seems a reasonable thing to do, its effect varies and is not too big; this is in part caused by the fact that in dProlog the lookup of the entry of a predicate, was kept on purpose very simple: only one indirection is needed
- **always trail:** this column represents unconditional trailing; its effect depends on the kind of benchmark - a benchmark in which each potential trailing is actual, will obviously benefit from deleting the test -, but is never dramatic except for (s)nrev(swap); we also give the maximal trail size for comparison
- **trail overflow test:** if the implementation allocates enough space for the trail (as much as the heap in `heap_vars` and `tag_on_data`, six times <sup>10</sup> as much in `parma_vars`, as much as heap and local stack together in `wam_vars`), trail overflow testing is not necessary <sup>11</sup>; that is the default in dProlog; however, the space penalty might seem too high; so it is worth looking at the potential performance loss in testing for trail overflow; the effect is surprisingly high on some benchmarks
- **full integers:** lots of implementations sacrifice the range of integer numbers for the benefit of their compact representation; this means that in XSB for instance, integers are represented with just 28 bits, as in the basic 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

---

<sup>9</sup>this is done on the first execution of that instruction

<sup>10</sup>see appendix for explanation

<sup>11</sup>this came up during working with Paul Tarau but might be folklore - note that one has to be careful with this schema when destructive update is implemented

relative bad performance can be partly explained by the particular representation used for full integers, relying on a heap offset; the other reason is the suboptimal abstract machine code for full integers

- **glob bx, glob bp, register, no reg:** as in [4] we found it beneficial to assign the program counter of the WAM to a hardware register; [4] does not report that there 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; the default choice is to assign the program counter locally to ebx: `bf loc bx`; the meaning of `bf glob bx` and `bf glob bp` follows easily; we also tried just declaring the program counter locally as a register or giving no directions to the C compiler at all;

As for the choice of which register should be assigned to the WAM program counter, the table indicates that a global bx register is 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, the particular way of writing C code in the emulator (and elsewhere). Rather one should take this as an indication that one must try out which register assignment is good for oneself. 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.

	default	no bind call	always trail	check tr overflow	full integers	glob bx	glob bp	register	no reg
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	448296	448296	448296	674911	448296	448296	448296	448296
trail	57677	57677	403043	57677	57677	57677	57677	57677	57677
ls	190865	190865	190865	190865	238572	190865	190865	190865	190865
code	308152	308152	308152	308152	311044	308152	308152	308152	308152
indexa	10	12	11	11	11	11	11	9	11
indexa2	71	73	73	71	76	74	74	72	71
indexf	25	26	23	25	27	25	25	24	24
list	34	37	34	34	39	34	34	48	47
struct	51	54	52	52	59	49	49	51	50
metacall	39	40	40	44	43	41	41	41	41
plus1i	44	45	46	45	63	44	44	44	44
plus1f	133	130	126	129	137	129	129	127	128
snrev	58	64	72	63	64	61	61	61	62
snrevswap	60	63	72	64	65	61	61	61	63
heap	448296	448296	448296	448296	674911	448296	448296	448296	448296
trail	57677	57677	403043	57677	57677	57677	57677	57677	57677
ls	190865	190865	190865	190865	238572	190865	190865	190865	190865
code	321808	321808	321808	321808	325092	321808	321808	321808	321808

Table 4: The effect of emulator properties in dProlog

## 7.2 Varying the abstract machine code

We report here on variations in the generated abstract machine code: variations are caused by performing less instruction compression or instruction specialization.

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 from it. In table 5 we refer to it with **getlist**.

We have implemented two more instruction compressions and two instruction specializations:

- **dealloc**: compresses the sequence of the instructions *deallocate*, *proceed* into one (new) instruction *dealloc\_proceed*, the sequence *deallocate*, *execute* into *dealex* and *deallocate*, *builtin*, *proceed* into *builtin\_dealloc\_proceed*; as far as the experiment goes, this compression is not particularly effective
- **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 performs this compression and that's why we considered it worth measuring its effect in our implementation;
- **try**: specialized versions for *try*, *retry* and *trust* instructions are generated for predicates with arities 2 and 3; Yap performs this specialization for arities 0 up to 4; the figures indicate that this specialization is a good one
- **switch**: again a specialization found 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, then a specialized instruction is generated which as list exit jumps directly into the read-mode of the instruction **following** the *getlist*; *append/3* obviously benefits from this specialization; XSB generates a *switch\_on\_term* instruction when one alternative has a compound argument and the other alternative has an atomic argument; we have as specialized that for the list case to start with; since in our tagging schema, testing for a list and for compound have the same cost, this does not affect other issues

The specialization *try* and compression *dealloc* are independent of each other and the other transformations, which are performed in the order: *switch*, *getlist*, *uni*. Also, a *switch* specialization prevents the *getlist* compression of the corresponding *getlist*, 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).

The tables show that the effect is more pronounced on particular artificial benchmarks, but the effect on larger benchmarks is smaller. This is hardly a surprise.

Overall, the compression *uni* is the most effective.

