An improvement to PARMA variable trailing.

Tom Schrijvers, Bart Demoen

Report CW 326, December 2001



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

An improvement to PARMA variable trailing.

Tom Schrijvers, Bart Demoen

Report CW 326, December 2001

Department of Computer Science, K.U.Leuven

Abstract

The PARMA variable representation requires *value trailing*. This leads to a trail stack consumption that is about twice as large as for the WAM. We present two simple techniques by which this disadvantage is largely countered, without causing undue overhead. We also investigate the impact of a similar trailing improvement for backtrackable destructive assignment. Additionally we speculate on several ways to improve the time and space efficiency of trailing through analysis.

An improvement to PARMA variable trailing

Tom Schrijvers and Bart Demoen

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

Abstract

The PARMA variable representation requires *value trailing*. This leads to a trail stack consumption that is about twice as large as for the WAM. We present two simple techniques by which this disadvantage is largely countered, without causing undue overhead. We also investigate the impact of a similar trailing improvement for backtrackable destructive assignment. Additionally we speculate on several ways to improve the time and space efficiency of trailing through analysis.

1 Introduction

We will assume working knowledge of Prolog and its implementation without further explanation. For a good introduction to Prolog see [7]; to WAM, see [1, 9].

The disadvantage of the variable representation in the WAM is that testing whether a variable is bound requires dereferencing. Every unification depends on whether the involved variables are bound, hence dereferencing is required quite often. In his PARMA-system Taylor has introduced a different variable representation scheme that does not suffer from a massive dereferencing need, see [8, 5]. In this PARMA scheme all unbound variables are represented as circular chains. In fact every variable points to a cell in such a chain. For every cell in the chain there is a variable that is aliased to the others. Bound variables are represented as atoms or as tagged pointers to structures. No dereferencing is required to verify whether a cell is bound. The tag immediately identifies a cell as being bound or not.

Although the PARMA scheme avoids dereferencing in boundness checks, this improvement over the WAM scheme has its toll. The trailing of unifications in the PARMA scheme is a lot more expensive. Demoen and Nguyen have observed in the dProlog system maximal trails that are on average twice as large as with the WAM scheme, see [3].

In this report we present two new trailing techniques for PARMA variables that largely limit the stack consumption. Experimental results of an implementation of these techniques in the dProlog system indicate that the proposed techniques impose no undue overhead.

In the next section we briefly explain the classic PARMA variable trailing. In section 3 the two new techniques are presented. Experimental results of the implementation in dProlog are compared to the classic PARMA trailing scheme and to other variable representations in section 4. Section 5 evaluates a similar improvement to trailing in a different setting: trailing for backtrackable destructive assignment in the hProlog system. Two new analysis based optimisations are proposed in section 6. Finally section 7 concludes this report by discussing the impact of the proposed techniques on existing applications of the PARMA scheme.

2 Classic PARMA variable trailing

The classic PARMA variable scheme uses a kind of trailing, named *value trailing*, described by the following C macro:

```
#define trail(p, tr, bh) \
   if (p < bh) {         \
      *(tr++) = *p;      \
      *(tr++) = p;       \
   }
```

In this macro `p` is the location of a cell in a PARMA chain, `tr` is the top of the trail stack and `bh` is the position of the top of the heap at the beginning of the last choicepoint.

This value trailing stores a cell (pointer) location together with its contents on the trail stack. Untrailing simply consists of reading the pointer from the trail, next reading the value location and finally storing the value in the location. If the cell did not exist before the last choicepoint, it does not have to be restored upon backtracking. This is accomplished by not trailing it in the first place.

The value trailing is used in two different situations: variable-variable unification and variable-nonvariable unification.

2.1 Variable-variable unification

Variable-variable unification implies the merging of the chains of the two involved variables. This merging is obtained by swapping the successors of the cells that the two variables point to. This swapping of successors only changes the two cells directly referenced by the variables. Hence they are the only ones that have to be trailed. Simplified, this looks like:

```
/* variable-variable unification X = Y
 * Precondition:
 *    X and Y point to a cell in a different chain
 */
trail(X,tr,bh);
trail(Y,tr,bh);
oldX = *X;
X = *Y;
Y = oldX;
```

The two cells of the variables are trailed independently.

