A Novel Term Compression Scheme and Data Representation in the BinWAM

Paul Tarau[1] and Ulrich Neumerkel[2]

[1] Université de Moncton
Département d'Informatique
Moncton, N.B. Canada, E1A 3E9,
tarau@cs.sfu.ca
[2] Technische Universität Wien
Institut für Computersprachen
A-1040 Wien, Austria
ulrich@mips.complang.tuwien.ac.at

Abstract. In this paper we present the novel term representation of the BinWAM (a simplified WAM engine for executing *binary logic programs*) and evaluate its impact in BinProlog, a C-emulated system[3] based on the BinWAM and on the mapping of logic programs to binary Prolog introduced in [13]. Terms in the BinWAM are compressed with a new technique called last argument overlapping which takes advantage of an unconventional untagged pointer representation, called *tag-on-data*. A Cheney-style copy_term algorithm using these term representations is described for BinProlog's fast *copy once* implementation of findall. While BinProlog's performance is competitive with the best commercial Prolog systems, its implementation is significantly simpler. Our analysis shows that this term representation and a limited amount of instruction folding on top of a reduced basic instruction set make the BinWAM a realistic alternative to its more complex forerunner.
Keywords: *implementation of Prolog, WAM, term representation, last argument overlapping, continuation passing style*

1 Introduction

BinProlog is a C-emulated Prolog engine for a subset of Prolog [13] that uses a simplified WAM [16] called BinWAM [9]. A transformation called binarization [13] maps full Prolog to binary logic programs. In this manner the BinWAM is used to implement full Prolog.

Our primary motivation was to investigate whether specializing the WAM to binary programs yields new optimization opportunities that compensate for the extra heap consumption. Because WAM's highly optimized environments were traded for a heap-only run-time system, a slow-down was expected, although it was clear from the start that, at least, the implementation will be much simpler. Our experience with BinProlog shows that its execution speed is competitive with highly optimized implementations of the WAM. This paper

[3] Available by anonymous ftp from clement.info.umoncton.ca.

compares WAM and BinWAM and presents a novel term-compression technique called last argument overlapping.

Contents. In Section 2 we present binary Prolog a subset of full Prolog used as the intermediate language for the Prolog-engine. Section 3 presents our simplified WAM, called BinWAM, and discusses the impacts of its simplified design. The term representation of the BinWAM is compared with traditional structure copying representations. Section 4 compares the actual implementation, BinProlog, to existing systems.

2 Binary Prolog

Binary Prolog is a subset of Prolog based on binary definite programs of a very simple form: one atom in the head and one atom in the body. This subset, enhanced by a labeled cut and some in-line built-ins [5], executes on a WAM engine without using WAM's environments.

Binarization: from full Prolog to binary Prolog. The natural framework of binarization covers the transformation from *definite metaprograms*, i.e., programs with metavariables in atom positions to *binary definite programs*, i.e., programs with atomic head and body. We refer to [13] for a formal definition and a study of certain semantic properties of this transformation. We give here only a few examples:

Source clause:	Binary clause:

p(X) ←
  q(X),
  r(X,Y),
  s(Y).
	p(X,Cont) ←
	  q(X, r(X, Y, s(Y, Cont))).

append([],Ys,Ys).	append([],Ys,Ys,Cont) ← true(Cont).

and(X,Y) ←
  X,
  Y.
	and(X,Y,Cont) ←
	  call(X, call(Y,Cont)).

Note that the goal true(Cont) executes the continuation Cont.

3 The BinWAM

Since binary Prolog is a subset of full Prolog every Prolog engine can be used to execute binary Prolog, although less efficiently than the BinWAM. We discuss in this section the simplified design of the BinWAM with respect to the full WAM.

3.1 Data areas

The WAM does not allocate an environment for a binary clause. The BinWAM can thus be understood as a WAM with a split stack model, where only the choice stack but not the environment stack is needed. Other data areas remain basically the same. The omission of the environment stack simplifies both the layout of choice points in the OR-stack and the handling of variables.