Note that even though the compiler didn't generate some instructions, all instructions were present in the emulator at all times.

	all specs compr	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	308152	309896	309908	308020	309268	313420	309268	313156	315652	315652
indexa	10	11	12	10	10	11	10	10	11	10
indexa2	71	70	72	71	71	71	71	71	71	71
indexf	25	25	26	25	25	25	25	25	25	25
list	34	34	34	49	34	40	40	49	40	54
struct	51	49	50	48	51	66	50	66	66	66
metacall	39	41	39	38	39	39	39	39	39	39
plusli	44	44	45	44	44	44	44	44	44	44
plus1f	133	132	132	133	133	132	133	132	132	132
snrev	58	60	60	60	60	75	58	76	76	76
snrevswap	60	60	61	59	60	76	59	77	77	77
code	321808	323664	323820	321664	322924	327148	322924	326860	329380	329380

Table 5: Variations of abstract machine code in dProlog

### 7.3 Changing the sizes in the abstract machine code

Table 6 shows how performance changes when the size of (some objects in) the abstract machine code is changed: **areg2-yvar2** means that for the index in the argument register array, 2 bytes are used in the internal instruction representation and also 2 bytes for the offset of a permanent variable in its environment. Similar for **areg4-yvar4**. **align20** means that the alignment (see 4) of the whole instruction is set to 20: without changing any other sizes this results in a close to doubling of the code size for the threaded version with double opcode. **align36** results in a close to quadruple code size.

	switch1					thread2				
	default	areg2 yvar2	areg4 yvar4	align20	align36	default	areg2 yvar2	areg4 yvar4	align20	align36
boyer	54	57	55	55	55	40	39	39	41	41
browse	81	81	80	80	81	48	49	47	49	49
cal	87	86	89	87	87	77	78	78	78	78
chat	54	55	55	61	64	45	44	46	48	52
crypt	70	71	70	71	71	53	54	53	53	54
ham	99	101	98	99	99	59	60	59	60	61
meta_qsort	76	78	76	77	77	51	52	51	52	53
nrev	130	140	133	131	131	43	43	43	44	44
poly_10	46	48	47	47	47	32	32	32	32	33
queens_16	118	120	124	117	117	88	93	88	89	89
queens	176	179	182	177	175	106	109	105	106	106
reducer	26	26	26	27	28	19	18	19	19	21
sdda	51	52	54	54	59	40	42	41	42	44
send	70	70	73	70	72	49	54	51	50	50
tak	101	102	105	100	101	73	76	75	79	75
zebra	103	106	105	105	104	82	91	85	86	84
code	193004	236504	329584	683500	1230300	308152	309620	353120	683500	1230300
indexa	35	38	37	36	36	10	11	10	10	11
indexa2	133	137	136	135	133	71	80	74	73	73
indexf	60	61	60	60	60	25	26	24	25	27
list	109	111	114	110	108	34	35	34	34	36
struct	158	163	166	160	159	51	57	49	56	52
metacall	97	98	95	99	97	39	41	40	40	40
plusli	104	107	104	105	104	44	48	47	46	45
pluslf	189	193	195	195	192	133	139	135	138	136
snrev	166	169	173	168	165	58	63	62	62	61
snrevswap	164	168	174	166	168	60	66	63	64	62
code	202168	246948	343532	714640	1286352	321808	323276	368056	714640	1286352

Table 6: Variations due to abstract machine code size in dProlog

The columns areg2-yvar2 and areg4-yvar4 indicate that there is little reason to restrict the number of permanent variables to 255 (or close to that) as is common practice in many implementations. Actually, the performance penalty for larger code size is surprisingly small, both for the single opcode switch version and the double opcode threaded version. However, we have also

made some measurements on a smaller machine, and there the penalty is significantly higher. One can expect also that for larger programs than were tested here, the code size increase will also significantly degrade performance.

## 8 Different term representation schemas in dProlog

Table 7 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 7.1. As for the stack sizes, only heap and trail are mentioned, as the choice point stack and trail (and also the code size) is the same for all four versions.

A priori, one can say that `heap_vars` and `parma_vars` 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. For `parma_vars`, this is because when a variable is bound to a non-variable, all the cells in the chain representing the variable are bound and subject to trailing. Note also that in `parma_vars` trailing one cell needs two entries on the trail stack. It is clear that `wam_vars` must (conditionally) trail on globalizing a permanent variable (`unipval` and `putuval`). But even for permanent variables that are never globalized, it can be seen that `wam_vars` potentially trails more than `heap_vars` from the following example:

query :- choice, bind(X), use(X).	choice. choice :- fail. use(_).	bind(666).
--	---------------------------------------	------------

In `wam_vars`, the variable `X` is allocated in the environment of `query/0`. This environment is older than the choice point made by `choice/0`. So, binding `X` to `666` requires trailing. On the other hand in `heap_vars`, the cell that is bound to `666` is on the heap and younger than the choice point, because the initialization of `X` happens after the choice point was created. Therefore, in `heap_vars`, no trailing occurs. This effect can be observed in the benchmarks, but it is not nearly as pronounced as the difference in heap consumption between `heap_vars` and `wam_vars`.

The figures in table 7 are in accordance with the above reasoning. There seems to be a huge difference between the heap usage of `wam_vars` and the others. The reported heap usage happens to be the heap usage for `boyer` (all other benchmarks use less and don't show up in the figures). Closer inspection of `boyer` teaches that 99.6% of the extra heap usage is caused by the particular builtin calling mechanism from XSB: indeed, 99.6% of the difference is due to the double calls to `functor/3` and `arg/3`.

`Wam_vars` comes out as the winner with `heap_vars` a close second. `Tag_on_data` is clearly the big loser but also `parma_vars` suffers badly from its disadvantages on some benchmarks. It might be tempting to generalize from this, but such a generalization would be unscientific. What is definitely true: a straightforward implementation of `parma_vars` or `tag_on_data` will not beat easily the other schemas. So more advanced intermediate code or tagging schemas are necessary to make them competitive or perhaps better. To us, it came as a surprise that the disadvantage of `heap_vars` (more heap usage) is not compensated for by the simplification in some instructions and trailing. Also here more work is needed to achieve this compensation.

