Dynamic Statistics of Sequential Prolog

Ignacio Celis and Jonathan W. Mills
Computer Science Dept.
Indiana University
Lindley Hall 215
Bloomington, IN 47405
e-mail: {celisi, jwmills}@cs.indiana.edu
phone (812)-857-7038

Keywords: Implementations and Architectures

Abstract

This paper presents dynamic statistics taken by simulating sequential Prolog programs. We used the Warren's Abstract Machine (WAM) as the abstract machine to simulate these programs. We show the dynamic data for the following areas: Dereferencing Chain, Unification, Trailing, Untrailing, Choice Point Creation, Environment Creation, Backtracking, etc. This information can be useful for compiler writers and special purpose hardware designers to concentrate their effort toward those features of Prolog that are more frequently used.

Introduction

In this paper we present the run time data of sequential Prolog. This information can be useful for compiler writers and special purpose hardware designers. We used Warren's Abstract Machine (WAM) described in the paper "An Abstract Prolog Instruction Set" [15]. Almost all implementations of Prolog are based on this machine. We assume familiarity with Prolog and the WAm, for a description of the WAM the reader is referred to [1,5,15].

First, we gathered Prolog programs that would execute large amounts of instructions to get a more realistic information (almost all of them are known Prolog benchmarks). Then, we used the PLM [14] Prolog compiler, as defined in [3] with some modifications done by Touati [13] and some others done by us, to generate WAM code. This WAM code was taken as input to our WAM simulator, which yielded the statistics presented in this paper. Figure 1 shows this process.

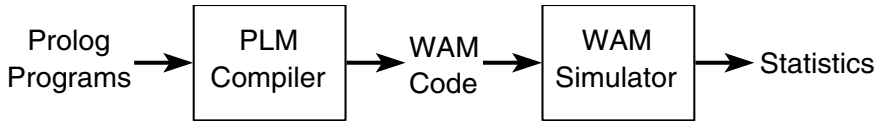


Figure 1

In the first section of this paper, we describe the differences between the WAM and the PLM instruction set. Next, we give a brief description of the Prolog programs we simulated to gather the statistics. Then, we present the statistics for the instructions referencing, memory referencing, dereferencing, calls, choice points creation, environments, unification, trailing, backtracking, unsafe variables and escapes. Finally we include some conclusions and further work.

Difference Between the WAM and the PLM

The PLM instruction set has extra instructions in addition to those of the WAM; including instructions to support cuts (which are not specified in Warren's paper), instructions to support lists using a cdr-coding representation (which we did not use), a different *switch_on_term* instruction, some modifications for garbage collection and some additional instructions for list processing.

Since in the WAM specification there is no support for the Prolog instruction ! (cut), we took the same approach as the one designed in the PLM machine. We added the new instruction *cut* to the WAM instruction set and a bit flag to the processor status to reveal if the current procedure has a choice point. This flag is set by the WAM instructions: *try*, *try_me_else*, *retry*, *retry_me_else* and reset by the instructions: *call*, *proceed*, *execute*, *trust* and *trust_me_else*. Every time an environment is allocated, the flag and the current choice point are saved to allow the pruning of the choice point.When a *cut* instruction is executed, the choice point pointer and the cut flag are read from the environment, if the cut flag is set, it means that the current procedure created a choice point that has to be discarded. This is done by updating the current choice pointer register (B) with the value of the previous choice point.

The disadvantages of this strategy are that, the current choice point pointer has to be saved in the environment even if there is no cut in the procedure and, that environments are created with the only purpose of saving the choice point pointer, so that, the *cut* instruction can read it from the environment. Figure 2 shows some examples of the second disadvantage. To solve this problem, we added a new instruction (*Cutb*). This instruction will discard the current choice point whenever the cut flag is set without needing to allocate an environment.

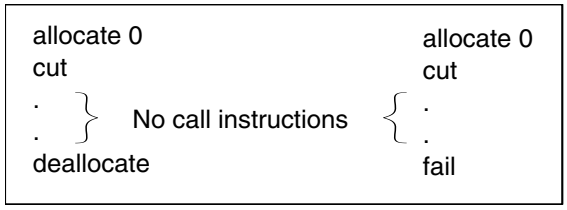


Figure 2

The PLM implementation of the *switch_on_term* instruction is different than that of the WAM. Instead of having the 4 address for jumping, it only has three, and it will branch depending on the tag of register A1 or it will continue with the next instruction if it is a reference.

For Garbage Collection purposes, the environment will save its size. Likewise the choice point will save its size (the number of Argument register used) and the current size of the current environment.

The garbage collection requires that all variables in the environment be initialized before the first *call* instruction (this is when the garbage collector may be trigger). The instruction we used for this, as in the PLM, is the *init* instruction and it is used only for the environment words that have not been initialized before the call. Figure 3 shows an example of this. The *init* instruction creates an unbound variable in the environment.

```
allocate 2                    allocate 2
get_variable Y1,X1            get_variable Y1,X1
call foo                      init Y2
put_value Y1,X2               call foo
put_variable Y2,X3            put_value Y1,X2
                              put_variable Y2,X3
```

Figure 3.

When an element of a list is going to be unified (except for the first element), two WAM instructions are executed (*unify_variable_x* and *get_list*). Figure 4.a shows the sequence to unify the Prolog list [a,b,c,d]. Touati defined a new instruction for handling this situation more efficiently, the new instruction *get_cdr_list* will do the same as the sequence of instructions *unify_variable_x* and *get_list* without all the inefficiencies. Figure 4.b, shows the new code for the unification of the list [a,b,c,d] using the new instruction.