Choice point. In addition to the argument registers, the BinWAM stores three pointers (next clause, heap and trail) in the choice point. Binary clauses contain always an additional argument to hold the continuation. The smallest choice point contains therefore four elements. The WAM choice points require an additional pointer to the top of the environment stack and in some implementations also the arity of the predicate.

P	next clause address
H	top of the heap
TR	top of the trail
A_N	argument register N
...	...
A_1	argument register 1

BinWAM choice point

Variable handling. Since variables are allocated only on the heap, the BinWAM performs a faster trail-check that requires a single comparison.

3.2 Simplified instruction set

All instructions related to WAM's Y-variables (local variables) are omitted because the environment stack is absent. Similarily, there is no ALLOCATE, DEALLOCATE, CALL. Special list instructions are omitted as well as instructions for void variables and constants. A simpler indexing mechanism avoids some indexing instructions. Starting from this reduced instruction set some frequently occuring sequences have been combined in BinProlog to minimize interpretation overhead [12].

3.3 Goal versus environment based implementations

The BinWAM represents every goal after the first one with a separate structure. A similar approach that uses a separate stack for these goals is called goal stacking. It was proposed in the original WAM-report [16] as an alternative to environment stacking.

We illustrate the differences using the non binary clause h $\leftarrow$ g_0, g_1, ..., g_n with $n > 0$. The minimal clause is therefore of the form h $\leftarrow$ g_0, g_1. Facts and binary clauses are not discussed because they do not consume space in both approaches. The arity of goal g_i in the original Program is a_i. To outline the main points, we concentrate only on clauses that contain no functors in goals.

Dynamic space requirements. The WAM requires at least two elements in an environment to represent linking and the code continuation (CE and CP). The worst case of space consumption in the WAM occurs when all arguments of the goals are variables and when all these variables occur once in h or g_0 and once in a goal g_i, $i > 0$. In this case, every variable has to be allocated in the environment. To represent an environment, the WAM needs $\text{DYNSPACE}_{\text{WAM}}$ cells on the local stack.

$$2 \leq \text{DYNSPACE}_{\text{WAM}} \leq 2 + \sum_{i=1}^{n} a_i$$

The WAM reduces its space requirements for argument constants and variables occurring more than once within the goals g_i, $i > 0$. Furtheron the WAM is able to trim the environment under certain circumstances.

The BinWAM requires $\text{DYNSPACE}_{\text{BinWAM}}$ cells allocated on the heap. In contrast to the WAM, the BinWAM cannot reduce its space requirements. Only program transformations can be used for this purpose. No direct trimming is possible in the BinWAM, however, during garbage collection unused parts of the continuation can be reclaimed.

$$\text{DYNSPACE}_{\text{BinWAM}} = 1 + n + \sum_{i=1}^{n} a_i$$

In conclusion, the BinWAM requires ΔDYNSPACE more cells to represent an environment.

$$0 \leq \Delta\text{DYNSPACE} \leq n - 1 + \sum_{i=1}^{n} a_i$$

Code space requirements. While the dynamic space requirements of the BinWAM can be significanlty larger than those of the WAM, the size of generated code is approximately the same, favoring the BinWAM by a trifle. Since the instructions needed for head-unification, head built-ins and the initialization of the first goal are the same, we consider in the following comparison only the size of the code required for the goals g_1 to g_n.

Before executing a goal the classic WAM initializes all arguments. The BinWAM initializes all continuations directly after the head. So both machines have to execute one instruction for each argument. However, in addition to the initialization of the arguments, the WAM has to set up the environment for those variables that occur first in the head or g_0 and in a later goal $g_i, i > 0$. We illustrate this case by the following clause:

a(A_1, ... A_m) ←
    b,
    c(A_1, ... A_m).

In this case the BinWAM simply creates the continuation $c/m + 1$ using $m + 1$ instruction to initialize the arguments. The WAM, however, has to save all m

variables before executing the goal b using m get_variable-instructions. Then, after executing b the m argument registers have to be initialized with m put_value-instructions. In this (contrived) worst case, the WAM requires[4] therefore twice as much instructions as the BinWAM.