	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
indexa	10	9	10	12
indexa2	71	74	72	72
indexf	25	25	26	23
list	34	34	34	41
struct	51	52	64	57
metacall	39	40	41	43
plusli	44	45	45	45
pluslf	133	130	132	128
snrev	58	61	75	74
snrevswap	60	61	75	76
heap	448296	400185	448296	448366
trail	57677	58635	115366	57677

Table 7: Four schemas for term representation

## 9 Related work

There is lot of literature on Prolog implementation and there exist lots of excellent Prolog implementations. In particular [4] is worth mentioning because it gives good advice on implementing an emulator for Prolog. We have followed to a large extent this advice and benefitted a lot from it. We can also confirm the findings of [4], albeit in the context of a different intermediate code generator and a different stack layout. Not reported here in detail, but we also found that storing the address of the argument registers in the instructions instead of the index in the array of argument registers, gives only marginal speedup (and sometimes even a slowdown)<sup>12</sup>. On the other hand, [4] does not stress the importance of the quality of the generated machine code, probably because its compiler is of sufficient quality. So, our work is partly in line with [4], redoing partly its experiment in a different setting. On the other hand, [4] was not aimed at providing empirical data on alternative

<sup>12</sup>dProlog always uses indices for the arguments

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 comparisons on paper is concerned: [11] describes nicely the issues involved in implementing PARMA variables and relates them to 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] hardly discusses the comparison of tag-on-data with the usual WAM representation, but focusses on the issue of term compression and stack usage.

It is entirely possible that other implementors have done preliminary testing similar to what is reported here, before making final implementation choices, but have never published the obtained results.

## 10 Conclusion

The main point of this paper is to make the figures available that we have gathered, about the four term representations that are easily compatible with WAM and about choices in the implementation of the abstract machine compiler and emulator. The previous sections contain some interpretation of the figures, but the main conclusion about the four term representations is that the `tag_on_data` representation seems less attractive than the other three. 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 could make any implementor water mouth. 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. Also `parma_vars` seems less attractive than the other two, but note that for some admittedly artificial examples, `parma_vars` will perform arbitrarily better (see appendix E). As for choosing between `wam_vars` and `heap_vars`, a space-time trade off must be taken into account and this trade off changes with new architectures. The conclusions about the tradeoff and also the other issues are best left to the curious implementor, maybe after having seen the actual implementation (available from [http://www.cs.kuleuven.ac.be/~bmd/dProlog\\_experiment](http://www.cs.kuleuven.ac.be/~bmd/dProlog_experiment), benchmarks included). Since we wanted the obtained figures to have some significance, we have put a lot of effort in making sure that we have not favored any one version, in particular, we have made sure that 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<sup>13</sup> is called is the same, that the indexing (in particular the occurrence of hash collisions) is the same etc.<sup>14</sup> 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 have little defense against such criticism: we are aware that given much more time, we can improve each of the four implementations, but we are not aware of having favored one over the other.

So, what to do with the figures? One use is when other issues are on the line. E.g. it is known that in SLG-WAM as implemented by XSB, the trail must be tidied on cut. However, tidy cut on trail can affect the complexity in a bad way (see appendix E). Also, it turns out that this tidying

---

<sup>13</sup>dProlog implements no occurs check, neither rational trees

<sup>14</sup>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 percent for the benchmark suite), `tag_on_data` has marginally less because of a different optimal sequence of instructions in the `switchonlist` instruction

of the trail in SLG-WAM is only needed for local stack variables. On the basis of the figures in this paper, one could come to the conclusion that it is worthwhile to trade the `wam_vars` schema for the `heap_vars` and not tidy the trail on cut.

The most important thing when contemplating the implementation of particular optimizations or representations, is to keep in mind what some issues will affect many parts of the system, while others will have only an impact in one place. E.g. instruction compression will probably affect locally the compiler and the emulator, while a different term representation affects the implementation more globally.

The figures give also good indications on which specializations, compressions and emulator optimizations are interesting and uncover weaknesses in the XSB compiler. As such, they are currently being used in a redesign of the XSB emulator.

There are definitely points at which our experience might not carry over immediately: we have gathered data only on one processor. For a different machine, the trade offs might be different. Since dProlog is portable to most architectures (and we have ported it to some) the experiments can be redone for each platform one chooses but it will require some extra work as well, mainly because of hardware registers to be assigned to WAM registers in a judicious way.

It is also not clear whether the results carry over to different implementations of logic.

Like in [4] we found that a fast emulator is a combination of the following factors:

1. C code written according to a certain discipline
2. selective use of gcc features
3. a decent basic abstract machine code generator
4. some instruction compression
5. some instruction specialization

We were missing mostly item 3: while [4] claims that for instance the sophistication of register allocation as in [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) builtins must be well thought out for an emulator: the XSB mechanism (setting up the arguments in particular argument registers) leads to more register shuffling and more executed emulator cycles.

There is value in redoing the effort described in [4] albeit in a different context <sup>15</sup>: it has indicated that the advice given in [4] is more generally applicable than within its own context.

Apart from the main results, there is another lesson to be learned from our experiment: the original intention was to make a fast system on the basis of the XSB compiler. In fact, our displeasure with the speed of the XSB system made us search for ways to improve XSB. Also, we wanted to rely on as few ad hoc hacks as possible: we were convinced that a disciplined way of writing an emulator would lead to a reasonably fast emulator, especially since [4] gives such good and easy to follow advice. This turned out to be only partially true: even with the suboptimal code generated by XSB, one can make an emulator that beats SICStus Prolog regularly. But on the basis of the XSB compiler one can't come close to a system like Yap that together with a good discipline for the implementation of its emulator, also uses a well thought out abstract machine compiler.