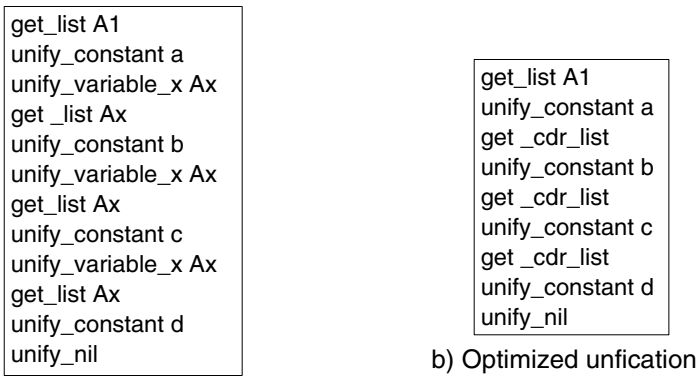


Figure 4

The PLM compiler assumes that the architecture has 8 argument registers and no scratch registers. Hence, for each procedure calls that have more than 7 arguments the PLM compiler will create a dummy structure to save the value of all arguments higher than the 6th argument. Later we will show how frequent this happens.

The unify instructions that follow a *put_structure* or *put_list* will execute in write mode. The compiler is clever enough to distinguish this situation and it will append the suffix *_write_* to the unify instructions if this situation occurs.

Our simulation uses two stacks growing in opposite directions, instead of a single stack and does not handle self modifying code (assert/retract). The garbage collection is disabled; however the bookkeeping to allow garbage collection is done in these simulations. We implemented only those escape routines that were needed.

Benchmarks

We gather 34 Prolog programs as our benchmarks. These programs, when compiled, 38,871 WAM instructions were generated and 30,589,208 WAM instructions were executed. Ten of the programs are known as the Berkeley benchmarks. Table 1 shows the list of programs with a small description for each of them. Table 2 presents for each program the number of goals, predicates, clauses, WAM instructions generated, the number of WAM instructions executed and the percentage of this with respect to all the programs. The program with the largest number of instructions is the chat_parser program and the smallest one is the tak program. In the other hand, the program with the largest number of executed instructions is the browse program (almost 19%) and the smallest one is the differen program.

Most of the programs are known as toy benchmarks. The input of nrev450 and qs2000 were made bigger than the original benchmark programs. The programs prover, sdda, circuit, differen, mumath, query, serialize were executed 10 times by a loop. Some of the benchmarks are explained in Parallel Logic Programming [11].

We now present the statistics gathered by simulating the Prolog Programs. The most relevant points of these statistics are: Most the memory references are reads and writes to the choice

point (45%). Dereferencing chains are short. Ninety percent of all choice points have less than four arguments. Most of the unifications are done with a reference and a high percentage of the backtrackings are shallow backtracks.

Program	Description
boyer	An extract from a Boyer-Moore theorem prover.
browse	Build and query a database, intended to perform many operations that a simple expert system might perform.
chat_parser	Parse a set of English sentences.
circuit	Design of a 2 to 1 multiplexer.
crypt	Cryptomultiplication problem.
cube	Find the first 10 solutions for 7-Cube puzzle. The goal is to stack the 7 colored cubes so that no color appears twice within a given side of the stack.
dcg1	A Definit Clause Grammar program.
differen	Symbolic differentiation of times10, divide10, log10, ops8.
dij	Find the bestpath in a graph using Dijkstra's algorithm
hanoi	Tours of Hanoi problem
ili	Intuitionistic Logic Interpreter.
iso	Generate all isomorphic tree for a given tree.
knigth	Puzzle to find all paths of knight-moves on a chessboard of size 4.
meta_qsort	Meta-interpreter running qsort.
mm	Find the first 100 solutions for breaking the code for a MasterMind game.
mumath	Prove Hofstadter's "mu-math" theorem
nand	Logic synthesis program based on heuristic search.
nrev450	Naive reverse of a list of 450 elements.
poly10	Symbolically raise a polynomial (1+x+y+z) to the 10th power.
pri2	Finds the prime number below 1000 using the Sieve of Eritosthenes algorithm
prove	Simple theorem prover.
puzzle	Given a 5x5x5 cube and some pieces of various sizes, find a packing of those pieces into the cube.
qs2000	Quicksort of list of 2000 numbers
queens_8	Solve the eight queens puzzle.
query	Population query on a static database.
reducer	Graph reducer based on combinators.
sdda	Dataflow analyzer that represents aliasing.
serialize	Calculate serial numbers of a list.
shoppers	Solve the shoppers problem.
tak	A variant of Takeuchi Function.
tri	Calculate all winning sequences of moves for the triangle game.
turtles	Solve the game of rearranging 9 cards so that border of 3X3 board shows the turtle figures of the same color.
waltz	Implements Waltz's algorithm for interpreting line drawings in three-dimensional solids.
zebra	Logical puzzle based on constraints.

Table 1. Set of Prolog programs used.