While the initialization code in the WAM is spread over n independent locations in addition to initialization in the head, the BinWAM performs initialization in a single sequence of instructions. The fact that the BinWAM initializes goals earlier might improve or deteriorate execution speed. Demoen and Mariën [5] discuss these impacts in detail and show how source-to-source transformations alleviate potential problems.

3.4 Stack versus heap allocation

Due to the absence of an auxiliary stack, be it an environment or a goal stack, heap consumption is increased in the BinWAM. Simplicity in the BinWAM's design is achieved by higher memory requirements similar to implementations of functional languages based on continuation passing style [15].

Note that the situation in Prolog is different from the situation in functional languages. While functional languages reclaim all memory allocated by doing garbage collection, the BinWAM uses still an OR-stack for backtracking. I.e. the AND-control is implemented with the heap, OR-control is still stack based. If a goal fails back, all memory is reclaimed by backtracking as in the WAM. In particular, calls to findall/3 produce as much AND-garbage as any other Prolog implementation. Only the space needed to represent the list of solutions remains on the heap. If a goal is known to be determinate, findall/3 can be used to reclaim all volatile terms needed to execute the goal. In Sect. 3.8 a copy-once implementation of findall/3 is described.

For these reasons it is evident that the BinWAM consumes similar or less heap space to be reclaimed by garbage collection than a machine for functional languages. Implementations like SML-NJ have shown that even in a functional language the overheads of increased heap consumption are manageable [1].

3.5 Term representation

As the BinWAM consumes more heap than the traditional WAM a compact representation of heap terms is of particular interest for the BinWAM. Similar to the simplification of the instruction set, the term representation has been simplified in the BinWAM. This representation is also of interest for other structure copying Prolog systems.

---

[4] This comparison applies only to the basic WAM as defined in [16] and to the BinWAM as a strict subset of the WAM. Some WAM implementations fold e.g. get_variable-instructions and calls. Also BinProlog folds instructions.

Tag-on-pointer representation. Current structure copying Prolog machines use several pointer types to represent terms. Usually, at least three pointer types are used. We call such a representation *tag-on-pointer* representation.

1. reference or variable
2. pointer to a structure
3. pointer to a list, as an optimization for structure ./2

The specialized pointer type for lists is not strictly needed. Yet, most machines implement this optimization. Usually, references are tagged by word alignment i.e. the lower bits of a reference are zero. The other pointer tags are encoded in the lower bits. When creating a pointer, a dedicated tag is added to the address.

We note that the coding effort for procedures that manipulate terms like copy_term/2, assert/1 or garbage collection increases with the number of different pointer types. In the classical WAM the situation is even worse because different pointer types can point to the same memory cell (lists and references).

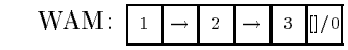
Tag-on-data representation. The BinWAM uses a *single* pointer type. Pointers are word aligned so no extra tagging is needed.

1. reference, variable, or pointer to structure

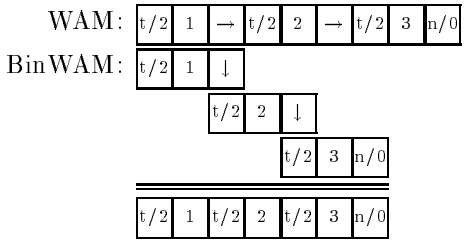
Thus, Prolog variables and C pointers have the same representation. The two other tagged data types of BinProlog are integers, and functors. Symbolic constants are implemented as functors of arity 0. Like integers, constants are either kept in registers or directly copied with no extra reference. 64 bit floating point numbers are mapped to a functor of arity 3. Unification and the basic instruction set is therefore not affected by these additional data types. Lists are treated as any other structure of arity 2.

Variable binding. When binding a variable to another dereferenced non variable term the BinWAM has to perform an additional operation to keep reference chains short. While binding corresponds usually to a single assignment, the BinWAM has to test whether the term is a structure or not. In the former case the reference is used, in the latter the value. Compared to common WAM-implementation, the more complex binding scheme does not deteriorate efficiency of unification because the BinWAM omits a comparison during trailing. In exchange, an additional comparison for variable binding is performed.