2.2 Variable-nonvariable unification

Variable-nonvariable unification pulls the entire chain apart. Every cell in the chain is set to the nonvariable. Since every cell in the chain is changed, every cell also has to be trailed, in order to be able to reconstruct the chain. The combined unification-trailing code looks like this:

```
/* variable-nonvariable unification X = T */
start = X;
do {
   next = *X;
   trail(X,tr,bh);
   *X = T;
   X = next;
} while (X != start);
```

3 Two improvements for reducing the trail stack size

The two proposed techniques are derived from the observation that both variable-variable and variable-nonvariable trailing store redundant information. For both kinds of unifications we present a different kind of trailing that avoids redundancy as much as possible. However we do not abandon value trailing altogether, for reasons that will become clear when variable-nonvariable trailing is explained.

The trail stack will be used for three kinds of trailing. It is possible to make a distinction between these different kinds of trailing without a space overhead because the first two bits, the tag bits, in every unbound variable address are zero.

3.1 Improved variable-variable trailing

Consider the the case where both involved variables have to be trailed, i.e. they are older than the last choicepoint. In the classic PARMA approach, both variables are value trailed. This takes up four slots in the trail stack. Looking at the variable-variable unification, the swapping of successors, it is easy to come up with an inverse operation that is not as wasteful. Since swapping is its own inverse and we already know it only requires the addresses of the two variables, it is sufficient to only trail those addresses. For this case, trailing looks like:

```
*tr++ = X;
*tr++ = Y | 0x1;
```

The first tag bit of the cell address referenced by variable `Y` is set to indicate that this is an improved variable-variable trailing. Upon untrailing, a slot is read from the trail and its tag is detected to be 1, the tag is cleared, the next slot is also read, the two read values are treated as addresses of cells and their contents is swapped.

Note that we do not abandon value trailing in the cases where not both cells have to be trailed.

3.2 Improved variable-nonvariable trailing

Suppose that all cells in the chain of the variable have to be trailed. In that case the classic trailing scheme will use 2n slots, where n is the length of the chain that is being trailed. Value trailing stores the address of every cell in the chain together with its contents, which happens to be the address of the next chain. This effectively stores the address of every cell twice.

The obvious optimisation is to store the address of every cell in the chain just once. From these addresses the chain can easily be reconstructed, while requiring only n slots of the trail stack.

Unfortunately this is a very restricted case: all cells in the chain have to be trailed. Even if not all cells have to be trailed, we could still trail all of them. Hence the cost would still be n slots. The classic trailing scheme on the other hand, only trails those cells it has to, say k of n cells. The 2k slots consumed by the old approach may very well be less than the n slots used by the new approach. In fact this quite often appears to be the fact, since for many chains $n = 1$ and $k = 0$.

Fortunately, the value trailing that we did not abandon comes to the rescue. Consider what happens if we use the above scheme and only trail the addresses of the k cells that have to be trailed. After untrailing them, all cells that had to be trailed have a correct successor except those that had as their successor a cell that was not trailed.

Consider such a trailed cell with an untrailed successor Y. Since this untrailed successor Y did not exist before the last choicepoint, it must have been merged into the chain after the last choicepoint in a variable-variable unification X = Z. The variable Z in this unification either did exist before the last choicepoint or it did not:

- If Z did not exist before the last choicepoint, then Z might very well just be Y. Whether Z is equal to Y or not does not matter. The fact is that in this case the X-Z unification involved only the trailing of X, a value trailing. Hence after the untrailing of the chain, the untrailing process of X will encounter this value trailing of X before it encounters any other trailing of X or before it stops its backtracking. So thanks to the value trailing X, the proper value of X before the variable-nonvariable and variable-variable unification will be restored.
- If Z did exist before the last choicepoint then it must have been swap trailed with X. Hence untrailing will encounter and reverse this swap trailing. This ensures that the value of X before the trailing of the chain is recovered.