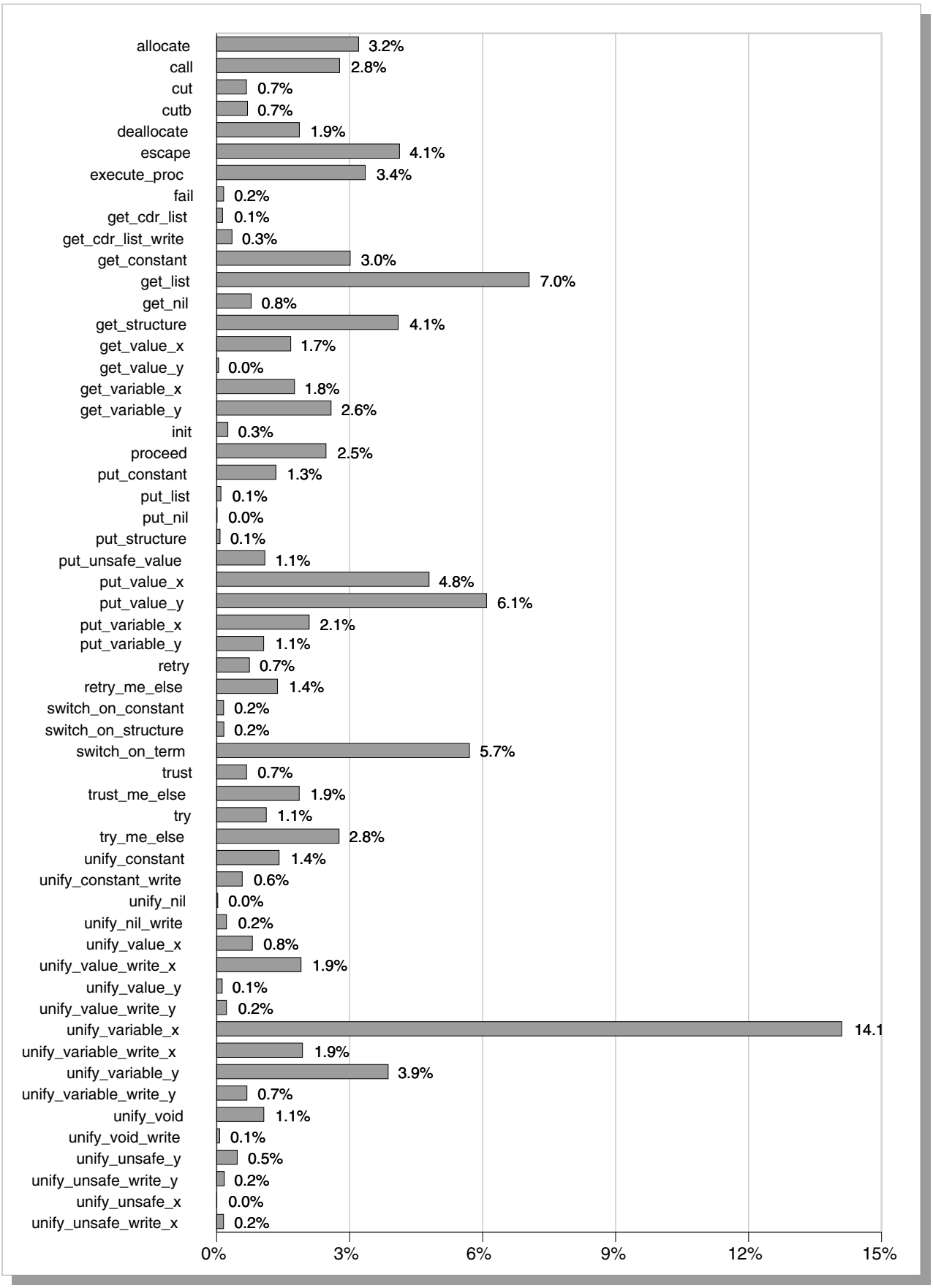
	Goals	Predicates	Clauses	Static	Executed	%
boyer	166	91	135	1,940	4,488,120	14.67
browse	77	17	29	582	6,416,096	20.98
chat_parser	801	376	515	5,851	958,487	3.13
circuit	36	16	20	299	84,748	0.28
crypt	93	22	27	396	29,589	0.10
cube	43	15	23	577	3,141,006	10.27
dcg1	55	14	35	451	600,656	1.96
differen	59	8	18	317	10,338	0.03
dij	229	165	175	3,616	163,675	0.54
hanoi	13	4	6	57	204,762	0.67
ili	502	152	233	3,943	318,940	1.04
iso	30	9	11	248	1,960,219	6.41
knight	49	14	17	230	371,945	1.22
meta_qsort	63	11	25	397	49,512	0.16
mm	58	21	23	328	1,637,637	5.35
mumath	32	11	21	177	139,368	0.46
nand	360	60	133	2,978	1,003,634	3.28
nrev450	12	6	6	951	817,669	2.67
poly_10	80	14	33	463	346,341	1.13
pri2	26	6	9	108	549,123	1.80
prover	78	23	39	525	81,908	0.27
puzzle	99	10	25	991	551,305	1.80
qs2000	17	6	7	4,075	417,799	1.37
queens_8	27	9	11	120	580,514	1.90
query	72	55	56	289	197,938	0.65
reducer	265	67	119	1,706	299,777	0.98
sdda	250	38	89	1,296	93,478	0.31
serialize	39	12	16	231	32,408	0.11
shoppers	282	78	116	2,493	1,005,868	3.29
tak	16	2	3	58	874,626	2.86
tri	54	41	42	1,411	2,432,532	7.95
turtles	86	44	46	534	238,618	0.78
waltz	89	21	38	961	325,703	1.06
zebra	33	7	11	272	164,869	0.54
Total	4,191	1,445	2,112	38,871	30,589,208	

Table 2. Static and dynamic dimensions of each program.