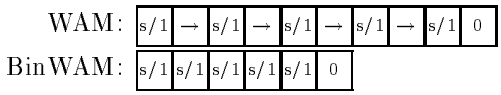
Last argument overlapping. While the BinWAM does not provide a specialized representation for lists, it allows a more general optimized representation useful for any structure. References to structures in the last argument of another structure can be replaced by the structure itself. The WAM represents a list of n elements by $2n$ memory cells. We illustrate the WAM representation with the list [1,2,3].



If another structure of arity two is used, space consumption increases to $3n$ cells. The term t(1,t(2,t(3,n))) is thus represented as below. The BinWAM does not require a pointer in the last argument of t/2 if the subsequent structure can be found directly after in memory. In general, n elements require $2n + 1$ memory cells. The name of the structure has no impact on space consumption. It is in particular not necessary to use always the same functor.



In the best case last argument overlapping halfs memory consumption. E.g., the term s^n requires $n + 1$ memory cells, while the WAM uses always $2n$ memory cells. WAM and BinWAM represent the term $s^5 = s(s(s(s(s(0)))))$ as follows:



Note that the compact representation is only possible if the structures are created in the right order. If last argument overlapping is not possible during creation, procedures that copy (e.g. copy_term/2, assert/1) or reorganize terms (garbage collection) should generate more compact representations.

As a positive impact on programming style our new data representation makes the usage of dotted pairs (i.e. ugly constructs like [a|b]) for efficiency reasons unnecessary.

3.6 Adaptations for last argument overlapping

Several built-ins had to be modified in order to read the new compact representation. Instructions creating structures were modified to create overlapping structures if possible.

General unification and built-ins. For general unification, care has to be taken when unifying last arguments of structures. In this case instead of the value of the last arguments, references to them are unified. In a similar manner the built-in true/1 used to call continuations has been adapted. The runtime overhead of these modifications is empirically negligible. Built-ins like univ/2, name/2 and maplist/3 are changed to create overlapping lists.

GET_STRUCTURE-**instruction.** During head unification in write mode a new structure is created on the heap. If the variable to be bound to the structure is found as the last element on the heap we overwrite the variable with the new functor. No reference is created. The GET_STRUCTURE instruction is modified as follows.

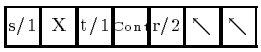
```
get_structure An:
    DEREF(An);
    if(VAR(An))
        {   trail(An);
            if (An + 1 == H)
                H --;
            *H = functor;
            ...
        }
```

PUT_STRUCTURE-instruction. This instruction is used to create structures that are passed further to the next goal. In particular, continuations are created using PUT_STRUCTURE-instructions. When a single structure is created, there is no possibility for last argument overlapping since the last argument will point to a previously created structure. Only garbage collection or copy_term will be able to reorder such structures.

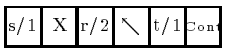
Nested structures are created in the WAM using a bottom-up compilation scheme. First, structures are created that refer to already initialized terms only. The reason for this compilation scheme is that the WAM instructions for initializing the arguments of a structure (WRITE-instructions) use the heap pointer as an implicit argument. As a consequence, the WAM creates only backward pointers, depicted as '↖'. Last argument overlapping requires a top-down compilation for those structures occurring in the last argument of another structure. All other structures are created in the same manner as in the traditional WAM. We illustrate this point by the following simple program.

Source clause:	Binary clause:
p(X) ← q, r(s(X)), t.	p(X, Cont) ← q(r(s(X), t(Cont))).

The two last goals in the clause above are encoded as terms. The last goal t appears in the last argument of r. Last argument overlapping is therefore applicable. In the usual WAM compilation scheme both s/1 and t/1 are created prior to r/2. Last argument overlapping allows the creation of t/1 immediately after r/2. In this manner both a memory cell and an instruction is saved.




```
put_structure s/1,var(3)
write_value put,var(1)
put_structure t/1,var(4)
write_value put,var(2)
put_structure r/2,var(1)
write_value put,var(3)
write_value put,var(4)
execute q,1
```




```
put_structure s/1,var(3)
write_value put,var(1)
put_structure r/2,var(1)
write_value put,var(3)
put_structure t/1,var(4)
write_value put,var(2)
execute q,1
```

Currently the code above listed with BinProlog's intermediate WAM-code listing tool asm/0 is generated by an experimental separate pass after WAM-code generation. While our first measurements with this modified goal compilation scheme indicate small savings (4%) in runtime and dynamic memory, our compilation scheme reduces the size of larger clauses. Since the continuation argument is currently always the last argument, continuations are only compacted statically if a clause contains at least three real goals.