Hence it is safe in the general case to omit all cells that are younger than the last choicepoint from the chain trailing. Thus the new trailing scheme consumes just k slots of the trail stack, which is always less than or equal to the 2k slots for the classic trailing scheme.

The simplified chain trailing code then looks like this:

```
/* unification and trailing of X = T */
start = X;
next = *X;
*X = T;
while (X > bh & next != start) {
   X = next;
   next = *next;
   *X = T;
}
if (X < bh) {
   *tr--;
   X = X | 0x1;
   do {
      if (X < bh) {
         *(++tr) = X;
      }
      X = next;
      next = *next;
      *X = T;
   } while {X != start);
   *tr = *tr | 0x2;
   tr++;
}
```

The first tag bit is set to indicate the start of the chain and the second tag bit is set for the end of the chain. When untrailing sees that the second bit is set, it will look for a slot that has the first bit set, meanwhile reading and restoring the cells of the chain.

4 Experimental results

The two new kinds of trailing discussed in the previous section have been implemented in the dProlog system. The trailing and untrailing code presented in the previous section has been optimized for the most common case. Table 1 how many times four kinds of trailing appear in different benchmarks. *Value trailing* is the classic value trailing that is still used in some cases of variable-variable unification. *Variable swaps* is the new trailing for variable-variable unification. Both *s*hort chains and *long chains* are the new trailing for variable-nonvariable unification. The former is for chains of length 1 and the latter is for longer chains. Clearly, trailing is most often, for many benchmarks even solely, used for short chains. Hence the trailing and untrailing code has been optimized to perform specially well for this case.

Table 2 shows the timing and maximal trail use for several benchmarks. All measurements were made on a Pentium 166 MHz 96 Mb. Time is given in 1/100 s. The time applies to the number of runs between brackets after the name of the benchmark. For every benchmark four timings have been measured. The first measured is not taken into account. Of the other three the smallest time is given in the table. The maximal trail is the maximum amount of used slots on the trail stack. The time difference between the classic and the new scheme appears to be negligible. Five benchmarks have the same time, the new scheme is faster in four cases and the classic scheme is faster in seven cases. The differences never amount to more than a few tens of milliseconds. Relatively the improved scheme is at most 7.0% slower, for the send benchmark. Overall the improved scheme is only .6% slower, which is quite acceptable.

The differences in the maximal trail however are substantial. While the two new schemes consume only half of what the classic value trailing does, the maximal trail is not guaranteed to be half that of the classic

Benchmark	short chains		long chains		variable swaps		value trails	
boyer	95,977	100%	0	0%	0	0%	0	0%
browse	198,679	87.3%	28,900	12.7%	0	0%	0	0%
cal	199,998	100%	0	0%	0	0%	0	0%
chat	312,551	88.7%	25,200	7.3%	2,235	.6%	12,320	3.5%
crypt	53,250	100%	0	0%	0	0%	0	0%
ham	281,690	99.8%	648	.2%	0	0%	0	0%
meta_qsort	94,255	59.5%	64,250	40.5%	0	0%	0	0%
nrev	61	100%	0	0%	0	0%	0	0%
poly_10	73,332	100%	0	0%	0	0%	0	0%
queens_16	211,673	100%	0	0%	0	0%	0	0%
queens	599,016	100%	0	0%	0	0%	0	0%
reducer	40,762	38%	60,960	56.9%	5,440	5.1%	0	0%
sdda	99,644	68.6%	0	0%	13,200	9.1%	32,400	22.3%
send	210,473	100%	0	0%	0	0%	0	0%
tak	477,068	100%	0	0%	0	0%	0	0%
zebra	915,630	69.8%	0	0%	354,750	27%	41,700	3.2%
comp	657,855	73.3%	175,240	19.5%	27,929	3.1%	36,634	4.1%

Table 1: The number of times the different kinds of trailing are used in a benchmark.