Instruction Referencing

More than 30 millions WAM instructions were executed. We found that the most common instruction is the *unify_variable_x*(read_mode) which doubles the frequency of the next most common instruction, *get_list*.

Graph 1 shows the relative frequency for all the WAM instructions. Those instructions that execute in write mode will be counted as its respective write instruction.



Graph 1. Frequency of the WAM Instructions.

Memory Referencing

The memory space was made large enough so that, the entire heap, environment, choice point and trail stack would fit, and no garbage collection would be required. For the PDL stack a small vector was used. Table 3 summarizes the memory reference distribution by area. Graph 2 plots the data references by area.

		Times			% From Total			% From Area	
		Read	Write	Total	Read	Write	Total	Read	Write
.	Choice Point	17,821,756	13,589,178	31,410,934	27.3%	20.9%	48.2%	56.7%	43.3%
	Environment	4,821,309	7,443,430	12,264,739	7.4%	11.4%	18.8%	39.3%	60.7%
	Heap	12,948,296	5,979,385	18,927,681	19.9%	9.2%	29.0%	68.4%	31.6%
	Trail	1,073,058	1,213,132	2,286,190	1.6%	1.9%	3.5%	46.9%	53.1%
	PDL	105,120	173,304	278,424	0.2%	0.3%	0.4%	37.8%	62.2%
	Total	36,769,539	28,398,429	65,167,968	56.4%	43.6%			

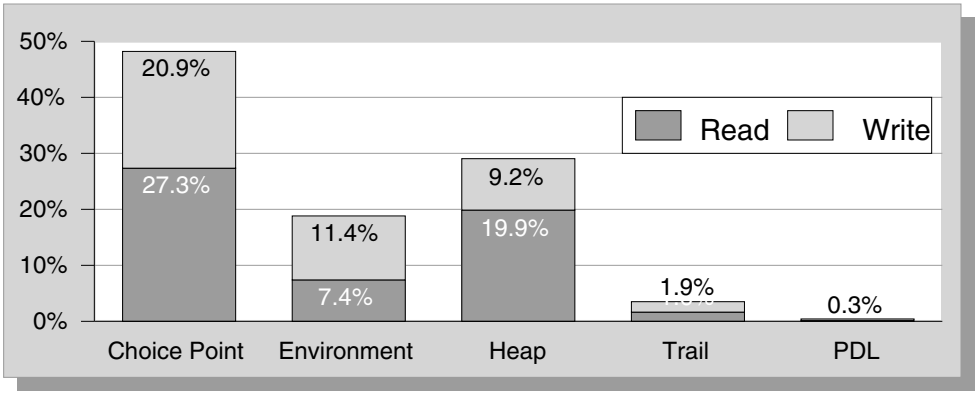
Table 3. Memory References.

As is shown in the Table 3, the area with more memory accesses is the Choice Point with almost 48% of the memory traffic. The ratio writes to reads is bigger than 1 for the Environment area, since many environment words are allocated (four environment words are written for bookkeeping control) and never read because of failure. Also we should take into account that all environment words are initialized before the first call.

The PDL ratio between writes and reads is greater than 1.65. This is because when unifying structures, the entire structures are pushed into the PDL, however if the unification fails they will be discarded and never read.

On average there are 2.1 memory references per WAM instruction.

These statistics differ slightly from those presented by Tick [10]. References to the Heap are less frequent (23%) and Environment references are more frequent (19.2%). This could be due to the different mix of programs.



Graph 2 Percentage of Memory References by Area.

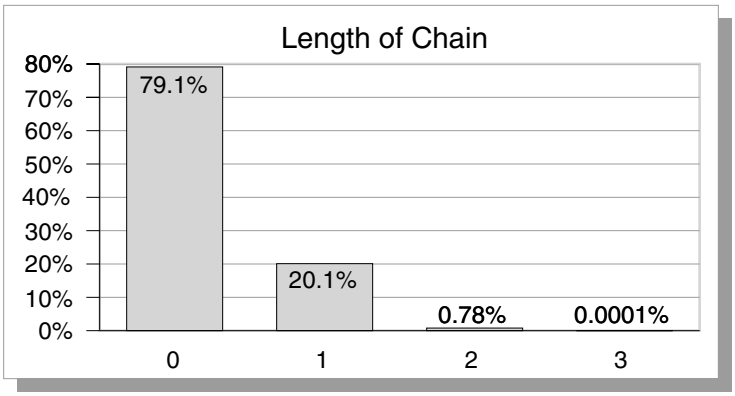
Dereferencing Chain

Table 5 shows the dereferencing chain length for all the programs. Each row represents the number of extra reads that had to be done to completely dereference a variable. For example, if the dereferencing starts in a register and it is not a reference, the length is 0. If dereferencing starts from a memory word (read-dereference) and after the first read, the content is a non-reference or a reference pointing to itself, the length is also 0.

In our simulation 11,678,296 (61%) dereferences were started from registers; 68% of them were of length 0 (the content of the register was not a reference). A total of 7,417,127 read–dereferences (a dereference that starts in a memory word) were executed. The average length of the dereferencing chain is 0.22 dereferences. Graph 3 plots the percentage for the length of the dereferencing chain.