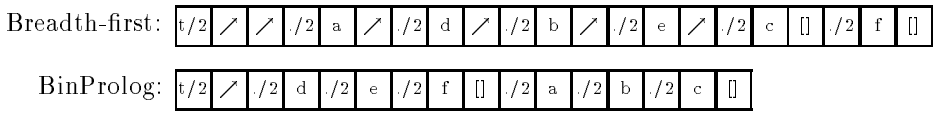
Most built-in predicates occurring directly after the head are compiled in-line and therefore do not count as a real goal. Further the most heavily used clauses contain only one or two goals. The biggest savings so far have been observed in clauses that are not frequently used, but that consume a lot of space for passing structures further on.

In ongoing work we implement a mixture between top-down compilation for structures occurring in the last argument and bottom-up compilation for other terms. For all structures that occur only as a last argument PUT_STRUCTURE-instructions are replaced by WRITE_CONSTANT-instructions. This compilation scheme for PUT-instructions does not require any modification or extension to the existing instruction set. In the case of a native code implementation, A. Marien's [7] method is of interest.

3.7 **Cheney-style compressing** copy_term

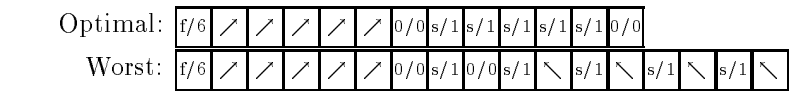
The classical algorithm of Cheney [3] copies a term in a breadth-first manner using forwarding pointers for already moved cells. This algorithm can be directly taken to implement copy_term. The single difference is that the forwarding pointers that destroy the original term, have to be trailed on a value trail. As long as no sharing of identical terms is present, the algorithm produces only forward references, depicted '↗'. We modified this algorithm to enforce last argument overlapping as follows. When a structure is copied, its last argument and all its subterms that are again found as a last argument are copied first. In this manner, most possibilities for last argument overlapping are discovered. While the classical method is a pure breadth-first algorithm, our adaptation introduces a depth-first component. In the frequent case of a list of atomic cells our algorithm creates a fully contiguous *vector-like* object.

We illustrate BinProlog's algorithm with the term t([a,b,c],[d,e,f]). Breadth-first copying leads to the first memory layout, while the second layout is the result of our algorithm in BinProlog. Note that the first algorithm slices up all linear chains so that no last argument overlapping is possible.



As long as there are no shared subterms, our algorithm is able to exploit all overlaps. However, in the presence of shared subterms, our algorithm may expand a compact term up to 50%.

Worst case. Consider the term f(s^1, s^2, ..., s^{n-1}, s^n, 0) where all common subterms are shared. The optimal representation requires $2n+3$ memory cells for all $n > 0 : n+2$ for the functor f/$n+1$ and $n+1$ for the term s^n. In this case, all terms s^i $0 < i < n$ are represented as pointers to subterms of s^n. Every structure s/1 up to one can be accessed twice: once from the structure f/$n+1$ and once from another structure s/1. In order to perform optimal copying, the topmost structure representing s^n should be copied first. However, a left-to-right scan first copies s^1, then s^2 etc. All last argument overlaps are hence destroyed. The resulting copy requires $(n+2)+2n = 3n+2$ cells. The following picture illustrates the case $n = 5$.



A complete visit of the term is needed to determine all possible overlaps. Modifying the simple left-to-right scanning alone only reduces or changes the worst case. An optimal copy_term algorithm requires a pass prior to copying for identifying all structures occurring as the last argument of another structure. Since the worst case occurs only for heavily shared structures and since a separate pass would approximately double execution time we have currently only integrated the single pass algorithm into BinProlog.