---

<sup>15</sup>in exactly the same context would probably be meaningless

There is lots of interesting work to do in the context of dProlog and related to speeding it up; we just name a few things. At the abstract machine code level, we think two more points are worth doing better: specialization of builtins (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 it.

In general, there are still many other things to study within a fixed implementation context: there is for instance no hard data on entirely different tagging schemas, e.g. like the one SICStus Prolog is using. Also different allocation schemas within a WAM-like implementation should be experimented with, together with different garbage collection strategies and principles. The problem with the latter is however that an entirely new type of benchmarks is needed.

We will continue using dProlog for experimenting and gathering information.

Finally, our work will continue with incorporating into the compiler and emulator information from declarations (as in [10] for instance) and studying the impact of that in the context of an emulator.

## Acknowledgements

We want to thank the XSB team for making their compiler and system available to us, and for explanations on its workings. In particular we want to thank Kostis Sagonas and David S. Warren. 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. Ait-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. The MIT Press, Cambridge, Massachusetts, 1991. See also: <http://www.isg.sfu.ca/~hak/documents/wam.html>.
- [2] 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. See also: <http://www.sics.se/isl/sicstus.html>
- [4] V. S. Costa, *Optimising Bytecode Emulation for Prolog*. Proceedings of PPDP'99, LNCS 1702, Springer-Verlag, 261-277, September, 1999. See also <http://www.ncc.up.pt/~vsc/Yap/>.
- [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 International Workshop on Practical Aspects of Declarative Languages (PADL'00), Boston, Jan. 2000, pp. 93–108

- [8] G. Janssens, B. Demoen, A. Mariën. *Improving the register allocation in WAM by reordering unification*. International 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 North American Conference on Logic Programming, Cleveland, Ohio, oct 1989, p. 1030-1047
- [10] Phuong-Lan Nguyen. *Optimisation du Code produit par un Compilateur Prolog*. Rapport de DEA, ENSIMAG, Laboratoire de Génie Informatique, Grenoble, 1988 In French.
- [11] T. Lindgren, P. Mildner, and J. Bevemyr. *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, sep 1994, Springer, Lecture Notes in Computer Science 844, pp. 73-87, Eds. M. Hermenegildo and J. Penjam See also <http://www.binnetcorp.com/BinProlog/index.html>.
- [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] XSB: See <http://www.cs.sunysb.edu/~sbprolog/>.

## A Why the name dProlog ?

When looking for a name, we started alphabetically: aProlog is clearly a bad name; bProlog and cProlog were out because there exist systems like that already; so dProlog was the choice :-)

## B Difference in number of times of dereferencing

Compared to heap\_vars:

- ls\_vars: more in putuval, bldpval, bldtval, unitval ... each instruction that *must* test whether something must be globalized
- parma\_vars: more in bldpval, bldtval, unitval, call\_unkn, metacall ... each instruction that potentially puts an undef inside a structure
- tag\_on\_data: less in switchonlist(\_skip): because of better instruction sequence

## C Maximal trail size in PARMA related to heap size

Let  $N == \text{heap\_size}$  and assume that the heap contains  $N$  free variables whose chain has length one (that's clearly the maximal number of free variables the heap can contain). Each binding of two different chains (even the trivial chain of length one) representing free variables has as effect that there will be one chain less and that at most two locations are value trailed. Now you end up with one free variable on the heap, and this has cost at most  $2 * N$  trailings, so  $4 * N$  trail entries. Now bind the variable to an atomic value; again  $N$  trailings, so  $2N$  trail entries. So the number of value-address entries on the trail will be at most  $3 * N$  and that means that the trail needs to be six times as large as the heap.

## D PARMA same variable test

Below is the code for the test *test\_same\_var* (from the dProlog parma\_vars variant) given two dereferenced terms p and q. It is meant to be called as *test\_same\_var(p, q, goto label1, goto label2)*. The idea is to move along both cycles at the same time and continuously test whether one is back at the starting point of the same **or** of the other cycle.

```
#define test_same_var(p,q,sameaction,diffaction)
{
    dlong *sp, *sq;
    sp = p; sq = q;
    while (1)
    {
        if (q == sp) sameaction;
        if (p == sq) sameaction;
        p = (dlong *)*p;
        q = (dlong *)*q;
        if (p == sp) diffaction;
        if (q == sq) diffaction;
    }
}
```

```

    }
}

```

The number of times the loop body is executed always equals at most the size of the smallest of the cycles starting at  $p$  or  $q$ . This cannot be mimicked in WAM which in the worst case must traverse both reference chains to detect inequality of two free variables, even if one takes into account the order (from newer to older variable) in which WAM makes bindings. [11] does not note this difference probably because their *test\_same\_var* for PARMA variables is asymmetric.

The above code was basically developed in the context of HAL [5]: the original code was asymmetric and accounted for some strange performance figures in some benchmarks. Symmetric treatment of  $p$  and  $q$  lead to the above.

## E Tidying the trail can harm you

Beginning Jan 2000, Henning Makhholm wrote in comp.lang.prolog

```

> If that is correct, it ought to imply that a program such as
...
> takes  $O(N^2)$  time to run because each of the  $N$  cuts needs
> to consider a trail of length  $N$ .
>
> My problem is that the program seems to run in linear time
> on the SICStus 3.8 system we use here.