		Times	%
chain length	0	15,107,906	79.1%
	1	3,838,267	20.1%
	2	149,230	0.78%
	3	20	0.0001%

Table 5

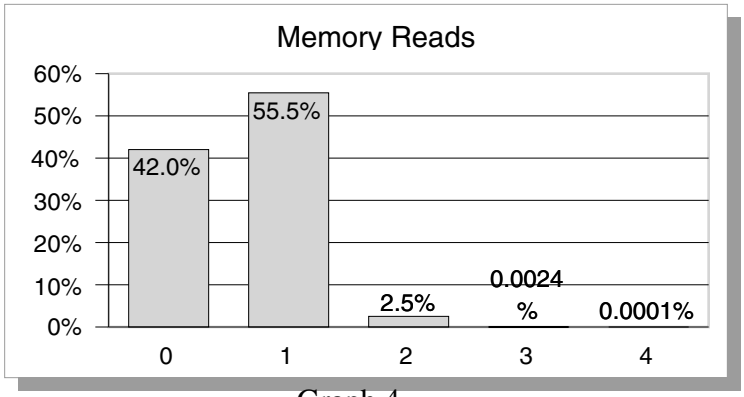


Graph 3

Tick [10] states that the average length of the dereferencing chain is 0.32 and 67% of dereferencing is of length 0 and 32.3% is of length 1. The length for dereferencing given by Touati and Despain are: 63.5% are of length 0, 36.3% of length 1, and 0.2% of length 2. As can be seen in our benchmarks the chain length is shorter.

		Times	%
Memory Reads	0	8,023,910	42.0%
	1	10,589,613	55.5%
	2	481,419	2.5%
	3	461	0.0024%
	4	20	0.0001%

Table 6



Graph 4

Other result that we found interesting is the number of memory reads needed to dereference. This is shown in Table 6 and Graph 4. Dereferences that start in a register and its value is not a reference do not need to read from memory. On average, 0.61 memory reads are needed to dereference a variable.

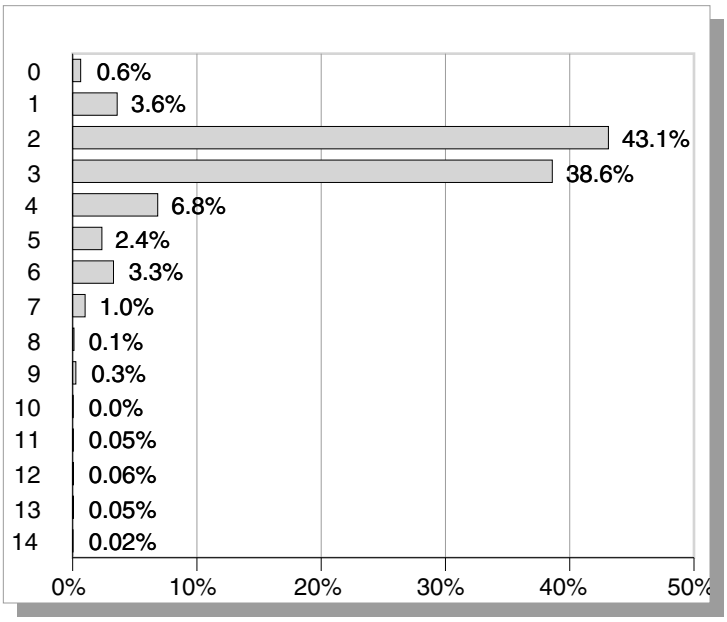
Call

We executed 1,733,457 procedure calls (call or execute instructions). Table 7 shows the number of times a call with n number of arguments occurs and also shows the accumulated percentage. Graph 5 plots the percentage of each call with different number of arguments.

The average number of arguments in a call is 2.8. Notice that more than 99% of the calls have less than 8 arguments.

		Times	%	Acc %
Call Size	0	11,171	0.6%	0.6%
	1	62,103	3.6%	4.2%
	2	747,345	43.1%	47.3%
	3	669,061	38.6%	85.9%
	4	118,652	6.8%	92.8%
	5	40,776	2.4%	95.1%
	6	56,961	3.3%	98.4%
	7	17,370	1.0%	99.4%
	8	1,827	0.1%	99.5%
	9	4,467	0.3%	99.8%
	10	659	0.0%	99.82%
	11	792	0.05%	99.87%
	12	1,002	0.06%	99.93%
	13	947	0.05%	99.98%
	14	324	0.02%	100.0%

Table 7



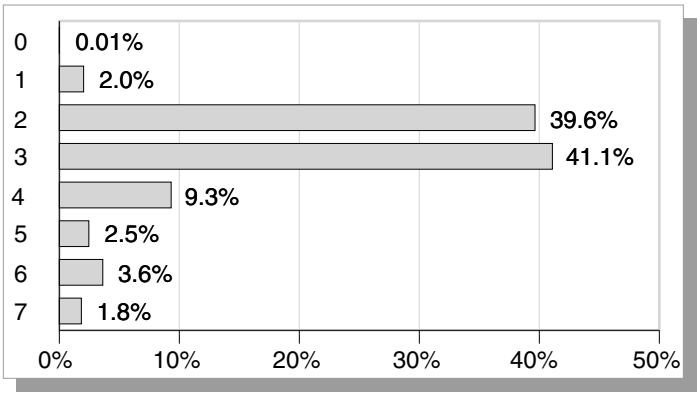
Graph 5

Choice Points

In our simulation 1,187,169 Choice Points were created with an average size of 10.9 words, 2.9 words for the argument registers and 8 words for bookkeeping. Table 8 shows the number of occurrences, its relative percentage and accumulated percentage of the creation of a Choice Point with n arguments. Graph 6 plots the percentage. All Prolog procedures that have more than 7 arguments will create a Choice Point with only 7 arguments saving the 7th and up arguments in a structure pointed by the 7th argument.