Benchmark	Time		Maximal trail	
	cparma	iparma	cparma	iparma
boyer(1)	166	166	115,366	57,683
browse(1)	175	174	2,326	1,163
cal(10)	104	105	112	56
chat(5)	78	77	932	498
crypt(200)	79	79	118	59
ham(2)	95	97	196	98
meta_qsort(125)	93	90	3,238	1,894
nrev(5000)	107	109	112	56
poly_10(10)	55	55	13,470	6,735
queens_16(2)	116	117	176	88
queens(10)	174	176	170	85
reducer(20)	33	33	4,850	2,555
sdda(1200)	64	66	326	201
send(10)	71	76	122	61
tak(10)	158	153	95,522	47,761
zebra(30)	140	140	406	205
relative average	100%	100.6%	100%	51.7%
comp	1,276	1,280	644,170	337,856

Table 2: Timing and maximal trail results for both the classic (cparma) and the improved trailing scheme (iparma).

Benchmark	Time			Maximal trail		
	hpvar	lsvar	iparma	hpvar	lsvar	iparma
boyer(1)	166	161	166	57,677	58,635	57,683
browse(1)	178	175	174	1,150	1,158	1,163
cal(10)	106	104	105	49	56	56
chat(5)	70	70	77	362	412	498
crypt(200)	80	79	79	53	79	59
ham(2)	92	91	97	90	97	98
meta_qsort(125)	82	81	90	822	829	1,894
nrev(5000)	96	102	109	49	56	56
poly_10(10)	55	55	55	6,729	6,736	6,735
queens_16(2)	121	114	117	82	89	88
queens(10)	171	171	176	79	86	85
reducer(20)	29	27	33	1,382	1,444	2,555
sdda(1200)	60	60	66	111	118	201
send(10)	78	70	76	55	66	61
tak(10)	154	144	153	47,755	47,762	47,761
zebra(30)	115	113	140	137	148	205
relative average	102.4%	100%	107.2%	91.8%	100%	118.7%
comp	1,325	1,250	1,280	228,396	238,484	337,856

Table 3: Timing and maximal trail results for heap variables (hpvar), classic WAM variables (lsvar) and PARMA variables with the improved trailing scheme (iparma).

scheme since value trailing was not entirely abandoned. Yet the experimental results show that in practice the new trailing scheme uses exactly half the trailing space of the classic scheme for most of the benchmarks. This is the case for eleven out of sixteen benchmarks. Four of the other five use less than 55%. The fifth and worst benchmark, sdda, still only consumes 61.7%. The new PARMA trailing scheme thus truly is an improvement.

The larger comp benchmark confirms the results of the smaller benchmarks. The improved scheme is .3% slower and uses only 52.4% of the maximal trail stack of the classic scheme.

Table 3 compares the improved PARMA trailing scheme to two other variable representations that are available in the dProlog system: the WAM representation and the heap representation. Both differ only in where they initialize permanent variables. The heap representation always does this on the heap whereas the WAM representation does this in the environment when possible.
The timings do not show too big a difference for most benchmarks, though in the cases that that they are significant this is always to the disadvantage of the improved PARMA scheme.
Although the improved PARMA scheme already halves the trail use, it still has the potential to use a lot more than the WAM scheme. Variable-variable unification takes up 0 or 2 trail slots, while only 0 or 1 slot for the WAM scheme. Variable-nonvariable unification takes up k slots, while only 0 or 1 in the WAM scheme. The traditional WAM representation consumes slightly more trail space when it has to globalize local stack UNDEF variables to the heap. The experimental results show that the heap representation is the most economic, while the classic WAM and improved PARMA scheme are about equal for most small and large benchmarks. The improved PARMA scheme is significantly worse for several medium sized benchmarks. On the other hand it is noticeably better than the classic WAM presentation for boyer.

For the comp benchmark, the improved PARMA scheme is a little faster than the heap variables scheme and a little slower than the WAM scheme. The maximal trail stack use is considerably worse than for the other benchmarks: it is 41.7% more than that of the WAM scheme, while the heap variables scheme is 4.2% less than the WAM scheme.

All in all the improved PARMA scheme is still no match for the WAM representation, although the gap has been closed significantly.