```

The example by Henning Makhholm was wrong.

This and other trailing issues have kept us busy for some time while working together with K. Sagonas, so we had a real example ready. Bart Demoen posted it to comp.lang.prolog:

```

q(N) :-
    mkfreelist(N,L),
    cputime(A),
    (mmc(N,L), fail ; true),
    cputime(B),
    X is B - A,
    write(X), nl, fail.

mmc(N,L) :-
    N > 0,
    M is N - 1,
    mmc(M,L),
    !.
mmc(N,L) :-
    mkground(L).

mkfreelist(N,L) :-
    (N = 0 ->
        L = []
    ;
        NN is N - 1,
        L = [_|R],
        mkfreelist(NN,R)
    ).

mkground([]).
mkground([a|R]) :-
    mkground(R).

```

```
[for SICStus add: cputime(X) :- statistics(runtime,[X|_]).]
```

The goal  $q(N)$  is now quadratic in  $N$  in an implementation doing trail tidying on cut (as XSB), and linear if not.

## PARMA quadratic and WAM linear

The following program - with query `?- t(N)` - is quadratic in  $N$  for SICStus Prolog and linear in `parma_vars`. The basic idea is: make a list of variables, bind them all together, then bind the first one to the integer 1, then test whether all elements of the list are integer. For the latter, accessing them is enough, the `integer/1` test is merely an example.

```
mkvarlist(N,In,L) :-
    (N = 0 ->
        L = In
    ;
        NN is N - 1,
        mkvarlist(NN,[_|In],L)
    ).

aliasall([_]) :- !.
aliasall([X|R]) :-
    R = [Y|_],
    X = Y,
    aliasall(R).

test([]).
test([X|R]) :-
    integer(X),
    test(R).
```

```
t(N) :-
    statistics(runtime,[T0|_]),
    mkvarlist(N,[],L),
    aliasall(L),
    L = [1|_],
    statistics(runtime,[T1|_]),
    test(L),
    statistics(runtime,[T2|_]),
    A0 is T1 - T0,
    A1 is T2 - T1,
    write(A0), nl,
    write(A1), nl, fail.
```

Does the opposite also exist: programs for which PARMA is quadratic and WAM linear? An open question for me ...

## Lots of numbers

The next tables give all the numbers related to each of the four term representation, the 5 different basic modes and for the different assignments or declarations of the WAM P register to different hardware registers - that makes up 120 engines. All the timings are given in 1/100 seconds. The sizes are in 4-byte words. We refrain from further interpretation of these numbers.

	locpc = bx						locpc = register						plain locpc					
	sw1	sw2	ju1	ju2	th1	th2	sw1	sw2	ju1	ju2	th1	th2	sw1	sw2	ju1	ju2	th1	th2
boyer	52	66	42	42	38	38	54	65	43	42	40	40	54	65	43	42	40	39
browse	80	95	50	49	46	46	81	95	53	50	50	49	81	95	52	50	50	49
cal	85	104	83	82	75	75	87	103	85	82	75	75	85	103	84	81	75	75
chat	55	65	47	47	45	44	56	65	48	47	46	44	56	65	49	47	46	44
crypt	71	83	57	56	52	52	72	84	57	57	54	53	72	84	57	56	54	53
ham	98	115	64	63	59	58	99	117	69	66	63	62	99	117	68	65	63	62
meta_qsort	75	89	54	54	49	48	75	89	55	54	51	50	76	89	55	54	51	51
nrev	134	160	47	46	43	44	137	162	55	51	54	51	137	162	54	50	55	53
poly_10	47	56	35	34	33	31	48	55	36	34	33	32	48	55	36	34	33	33
queens_16	114	139	93	92	85	85	117	139	95	94	89	87	117	139	95	94	88	87
queens	176	213	111	110	103	103	181	214	116	114	111	106	181	214	115	114	111	107
reducer	26	30	20	19	19	18	26	30	21	20	19	19	26	30	21	20	19	18
sdda	53	64	44	44	42	42	53	64	44	45	43	42	53	64	44	45	43	42
send	64	79	54	52	44	45	65	80	54	54	49	49	65	80	54	54	49	49
tak	99	123	75	75	70	71	100	122	76	75	72	71	100	122	76	75	71	71
zebra	103	112	84	84	82	82	104	113	86	86	85	83	104	113	86	86	85	83
indexa	37	48	12	10	10	9	32	45	11	12	10	10	32	45	11	12	10	11
indexa2	136	170	76	80	75	74	136	167	79	83	74	74	136	167	78	81	75	76
indexf	61	73	26	27	25	25	61	73	26	26	27	25	61	74	26	26	25	23
list	108	136	34	34	35	34	120	141	50	45	50	49	120	143	50	45	50	49
struct	160	201	57	56	53	52	160	194	58	57	52	50	160	194	57	57	52	50
metacall	96	118	46	46	40	40	101	118	48	48	40	42	103	119	48	46	40	41
plusli	103	130	46	48	45	45	110	125	47	46	45	46	110	125	48	47	45	46
pluslf	194	236	141	142	133	130	194	232	140	138	134	131	194	232	142	137	134	131
snrev	162	202	66	64	65	61	166	194	65	64	59	61	166	194	67	65	59	61
snrevswap	162	201	66	64	64	61	173	193	67	66	65	62	174	193	65	64	61	62