	Times	%	Accu %
0	117	0.01%	0.01%
1	24,047	2.0%	2.0%
2	470,608	39.6%	41.7%
3	487,787	41.1%	82.8%
4	110,636	9.3%	92.1%
5	29,193	2.5%	94.5%
6	43,001	3.6%	98.2%
7	21,780	1.8%	100.0%

Table 8



Graph 6

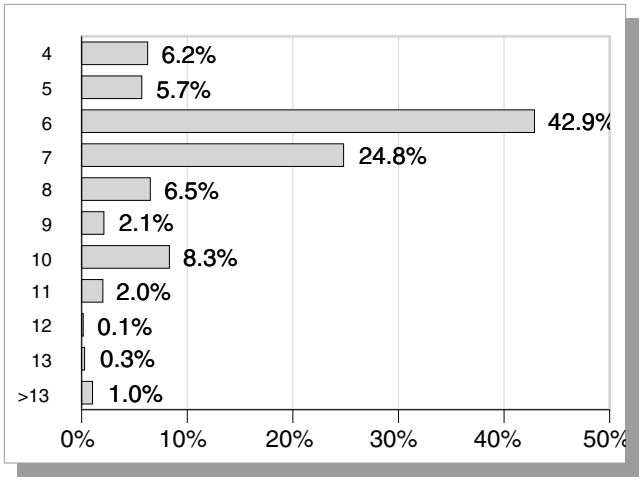
Also, we noticed that 776,080 choice points (65.4%) were removed by the *trust* and *trust_me_else* instructions, 205,248 (17.3%) by the instruction *cut* and 212,882 (17.9%) by the instruction *cutb*. The instruction *cut* trimmed an average of 0.76 choice points each time it was executed. For the instruction *cutb* the average was 0.98 choice points.

Environments

In our simulation 979,287 environments were created. The largest environment had 48 words, the mean size of the environment was 6.85 words, 4 words are for bookkeeping, which gives an extremely high overhead of 58%. This overhead can be reduced (as we pointed before) if the Choice Point pointer is saved only when the procedure has a cut. Table 9 shows the number of occurrence, the percentage, and the accumulated percentage for the *allocate* instruction, Graph 7 plots the percentage.

		Times	%	Accu %
AllocateSize	4	61,178	6.2%	6.2%
	5	55,748	5.7%	11.9%
	6	419,913	42.9%	54.8%
	7	242,910	24.8%	79.6%
	8	63,676	6.5%	86.1%
	9	20,630	2.1%	88.2%
	10	81,380	8.3%	96.5%
	11	19,652	2.0%	98.5%
	12	1,444	0.1%	98.7%
	13	2,707	0.3%	99.0%
	>13	10,049	1.0%	100.0%

Table 9



Graph 7

As we mention earlier the garbage collector requires that all environments words should be initialized before the first call to a procedure. The compiler will only generate the new instruction *init* for those environments words that have not been initialized. This technique saves many unnecessary writes to the environment. From our simulations we get that only 2.8% of the environment words created were initialized by the *init* instruction.

These *init* instructions are executed just before a call, since a failure can occur in between the allocation of the environment and the initialization of the words, this reduces the number of writes to the environment.

Unification

In this section we present the statistics for unification. There are two kinds of unification; one is the general unification where the types of the arguments to be unified are unknown, this is called by the *get_value*, *unify_value* (read mode), *unify_unsafe_value* (read mode) and some escape instructions. The other kind of unification is when we know that the type of one of its parameters is a constant. This is called by the *get_constant*, *get_nil*, *unify_constant* (read mode), *unify_nil* (read mode) and some escape instructions. Forty-seven percent of all the unifications are of the second type.

		First Argument				
		Constant	List	Structure	Reference	Total
Second Argument GU	Constant	291,647	7,217	6,208	253,517	558,589
	List	4,672	3,940	0	135,777	144,389
	Structure	150	0	29,918	50,524	80,592
	Reference	127,672	49,206	99,171	102,796	378,845
	Total	424,141	60,363	135,297	542,614	1,162,415

Table 10. Number of Unifications for each kind of Unification.

Table 10 shows the number of occurrence for each type of unification for the general unifier (subsequent unifications for recursive unification are not included) and Table 11 shows the percentage. Some interesting data is that 70% of all unifications are done with a reference and 3.0% are unifications between compound structures (lists and structures) which requires recursive unification. Table 12 shows the occurrence in percentage for unification with a constant.

		First Argument				
		Constant	List	Structure	Reference	Total
Second Argument GU %	Constant	25.1%	0.6%	0.5%	21.8%	48.1%
	List	0.4%	0.3%	0.0%	11.7%	12.4%
	Structure	0.0%	0.0%	2.6%	4.3%	6.9%
	Reference	11.0%	4.2%	8.5%	8.8%	32.6%
	Total	36.5%	5.2%	11.6%	46.7%	100.0%

Table 11. Relative Percentage for each kind of Unification.

	%		
	Const	Fail	Ref
escape	7.2%	0.0%	92.8%
get_constant	51.3%	0.6%	48.1%
get_nil	28.9%	18.3%	52.8%
unify_constant	99.0%	0.1%	0.9%
unify_nil	51.1%	5.7%	43.3%
Average	49.2%	2.4%	48.3%

Table 12. Unifications with a constant.