5 Another assessment of the cost of tagging trail entries

The above experiments were conducted in a dProlog version with the PARMA representation of variables. In this section, we will use hProlog - a descendant of dProlog - with heap variables only, but with a value trail: one trail entry consists of an address and a value which is the the old value at that address. For uniformity, we have chosen this layout for trail entries even for ordinary unification, in which case the address and the (old) value are actually the same. This results in a doubling of the trail stack as compared to usual WAM trailing. One can of course use here also tagged trail entries and in this way save space. This idea is implemented in several systems and we intend here to measure the cost of the uniformity we had chosen for. The following three versions of hProlog are made:

- the default one with uniform address-value trailing – referred to later as *valuetrail*
- a trail with tagged trail entries: when the address and the value is the same, only one cell is pushed and no tag is added (or one could say that the tag is zero); if the value is different from the address, the address is tagged with tag 0x1 and pushed on the trail stack together with the value - *taggedtrail*
- a trail which does not support backtrackable destructive assignment, i.e. as in WAM – *wamtrail*

Note that *valuetrail* and *taggedtrail* offer the same functionality w.r.t. backtrackable destructive assignment. Valuetrail uses twice the amount of trail cells as the other two versions on the benchmark suite.

The measurements in Table 4 were made with hProlog 1.4, on a Pentium III, 500MHz, 128 Mb. The timings represent the best of 4 timings.

Benchmark	valuetrail	taggedtrail	wamtrail
boyer	880	870	860
browse	220	210	210
cal	280	290	300
chat	230	220	230
crypt	170	160	160
ham	340	340	330
meta_qsort	220	220	230
nrev	260	240	240
poly_10	130	120	120
queens_16	310	310	290
queens	530	520	520
reducer	70	60	70
sdda	160	160	160
send	190	190	190
tak	330	300	290
zebra	410	380	390
Total	19,910	19,330	19,220
comp	3,260	3,240	3,160

Table 4: Comparing different trail implementations in hProlog

The row labelled "Total"in Table 4 gives the total time (in msecs) it took to execute the total suite with small benchmarks. It indicates that an overall performance gain of about 3% is possible for the taggedtrail version compared to the valuetrail version. Wamtrail is only slightly better.

The *comp* benchmark shows a smaller difference between the value trail and the tagged trail, and a larger difference with the ordinary WAM trailing.

In the future hProlog might support also undo actions on the trail. Together with the support for backtrackable destructive assignment, a tagged trail seems an acceptable solution performance wise.

6 Further room for improvement

We believe it is not possible to further improve on the efficiency of the PARMA variable representation without at least a little program analysis. Although there is often little information to be gained from unannotated Prolog programs, the analyses we propose here work for predicates that are not declared to be `dynamic`. This at least guarantees that the predicates under consideration are not going to change at runtime.
The analyses that are proposed here heavily rely on implementation details. In [2] Debray presents a simple code improvement scheme for this kind of low level optimisations. He uses this framework to implement trail check elimination, which is more or less the opposite of the first analysis we propose and which is for the WAM variable representation. Debray tries to determine what variables do not have an UNDEF cell that is older than the most recent choicepoint, in order to avoid their trailing at compile time, while we will try to prove that certain cells definitely need to be trailed.

6.1 "Must trail" analysis

While an analysis that can decide whether it is safe to omit the trailing of a certain unification requires more accurate determinism information than can be obtained from a traditional Prolog system, the opposite analysis does not. In many cases it is quite easy to tell on a predicate or clause level, what variables in a unification certainly will have to be trailed if they are not ground. Whenever a choicepoint is created in a predicate, either because of multiple clauses or because of an explicit disjunction, it is quite easy to determine on a syntactical level what variables certainly will have to be trailed if they are not bound. This analysis can be used to select specialized unifications that do not perform the `p > bh` test. This would make it possible to decide at compile time that for some variable-variable unifications variable swap trailing will be necessary.
Consider for example the append predicate:

```
append([],Y,Y).
append([X|Xs], Y, [X | Z]) :- append(Xs, Y, Z).
```

Because this predicate has two clauses it creates a choicepoint if it is called with the first argument unbound. Since the argument did exist before this choicepoint, as it is passed in from a call site, the test `p > bh` in the unification with `[]` will always fail. This also applies to other predicates with more clauses, to disjunctions of the form `(X = a; ...; X = z)` where X existed before the disjunction.