Last argument overlapping versus CDR-coding. Techniques for eliminating pointers in data structures of symbolic programming languages are well known. In particular, CDR-coding has been used in LISP-systems. In such systems, a CONS-cell does not contain the CDR-pointer to the next CONS-cell. Instead, a special tag in the CAR indicates that the next CONS-cell is located at the place of the CDR. CDR-coding has been investigated for Prolog in [6]. The technique is not used in major modern systems ever since because it was shown useful only in hardware implementations. Last argument overlapping is a more general optimization that applies to any structure and requires no special tags. In particular, continuations profit from our optimization.

3.8 Fast 'copy-once' findall

Some key extensions to Prolog take advantage of our compressed term representation and fast copy_term algorithm. We implemented findall/3 and findall/4 using a *copy once* technique. In traditional implementations every solution found is copied twice: first into the database, then back into a list on the heap. With the technique of *heap-lifting* [10] findall reserves in advance space on the heap for the list of solutions. Currently, 1/8 of the available heap space is reserved; the heap is split. The upper part of the heap is used for executing the goal. Every solution is copied into the lower part. Therefore, the list of solutions is constructed during the execution of the goal. After the exhaustive search, the heap is set after the copy of the last answer. An internal stack keeps track of split areas for recursive uses of findall. The Prolog part of findall appears as a failure driven loop. This has the advantage that it can be partially evaluated (for example in the case when the goal is known at compile time) and other findall-like predicates can be built using the same primitives. We refer the reader to [10] for the C-sources of findall. Note that the statically fixed ratio between answer-space and computation-space can be overridden by giving the desired size of the answer space as an optional argument to a given instance of findall. This technique can be applied in principle to any WAM-based implementation.

4 Performance evaluation of the BinWAM

Figure 1 compares the performance of BinProlog 2.10 with C-emulated and native code SICStus Prolog 2.1 and Quintus Prolog 3.1.1, all running on a SPARCstation ELC. The SICStus compiler is bootstrapped so that built-in predicates are compiled to native code (the fastest possible configuration). Timing is without garbage collection, as returned by statistics(runtime,_). The programs are available in the directory progs of the BinProlog distribution. NREV is the well-known naïve reverse benchmark, CNREV is obtained from NREV by replacing the list constructor with a different functor of arity two, the CHAT-Parser is the unmodified version from the Berkeley Benchmark, FIBO(16)x50 is the recursive Fibonacci predicate, Q8 is an 8-Queens program, PUZZLE is a definite clause solution to a logic problem, LKNIGHT(5) is a complete knight tour, PERMS(8) is a permutation generator, DET-P(8) is a deterministic all permutation program, FINDALL-P(8) is a findall-based all-permutations program, BFIRST-M is a naïve breadth-first Prolog meta-interpreter, BOOTSTRAP is our compiler compiling itself, CAL is an arithmetic intensive calendar program, DIFFEREN is a symbolic differentiation program from the Warren benchmarks and CHOICE is a backtracking intensive program from the ECRC benchmark. Although shallow-backtracking is faster in C-emulated SICStus due to the optimization described in [2] the overall performance for this benchmark is better in BinProlog due in part to its smaller and unlinked choice points. This materializes as a 7 Mbytes (SICStus) versus 5 Mbytes (BinProlog) OR-stack consumption.

Figure 2 shows the effect of various levels of instruction compression (I) and term compression (T) on speed, measured on a SPARCstation 10-20.

System *version*	SICStus 2.1_6 C emulator	BinProlog 2.10 C emulator	Quintus 3.1.1 asm emulator	SICStus 2.1_6 native
NREV	139 klips	223 klips	479 klips	566 klips
CNREV	120 klips	223 klips	130 klips	466 klips
CHAT-Parser	1.540 s	1.283 s	0.900 s	0.660 s
FIBO(16)x50	4.030 s	3.300 s	2.100 s	1.320 s
Q8	0.379 s	0.266 s	0.167 s	0.119 s
PUZZLE	0.100 s	0.083 s	0.050 s	0.050 s
LKNIGHT(5)	132 s	93 s	77 s	43 s
PERMS(8)	1.189 s	0.717 s	0.450 s	0.339 s
DET-P(8)	2.430 s	1.966 s	1.450 s	0.650 s
FINDALL-P(8)	8.500 s	2.867 s	31.333 s	7.450 s
BFIRST-M	0.490 s	0.217 s	1.233 s	0.360 s
BOOTSTRAP	20.550 s	20.683 s	23.517 s	12.260 s
CAL	1.381 s	0.950 s	0.566 s	0.410 s
DIFFEREN	1.380 s	0.950 s	0.566 s	0.399 s
CHOICE	5.899 s	3.617 s	7.717 s	1.690 s