Table 8: heap\_vars: different register allocations/declarations

	globpc = bx						globpc = bp					
	sw1	sw2	ju1	ju2	th1	th2	sw1	sw2	ju1	ju2	th1	th2
boyer	53	65	43	43	39	39	54	66	43	42	41	40
browse	80	95	50	50	46	46	81	98	54	52	50	50
cal	86	103	85	85	76	75	86	104	85	85	76	76
chat	55	65	48	48	46	44	57	67	49	48	47	45
crypt	70	84	57	56	53	52	71	86	59	58	55	54
ham	97	116	65	62	59	58	101	120	70	68	64	62
meta_qsort	74	90	54	55	49	50	74	91	56	55	51	51
nrev	130	159	46	44	46	45	137	166	56	51	53	53
poly_10	47	55	35	34	32	32	48	56	36	34	33	33
queens_16	113	136	94	94	85	85	114	140	98	98	93	89
queens	174	212	111	109	104	102	177	218	118	115	110	109
reducer	25	30	20	20	19	18	26	31	20	20	19	19
sdda	53	65	44	45	42	42	54	65	45	45	43	43
send	65	79	54	53	44	44	64	81	51	52	51	52
tak	101	121	86	76	72	71	99	123	75	73	74	73
zebra	101	111	82	81	81	80	109	118	91	90	88	88
indexa	34	46	13	12	10	11	35	45	10	11	10	6
indexa2	135	167	83	79	73	73	139	171	78	76	75	72
indexf	61	74	27	27	27	25	62	76	25	26	25	23
list	106	132	35	34	35	34	118	144	49	49	52	48
struct	156	195	55	57	54	50	160	198	59	54	54	51
metacall	97	118	47	45	40	40	98	123	50	49	44	43
plus1i	100	127	46	47	45	44	103	125	47	46	44	44
plus1f	215	231	143	143	126	129	195	232	140	139	131	132
snrev	164	200	66	65	63	62	163	202	67	64	64	60
snrevswap	163	199	66	65	63	62	164	200	66	64	63	62

Table 9: heap\_vars: different register allocations/declarations

	locpc = bx						locpc = register						plain locpc					
	sw1	sw2	ju1	ju2	th1	th2	sw1	sw2	ju1	ju2	th1	th2	sw1	sw2	ju1	ju2	th1	th2
boyer	54	67	44	44	40	40	54	68	44	44	42	41	54	68	44	44	41	41
browse	81	95	51	50	48	48	82	97	55	54	51	50	82	97	55	54	51	49
cal	87	105	87	87	78	77	86	105	88	84	79	77	85	106	87	85	80	78
chat	54	64	48	47	46	45	54	64	48	46	45	44	54	64	48	46	45	44
crypt	70	84	57	56	52	53	71	86	56	56	53	52	71	86	56	56	52	51
ham	99	115	66	65	60	59	101	116	70	67	64	64	101	117	70	67	64	65
meta_qsort	76	89	56	55	52	51	75	91	56	55	52	51	74	91	57	54	52	51
nrev	130	157	45	43	46	43	132	162	52	51	53	52	132	162	53	50	54	51
poly_10	46	56	36	34	33	32	47	56	35	34	33	31	47	56	35	35	32	31
queens_16	118	141	100	98	89	88	117	143	100	98	92	90	117	143	100	101	92	90
queens	176	211	113	110	107	106	180	215	119	116	111	107	180	215	118	117	109	107
reducer	26	30	20	20	19	19	26	31	20	20	19	19	25	31	21	20	19	19
sdda	51	62	41	42	41	40	52	63	42	42	41	41	51	63	42	42	41	40
send	70	85	59	60	49	49	69	85	61	60	54	54	69	85	59	62	54	54
tak	101	125	78	78	73	73	103	128	82	79	74	74	102	128	82	82	74	74
zebra	103	113	85	83	83	82	105	115	87	87	85	85	105	115	88	87	85	85
indexa	35	46	11	11	11	10	38	46	10	12	10	9	38	46	11	12	11	11
indexa2	133	165	79	79	75	71	137	168	80	77	70	72	137	169	78	78	71	71
indexf	60	73	28	25	27	25	61	77	24	25	25	24	60	77	24	26	25	24
list	109	138	35	34	35	34	117	143	48	48	50	48	117	142	50	46	50	47
struct	158	199	60	55	52	51	160	195	58	56	55	51	159	195	56	56	54	50
metacall	97	117	45	46	41	39	97	118	47	47	41	41	98	118	47	46	42	41
plusli	104	128	48	48	45	44	105	126	47	47	44	44	105	126	47	47	44	44
pluslf	189	233	138	136	129	133	192	235	138	135	132	127	192	235	137	137	132	128
snrev	166	197	65	64	63	58	164	197	65	64	63	61	164	197	66	65	63	62
snrevswap	164	197	66	64	65	60	163	197	66	64	62	61	163	197	65	65	63	63