Also this analysis does not strictly rely on the PARMA scheme. The WAM scheme could benefit equally from it.

6.2 Avoiding identical trailing on disjunct paths

Many disjunctions unify the same variable with a nonvariable in all, or at least a subset of consecutive disjuncts. Each time the same PARMA chain is trailed in exactly the same location on the trail stack. It would not be too difficult to only have the first disjunct actually write on the trail stack. Subsequent disjuncts just increase the pointer to the top of the trail stack with an amount equal to the length of the chain.
Consider this simple set of facts:

```
f(a).
f(b).
f(c).
```

If the the goal `f/1` is called with a variable argument `X`, the first two clauses and possibly the third clause too are going to trail the cell that this variables point to. Since the first and the second clause have an alternative, a choicepoint is introduced before trying the first clause and it is removed upon backtracking to the third clause.

Since the argument is a variable and a new choicepoint has been created, the first clause is going to trail all cells in the PARMA chain of the variable. The same is true for the second clause, after backtracking

over the first clause. Both trailings store the same values in the same locations on the trail stack. Instead of really writing the values on the trail stack, it suffices for the second to simply increment the pointer to the top of the trail stack with an amount equal to the length of the chain. The following specialized unification could be used for the second clause:

```
/* specialized unification and trailing of X = T */
start = X;
do {
   if (X < bh) tr++;
   next = *X;
   *X = T;
   X = next;
} while {X != start}
```

Fortunately, this optimisation does work with a compacting garbage collector because the garbage collector is not active between the backtracking of the first clause and the execution of the second clause. This is the only moment that the cells on the trail stack that are going to be recycled, are outside the upper bound of the trail stack.

For the last clause, things are different. The choicepoint has been removed, hence it is not statically known whether trailing is necessary. Moreover, trailing is not decided for the entire chain at once. A decision has to be made for every cell separately. If some but not all cells in the chain have to be trailed, reuse of all the trail entries is generally not possible. Reuse of the cell trailing is only possible until a cell is encountered that does not have to be trailed. Thereafter no further reuse is possible. Luckily, this covers the most common case of chains of length one.

This analysis could be extended to do reuse over more than one choicepoint. Take for example the following conjunction of disjunctions:

```
( X = 1 ; X = 2 ),
( Y = 1 ; Y = 2 )
```

The second disjunction is independent of the first. In the Ciao prolog system this independence is exploited to parallelized the disjunctions and execute every branch only once. In a sequential setting the second disjunction will be executed once for every branch of the first disjunction instead. Hence when all possible branches are explored `Y` will be unified four times. With the proposed analysis and in case `Y` consists of a chain of cells that all have to be trailed, `Y` would only have to be trailed twice: once for each branch of the first disjunction. However this could potentially be reduced to only once, if we consider the values on the trail stack each time `Y` is trailed. Only the first unification of `Y` will put a new value on the stack. All later ones will overwrite the same location on the stack with the same entry. The problem in optimizing this situation is in recognizing at runtime that the second disjunction has been executed previously. This should be detected cheaply so as not to annihilate the performance gain obtained by sparing out the trailing.

Alternatively the optimization need not be limited to the unification of only one variable in a disjunction. Cases were multiple variables are trailed can be dealt with just as well, depending on what is know about their instantiation. Take for example the disjunction:

```
( X = 1, Y = 2 ; X = 2, Y = 1 )
```

Not only the trailing of `X` can be reused, but also that of `Y`. This additional reuse however is more restricted. It depends on whether `X` is free when it enters the disjunction or not. If `X` is free, then both unifications of it will succeed. Hence if `Y` is also free, its trailing can be reused. If `X` on the other hand is bound to 2, then the first branch will never reach the unification of `Y`. Hence if `Y` is free it cannot count on reusing the trailing in the second branch. So this additional optimization can only be used when it is known that `X` is free. If it is known that `X` is bound, this unification might as well be translated into a switch. If nothing is known about `X`, maybe something similar to indexing for predicates could be attempted.