Holmer et. al [7] gives statistics for the general unifier. They found that 7.4% are quick failures (both arguments are constants and not equal or unifications between a list and a structure, a structure and a constant or a list and a constant) (16.3% for us), 4.0% are quick success (both arguments are identical) (10.9% for us), 6.7% are unifications where the first argument is a variable and the other is not (37.9% for us), 62.9% are unifications where the second argument is a variable and the other is not (17.8% for us), 15.8% are unifications between two variables (8.8% for us) and 3.0% were recursive unifications (3.0% for us).

Trailing

While binding variables 2,550,696 trails were attempted, but only 1,213,132 (47.68%)

succeeded. Nineteen percent of all writes to the heap and the environment were attempted to trail. Seventy one percent (858,430) of the successful trails were done for bindings of heap variables. The rest of the trailings were done for bindings of environment variables.

Backtracking

While executing the Prolog programs 1,422,314 backtracking occurred. Of these 3.4% were caused by the *Fail* instruction and 82% (1,165,796) were shallow backtracks. During failure 1,073,058 bindings were untrailed, which accounts for 89% of the bindings trailed. Around 51% (549,134) of the words untrailed were for shallow backtracks. For shallow backtracking the average sizes of the different memory areas were: The mean size of the untrail stack 0.47 words, the mean size of reclaimed environment area was 1.67 words, the mean size of reclaimed heap was 0.48 words. For deep backtracking the average sizes were: The mean size of the untrail stack 2.04 words, the mean size of reclaimed environment area was 7.91 words, the mean size of reclaimed heap was 5.96 words. All this data is summarized in Table 13.

	Mean Size		
	Shallow	Deep	Both
Untrail	0.47	2.04	0.75
Heap	0.48	5.96	1.47
Environment	1.67	7.91	2.80

Table 13.

Unsafe Variables

The WAM instruction *put_unsafe_value* will check if the word being saved in the argument register is unsafe. In other words, if the word is a reference to a variable in the current environment, it will create a variable in the heap and the argument register will point to it. Only 0.3% of the *put_unsafe_value* were unsafe and there was a need to create a variable in the heap. The *unify_unsafe_write_x* and *unify_unsafe_write_y* instructions are similar to the *put_unsafe_value* instruction, however, these instructions will check if it is a reference that points to the environment area. For *unify_unsafe_write_x* 0.4% were unsafe and for *unify_unsafe_write_y* 5.1%.

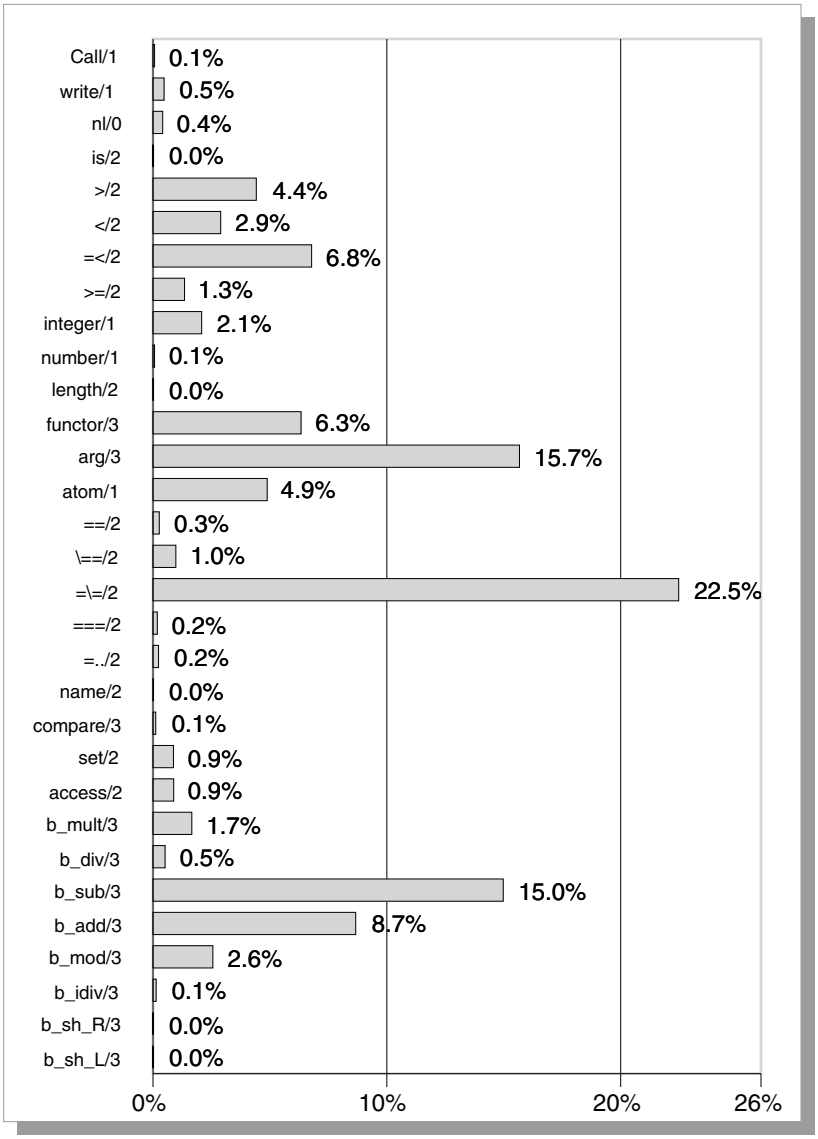
Escapes

The *escape* instruction is an extension of the WAM instruction set, its purpose is to call a routine which is not defined by the WAM. The escape routines range in complexity from simple things as arithmetic operations to more complex as I/O. In the PLM all the escapes are done by the main processor, in the VLSI-PLM some of the *escapes* are handled by the processor (mainly the arithmetic escapes).