Fig. 1. Speed of BinProlog versus environment stack implementations (Sparc ELC).

Bmark/Compiler	No	T	I2,T	I3	I3,T
NREV	229 kl	238 kl	326 kl	408 kl	436 kl
CHAT-Parser	0.81 s	0.81 s	0.72 s	0.69 s	0.70 s
PERMS(8)	0.49 s	0.48 s	0.41 s	0.36 s	0.37 s
DET-P(8)	1.46 s	1.47 s	1.19 s	0.93 s	1.02 s
FINDALL-P(8)	1.44 s	1.42 s	1.35 s	1.36 s	1.30 s
CHOICE	2.06 s	2.06 s	1.98 s	1.92 s	1.96 s
QSORT	0.72 s	0.67 s	0.61 s	0.61 s	0.55 s

Fig. 2. Impact of instruction and term compression on execution speed (Sparc 10/20).

Note that although term-compression implies a (small) amount of extra work, it actually accelerates some of the benchmarks due to the reduced memory bandwidth. Clearly AND-intensive programs like NREV, QSORT DET-P(8) benefit most from term compression.

Heap-savings (Figure 3) are also more noticeable for programs which do most of the structure construction in the head like NREV and CNREV. We expect useful savings on all programs when the optimization for PUT operations will be integrated in the compiler. For comparison we have given also heap consumption for SICStus.

Prog/Heap with no gc	No	T	Sicstus 2.1
NREV	547K	369K	361K
CNREV	547K	369K	541K
DET-P(8)	4788K	3387K	2329K
FINDALL-P(8)	4517K	3387K	4193K
QSORT	520K	380K	305K

Fig. 3. Impact of term compression on heap consumption.

Compiler version	No	T	I2,T	I3	I3,T
Code size in bytes	47460	47484	49356	51124	51244

Fig. 4. Impact of optimizations on the emulator's code size.

However, this is also true overall in the case of an OR-intensive permutation program like FINDALL-P(8), due to BinProlog's 'heap-lifting' (see [10]) based implementation of findall/3 which uses compressing term copying and manages to put on the heap directly the computed answer with no trace of the computation that generates it. In this case, BinProlog creates at the end a better heap-representation than SICStus 2.1 without garbage collection although SICStus 2.1 uses dedicated list tags and therefore list-cells of size 2. BinProlog achieves for FINDALL-P(8) an overall heap-consumption of 3387K which is reasonably close to what SICStus (with list-cells of size 2) gets after garbage collection for FINDALL-P(8) i.e. 2904K. The CNREV program (NREV with another constructor is used instead of '.'/2) helps to compare dedicated list-instructions to our general purpose term-compression scheme.

Emulator code size. Figure 4 shows the impact of our combined term and instruction compression optimizations on the emulator's code size. The increase in size is still less than 4K when comparing the original version (No) to the most optimized one (I3,T). Version I3,T uses term compression compresses up to three instructions [12].

5 Future work

Future work can be divided into the following two parts: source-to-source transformations and improvements on the BinWAM level.

Source-to-source transformations. Binary Prolog encodes the state of the AND-control in terms, visible at the source level. Optimizations usually per-

formed on a lower level can now be expressed at the source level. Source-to-source transformations that reduce the heap-consumption [4, 8] will be integrated into the BinProlog systems. In particular, an advanced technique, EBC-transformations, was developed that is able to perform interprocedural register allocation by mere source-to-source transformations in programs with difference lists or accumulators [8]. EBC-transformations transform a given binary Prolog program into another binary Prolog program. These optimizations cannot be observed directly on a traditional WAM, except in the case when the WAM is used to execute the derived binary programs. In this case, only WAM instructions that are present in the BinWAM are used. Comparable optimizations using the environment stack would require low level modifications to the WAM.