Table 10: wam\_vars: different register allocations/declarations

	globpc = bx						globpc = bp					
	sw1	sw2	ju1	ju2	th1	th2	sw1	sw2	ju1	ju2	th1	th2
boyer	55	67	42	42	40	40	54	67	44	44	42	42
browse	79	95	50	50	49	47	82	99	55	54	52	51
cal	85	104	87	87	78	78	84	105	86	86	77	78
chat	56	65	48	47	45	45	55	66	49	48	46	46
crypt	70	85	56	56	53	52	70	83	57	56	52	51
ham	99	118	64	61	60	59	100	120	71	68	66	64
meta_qsort	75	90	55	55	52	51	76	91	57	56	52	51
nrev	129	158	46	44	44	43	135	161	58	52	52	50
poly_10	47	56	34	33	32	31	47	56	35	34	32	32
queens_16	116	141	97	98	88	88	117	143	102	102	93	91
queens	175	212	111	108	103	103	180	217	120	117	111	110
reducer	25	30	20	20	19	18	26	31	21	21	19	19
sdda	51	61	43	42	40	40	52	64	43	43	42	41
send	70	86	52	52	50	49	69	87	60	58	55	55
tak	101	125	77	77	73	75	101	125	79	80	76	78
zebra	103	112	84	82	82	81	107	118	90	89	88	88
indexa	36	48	11	11	12	11	38	47	13	13	10	10
indexa2	136	166	79	78	71	74	138	169	78	80	72	73
indexf	61	77	28	25	25	25	61	77	26	25	26	24
list	108	135	35	34	34	34	118	142	48	48	52	48
struct	159	198	60	54	54	49	160	194	57	53	54	50
metacall	95	119	44	43	40	41	99	122	48	49	43	41
plus1i	104	126	48	47	45	44	99	123	47	41	37	41
plus1f	189	230	135	138	125	129	201	240	139	139	134	132
snrev	165	197	65	64	64	61	165	196	66	64	61	62
snrevswap	164	197	65	64	65	61	164	195	65	63	62	62

Table 11: wam\_vars: different register allocations/declarations

	locpc = bx						locpc = register						plain locpc					
	sw1	sw2	ju1	ju2	th1	th2	sw1	sw2	ju1	ju2	th1	th2	sw1	sw2	ju1	ju2	th1	th2
boyer	53	67	44	44	40	40	54	67	46	45	41	40	54	67	45	45	41	41
browse	77	96	52	51	48	46	78	96	52	52	48	47	78	96	52	52	48	47
cal	85	105	85	84	77	77	86	105	84	85	78	76	86	105	86	85	78	77
chat	56	67	50	49	47	46	55	66	51	49	48	46	55	66	51	49	48	46
crypt	71	83	56	56	53	52	71	83	57	56	54	53	71	83	57	56	54	53
ham	95	118	70	64	62	60	97	116	69	66	63	61	98	117	69	67	63	61
meta_qsort	77	94	59	58	55	54	78	94	59	59	54	54	78	94	59	59	54	54
nrev	131	163	51	47	48	47	144	169	59	57	57	55	143	169	59	57	57	55
poly_10	47	56	36	35	33	34	47	56	36	35	34	33	46	56	36	35	34	33
queens_16	117	141	97	98	91	90	117	139	100	98	90	92	117	140	100	98	90	92
queens	175	216	116	116	114	113	181	216	121	119	116	114	178	216	121	119	115	114
reducer	28	33	23	22	21	21	28	33	23	22	21	21	28	33	23	22	21	21
sdda	54	65	46	45	43	43	54	66	46	45	42	42	54	65	46	45	42	42
send	66	83	57	57	52	52	69	84	59	59	53	53	69	84	59	59	53	53
tak	103	128	81	81	75	75	103	128	82	83	76	75	103	128	82	83	76	75
zebra	119	128	100	100	98	97	118	126	98	97	97	97	118	126	98	98	98	97
indexa	33	45	12	12	10	10	35	45	13	11	11	11	35	45	13	11	11	11
indexa2	130	168	84	82	71	72	136	169	82	79	79	75	136	168	82	80	80	75
indexf	59	75	28	29	28	26	62	76	30	29	29	26	62	76	31	30	29	28
list	103	131	35	34	35	34	120	140	48	43	50	49	121	140	48	42	51	49
struct	159	206	70	65	65	64	164	204	68	70	67	63	163	204	68	66	83	63
metacall	91	115	46	42	40	41	93	117	45	46	39	40	91	117	45	46	41	40
plusi	105	128	49	47	44	45	106	126	47	46	44	43	106	126	47	46	44	43
pluslf	191	234	144	138	135	132	195	230	138	141	138	135	195	230	138	141	138	134
snrev	171	212	77	74	73	75	175	211	76	74	73	75	175	210	76	73	75	75
snrevswap	171	211	76	75	73	75	174	210	75	75	73	74	175	210	77	74	75	74

Table 12: parma\_vars: different register allocations/declarations

	globpc = bx						globpc = bp					
	sw1	sw2	ju1	ju2	th1	th2	sw1	sw2	ju1	ju2	th1	th2
boyer	55	67	44	44	40	39	55	68	46	46	41	41
browse	78	94	52	52	48	46	80	97	54	53	50	49
cal	84	105	84	85	78	77	85	106	87	85	77	76
chat	55	66	50	48	47	46	57	68	52	50	49	48
crypt	70	84	56	56	53	53	72	83	58	57	53	53
ham	98	117	66	64	62	60	100	119	69	68	64	61
meta_qsort	77	93	59	59	55	54	80	95	61	60	56	56
nrev	134	165	52	49	48	46	145	169	61	60	52	56
poly_10	48	57	36	34	33	33	47	57	36	36	34	33
queens_16	114	141	97	98	91	91	117	141	99	98	90	90
queens	176	217	115	115	114	112	183	220	123	121	117	115
reducer	28	33	23	22	21	21	29	33	23	23	22	22
sdda	54	66	45	45	43	42	55	67	47	47	44	43
send	67	83	57	57	53	53	69	84	59	60	49	48
tak	106	132	80	82	75	75	104	129	85	84	76	76
zebra	119	128	98	98	97	97	125	135	106	105	105	105
indexa	36	46	11	11	12	10	36	47	12	12	11	10
indexa2	139	170	78	82	77	74	136	170	82	81	76	71
indexf	60	77	30	28	28	26	59	77	28	28	29	27
list	109	133	35	34	35	34	121	140	47	45	51	50
struct	164	207	67	67	67	63	162	203	68	66	66	63
metacall	93	118	45	45	40	39	96	120	48	48	40	41
plus1i	105	126	50	52	44	45	104	126	48	50	46	44
plus1f	197	233	142	143	135	135	196	236	142	142	136	137
snrev	175	216	76	77	73	74	174	212	75	76	73	76
snrevswap	174	215	77	76	75	73	173	211	73	75	75	77