In our simulation more than 4.0% of the instructions executed were *escapes* instructions. Table 14 and Graph 8 show the frequency and percentage for the *escapes* called. About 29% of the *escapes* were arithmetic operations (b_mult, b_div, b_sub, b_add, b_mod, b_idiv, b_sh_R and b_sh_L). For these *escapes*, an arithmetic operation is done with argument registers A1 and A2 and the result is binded to the argument register A3. This creates an overhead since data has to be moved between registers so that, inputs and outputs are set before the *escape* instruction.

	Times
Call/1	855
write/1	5,967
nl/0	5,239
is/2	9
>/2	55,719
</2	36,520
=</2	85,577
>=/2	17,017
integer/1	26,265
number/1	828
length/2	111
functor/3	79,856
arg/3	197,698
atom/1	61,600
==/2	3,377
\==/2	12,331
=\=/2	283,598
===/2	2,321
=../2	2,995
name/2	180
compare/3	1,492
set/2	11,032
access/2	11,164
b_mult/3	20,949
b_div/3	6,500
b_sub/3	188,927
b_add/3	109,352
b_mod/3	32,299
b_idiv/3	1,690
b_sh_R/3	5
b_sh_L/3	5

Table 14



Graph 8

Conclusions

In this paper we presented some dynamic statistics obtained by running Prolog programs using the WAM model. This information can direct the effort of compiler writers and hardware designers to analyze and improve those areas where a feature of Prolog is more frequently used.

Some possible optimizations which require more research are: Since 74% of the backtracks were shallow and 45% of the memory references were in the choice point area, we can deduce that having a choice point buffer of the current choice point would reduce considerably the memory traffic. Try to remove many unnecessary dereferences because the WAM dereferences are so frequent and short. Because 72% of the unifications are done with a reference, special attention should be given to try to speedup this case. Also, some effort should be put to reduce the number of bookkeeping words that are saved in an environment. Additionally, the number

of argument registers can be reduced since more than 97% of all calls have 6 of less parameters. Finally, instructions for arithmetic operations should be included in the WAM instruction set since they are frequently used.

References

1. Hassan Ait-Kaci *Warren's Abstract Machine,* MIT Press, Cambridge, Massachusetts (1991).
2. Tep P. Dobry, Yale N. Patt, and Alvin M. Despain Performance studies of a Prolog machine architecture. In *Proceedings of the 12th International Symposium on Computer Architecture, Boston, Massachusetts,* IEEE Computer Society Press, Washington, D.C., June 1985, pp. 180-190.
3. Tep P. Dobry *A High Performance Architecture for Prolog,* Kluwer Academic Publishers, Parallel Processing and Fifth Generation Computing(1990).
4. Barry S. Fagin and Tep Dobry The Berkeley PLM Instruction Set: An Instruction Set for Prolog. Tech. Rept. No. UCB/CSD 86/257Computer Science Division (EECS), University of California, Berkeley, September, 1986.
5. John Gabriel, Tim Lindholm, and Ewing L. Lusk, and Ross A. Overbeek A tutorial on the Warren abstract machine for computational logic. ANL-84-84. *Argonne National Laboratory, Argonne, Illinois* (1984).
6. Bruce K. Holmer A Detailed Description of the VLSI-PLM Instruction Set: A WAM Based Processor for Prolog. Tech. Rept. No. UCB/CSD 90/610Computer Science Division (EECS), University of California, Berkeley, Berkeley, Califronia 94720, March, 1989.
7. Bruce K. Holmer, Barton Sano, Michael Carlton, Peter Van Roy, Ralph Haygood, Alvin M. Despain William R. Bush, Joan M. Pendleton, and Tep P. Dobry Fast Prolog with an Extended General Purpose Architecture. In *Proceedings of the 17th Annual International Symposium on Computer Architecture, Seattle, Washington,* IEEE Computer Society Press, Washington, D.C., June 1990, pp. 282-291.
8. Rikio Onai, H. Shimizu, Kanae Masuda, and Morithoshi Aso Analysis of sequential Prolog programs. *Journal of Logic Programming 2*(1986), 119-141.
9. Evan Tick Memory Performance of Lisp and Prolog Programs. In *Proceedings Third International Conference on Logic Programming. In Lecture Notes in Computer Science 225,* Ehud Shapiro, Springer-Verlag, July 14-18 1986, pp. 642-649.
10. Evan Tick *Memory Performance of Prolog Architectures,* Kluwer Academic Publishers (1987).
11. Evan Tick *Parallel Logic Programming,* MIT Press, Cambridge, Massachusetts (1991).
12. Hervé Touati and Alvin M. Despain An Empirical Study of The Warren Abstract Machine. In *Proceedings of the 1987 Symposium on Logic Programming, San Francisco, California,* August 1987, pp. 114-124.
13. Hervé Touati A Prolog Garbage Collector for Aquarius. Tech. Rept. No. UCB/CSD 90/610Computer Science Division (EECS), University of California, Berkeley, Berkeley, Califronia 94720, August, 1988.
14. Peter Van Roy *A Prolog Compiler for the PLM. Master's Report Plan II*, Masters thesis, Also techreport 84/203, Computer Science Division, University of California, Berkeley, November 1984.
15. David H.D. Warren An abstract Prolog instruction set. Tech. Rept. 309SRI International, Stanford, California, October, 1983.