BinWAM-optimizations. BinProlog is still an emulator-based system. Techniques as developed for Aquarius [14] are based on a machine (BAM) that is below the level of the WAM. More refined optimizations are possible on the BAM-level. Similar techniques could be adopted for the BinWAM.

6 Conclusion

The simplicity of BinProlog's basic instruction set and data-representation allows to apply low level optimizations easier than on standard WAM. However, our tag-on-data representation and its term-compression technique described in this paper can also improve environment-based WAM implementations.

Combined with the previously reported [11] virtualization of demo-predicates at WAM-level that largely eliminate the overhead of metaprogramming introduced by binarization and the simplified memory management approach described in [10], BinProlog's engine BinWAM can now be considered a realistic alternative to standard WAM.

Acknowledgments. We are grateful for support from NSERC (research grant OGP0107411 and equipment grant), the FESR of the Université de Moncton and the Jubiläumsfond der Stadt Wien. Suggestions and comments from a large number of BinProlog users helped improving the design of our logic engine and clarifying the content of this paper. Special thanks go to Bart Demoen and Mats Carlsson who shared with the first author their deep knowledge of WAM-internals at a point when his limited implementation experience would otherwise have a negative impact on the BinWAM's initial design. Professor Brockhaus has supported our work and provided helpful comments.

References

1. A. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.
2. M. Carlsson. *Design and Implementation of an OR-Parallel Prolog Engine.* Phd thesis, SICS, 1990.
3. C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of ACM*, 11(13):677–678, Nov. 1970.
4. B. Demoen. On the Transformation of a Prolog program to a more efficient Binary program. Technical Report 130, K.U.Leuven, Dec. 1990.
5. B. Demoen and A. Mariën. Implementation of Prolog as binary definite Programs. In A. Voronkov, editor, *Logic Programming, RCLP Proceedings*, number 592 in Lecture Notes in Artificial Intelligence, pages 165–176, Berlin, Heidelberg, 1992. Springer-Verlag.
6. T. Dobry. *A High Performance Architecture for Prolog.* Phd thesis, University of California at Berkley, 1987.
7. A. Mariën. An Optimal Intermediate Code for Structure Creation in a WAM-based Prolog Implementation. Technical report, K.U.Leuven, May 1988.
8. U. Neumerkel. A transformation based on the equality between terms. In *Logic Program Synthesis and Transformation, LOPSTR 1993.* Springer-Verlag, 1993.
9. P. Tarau. A Simplified Abstract Machine for the execution of Binary Metaprograms. In *Proceedings of the Logic Programming Conference'91*, pages 119–128. ICOT, Tokyo, 7 1991.
10. P. Tarau. Ecological Memory Managment in a Continuation Passing Prolog Engine. In Y. Bekkers and J. Cohen, editors, *Memory Management International Workshop IWMM 92 Proceedings*, number 637 in Lecture Notes in Computer Science, pages 344–356. Springer, Sept. 1992.
11. 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.
12. P. Tarau. Wam-optimizations in BinProlog: towards a realistic continuation passing prolog engine. Technical Report 92-3, Dept. d'Informatique, Université de Moncton, July 1992. available by ftp from clement.info.umoncton.ca.
13. P. Tarau and M. Boyer. Elementary Logic Programs. In P. Deransart and J. Małuszyński, editors, *Proceedings of Programming Language Implementation and Logic Programming*, number 456 in Lecture Notes in Computer Science, pages 159–173. Springer, Aug. 1990.
14. P. Van Roy. *Can Logic programming Execute as Fast as Imperative Programming.* Phd thesis, University of California at Berkley, 1990.
15. M. Wand. Continuation-based program transformation strategies. *Journal of the Association for Computing Machinery*, 27(1):164–180, 1980.
16. D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, Oct. 1983.

This article was processed using the LaTeX macro package with LLNCS style