The basic optimization also applies to the WAM-scheme. Although the potential of reuse is more limited as the WAM scheme would use at most one slot on the stack. This trailing however could be exploited rigorously in the case where is known that the variable is free. Instead of trailing the variable inside the

disjunction, it could be trailed immediately before. Hence the variable's UNDEF cell would be on the trail stack immediately before the new choicepoint. Untrailing hence would not restore the cell and remove the entry from the stack. On the other hand the address of the variable's UNDEF cell can easily be read from the last entry in the stack. Hence no dereferencing is required at all in any but the first branch. It is up to the last branch to remove the entry from the stack. This easily done by decreasing the stack pointer by one. If nothing is known about the freeness of `X`, then trailing and untrailing will have to be done inside the disjunction and dereferencing cannot avoided at zero cost.

7 Conclusion

The experimental results show that the presented new trailing scheme for PARMA variable representation on average consumes nearly 50% less of the trail stack, without undue overhead. The timing results are indeed on par with the classic PARMA trailing scheme. Moreover the new trailing scheme counters the major disadvantage of PARMA over WAM, as the observed results indicate that the maximal trailing space used with WAM and the new PARMA trailing are about equal.
The only substantial current application of PARMA that the authors are aware of is HAL, a strongly typed and weakly moded CLP language with major support for implementing and combining solvers, see [4]. Currently HAL compiles to both SICStus and Mercury. HAL has chosen the PARMA variable representation for its Mercury output because the PARMA representation of ground terms is identical to that of Mercury. Hence HAL is able to reuse all Mercury optimisations for ground terms. A drawback of the garbage collector used in Mercury is that there is no guaranteed ordering of old and new cells on the heaps. Hence all cells have to be trailed. When the new trailing scheme would be used in this setting, it would never use value trailing. Then the maximal trail space used would always be half that of the classic scheme.
Currently an analysis is being implemented for HAL to avoid trailing as much as possible, based on the determinism information either declared by the programmer or inferred by the compiler, see [6]. This analysis exploits its knowledge about the classic trailing scheme. Repeated value trailing of the same variable within a (semi)deterministic path between two choicepoints is detected and eliminated. The new trailing scheme would not allow this kind of optimisation, since it does not always (in this setting even never) use value trailing for variable-variable binding. Yet it is not unlikely that value trailing could be used, based on static analysis. In that case repeated trailing after a value trailing in a (semi)deterministic path could be avoided too. This would have a combined impact on necessary trail space: the two new trailing techniques half the used trail space and value trailing eliminates future trailing. It remains to be seen whether an optimal choice between the different trailing techniques is possible.

References

[1] H. Aït-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] Saumya Debray *A Simple Code Improvement Scheme for Prolog* Journal of Logic Programming vol. 13 no.1, May 1992, pp. 349-366

[3] B. Demoen, P.-L. Nguyen. *So many WAM variations, so little time.* Computational Logic - CL2000, First International Conference, London, UK, July 2000, Proceedings (V. John Lloyd, ed.), Lecture Notes in Artificial Intelligence, vol. 1861, Springer, 2000, pp. 1240-1254.

[4] Bart Demoen, Maria García de la Banda, Warwick Harvey, Kim Marriott, Peter Stuckey. "An Overview of HAL" Proceedings of the International Conference on Principles and Practice of Constraint Programming, Oct. 1999, Virginia, USA, pages 174–188, Springer Verlag

[5] Thomas Lindgren, Per Mildner and Johan Bevemyr *On Taylor's scheme for unbound variables.* Principles and Practice of Constraint Programming, pages 174-188, 1999.

[6] Tom Schrijvers, Bart Demoen, Maria García de la Banda and Peter Stuckey Trailing Analysis for HAL Technical Report 327, Dept. of Computer Science, K.U.Leuven, Dec. 2001.

[7] Sterling Shapiro The Art of Prolog, The MIT Press, 1986.

[8] Andrew Taylor *Parma - Bridging the Performance GAP Between Imperative and Logic Programming.* Journal of Logic Programming, 29(1-3):5-16,1996.

[9] D. H. D. Warren. *An Abstract Prolog Instruction Set.* Technical Report 309, SRI International, Menlo Park, U.S.A., Oct. 1983.