Table 13: parma\_vars: different register allocations/declarations

	locpc = bx						locpc = register						plain locpc					
	sw1	sw2	ju1	ju2	th1	th2	sw1	sw2	ju1	ju2	th1	th2	sw1	sw2	ju1	ju2	th1	th2
boyer	56	69	46	45	42	42	57	68	46	46	44	44	57	69	46	46	44	44
browse	87	104	59	56	53	53	86	103	60	58	55	54	85	103	59	58	54	54
cal	86	106	86	86	76	77	87	107	84	87	79	79	85	105	85	87	79	79
chat	57	66	49	48	48	47	55	67	51	49	48	47	55	67	51	49	49	47
crypt	72	85	59	59	55	54	73	87	60	58	57	56	73	87	60	58	57	56
ham	101	121	73	71	65	64	100	120	73	69	68	66	100	121	73	69	68	66
meta_qsort	78	92	59	57	54	53	78	93	59	57	54	54	77	93	58	57	54	53
nrev	141	168	58	56	57	56	132	171	58	61	62	62	133	171	58	58	63	63
poly_10	49	59	37	36	35	34	52	59	38	37	36	36	51	59	38	37	36	36
queens_16	117	145	100	99	91	90	116	143	100	98	94	94	116	143	100	100	95	94
queens	185	222	126	124	116	113	187	224	129	124	123	119	186	224	128	123	121	120
reducer	28	33	23	22	21	21	28	33	22	22	21	21	28	33	23	22	21	21
sdda	54	67	46	45	44	44	56	67	46	45	44	44	56	67	46	45	45	44
send	67	85	59	59	49	49	69	85	60	58	57	56	69	85	60	58	56	56
tak	100	126	79	79	72	73	105	126	80	80	76	75	106	126	80	79	78	75
zebra	111	120	94	93	90	90	112	121	93	93	91	92	112	121	93	93	91	92
indexa	37	47	11	10	11	12	35	47	13	12	10	11	36	47	11	12	10	10
indexa2	136	162	73	73	72	72	135	171	85	80	76	75	135	170	85	80	74	74
indexf	60	76	24	23	23	23	61	77	25	23	23	23	61	76	24	25	24	24
list	115	148	45	41	42	41	110	143	45	42	49	47	110	143	45	43	49	47
struct	161	203	65	62	60	57	164	204	67	62	69	68	164	204	67	62	69	68
metacall	99	122	50	49	45	43	98	122	51	49	46	46	98	121	51	50	47	45
plusli	105	129	47	47	45	45	105	126	46	45	44	43	104	126	46	45	43	44
pluslf	188	233	135	135	129	128	190	232	135	136	134	134	190	232	136	136	134	132
snrev	170	205	79	74	74	74	169	212	77	76	78	78	171	211	78	76	76	77
snrevswap	175	204	78	76	74	76	169	211	78	77	77	77	169	211	77	76	75	79

Table 14: tag\_on\_data: different register allocations/declarations

	globpc = bx						globpc = bp					
	sw1	sw2	ju1	ju2	th1	th2	sw1	sw2	ju1	ju2	th1	th2
boyer	55	68	46	45	42	41	56	70	46	46	44	44
browse	85	104	57	58	54	53	86	105	60	58	56	55
cal	86	105	84	85	77	77	85	105	86	85	78	77
chat	56	66	50	47	47	46	57	68	51	50	48	47
crypt	73	86	59	58	56	55	73	87	60	59	56	55
ham	101	120	71	70	66	65	101	123	73	71	69	66
meta_qsort	78	92	58	57	54	52	78	93	59	59	56	55
nrev	135	170	58	52	56	55	134	172	59	56	62	62
poly_10	49	57	38	38	35	34	50	58	38	37	36	35
queens_16	117	143	99	98	92	92	118	144	100	99	93	100
queens	185	223	125	123	116	114	184	225	130	126	121	121
reducer	28	33	23	22	21	21	28	33	23	22	22	21
sdda	55	65	47	45	43	43	56	68	47	46	45	45
send	67	85	59	59	54	54	69	85	60	61	57	56
tak	100	124	78	79	71	73	101	126	80	82	76	81
zebra	112	119	93	93	91	90	116	127	98	97	97	95
indexa	36	45	11	11	11	11	34	46	11	11	10	10
indexa2	133	163	76	76	71	71	135	168	81	81	73	74
indexf	61	77	23	25	23	22	60	75	23	24	23	23
list	115	143	45	42	42	40	112	148	44	43	49	48
struct	166	206	65	62	61	59	156	203	65	64	69	68
metacall	96	119	50	48	45	44	101	124	53	50	47	47
plus1i	106	126	54	54	45	45	100	126	44	43	37	40
plus1f	196	233	134	134	132	132	198	244	141	143	137	139
snrev	166	210	78	76	72	72	163	201	79	77	75	76
snrevswap	165	210	79	76	74	72	163	200	78	76	76	76

Table 15: tag\_on\_data: different register allocations/declarations