Compact representation of terms and instructions in the BinWAM

Paul Tarau
Département d'Informatique
Université de Moncton
Moncton, N.B. Canada, E1A 3E9,
and Simon Fraser University
tarau@cs.sfu.ca

Ulrich Neumerkel
Institut für Computersprachen
Technische Universität Wien
E185/1 Argentinierstr.8,
A-1040 Wien, Austria
ulrich@mips.complang.tuwien.ac.at

Abstract

The BinWAM is a simplified WAM engine that uses *binary logic programs* as an intermediary language. Binary logic programs are a subset of logic programs in continuation passing style. The BinWAM is essentially a heap-based WAM without environments and with a new and more regular term representation. The BinProlog system[1] has been entirely built around a C-emulated BinWAM. While BinProlog's performance is comparable with the best commercial Prolog emulators, its implementation is significantly simpler. The BinWAM uses a new low-level data representation and a new technique called last argument overlapping. We describe the adaptations for this new technique. A Cheney-style copy_term algorithm taking advantage of our term representations is used for the fast 'copy once' implementation of findall. 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 compilation*

1 Introduction

BinProlog is a C-emulated Prolog engine based on a program transformation introduced in [10]. It replaces the WAM [13] by a more compact continuation passing logic engine [8]. The transformation called binarization maps full Prolog to binary logic programs.

The BinProlog project has actually started before the [10] paper on the theoretical aspects of binarization with a small 6-instruction binary engine tested on top of ALS-Prolog's WAM.

[1] available by anonymous ftp from clement.info.umoncton.ca.

The main idea was to investigate whether specializing the WAM to binary programs yields new optimization opportunities that compensate for the extra heap consumption. As conventional WAM's highly optimized environments were given away for a heap-only run-time system, an important slow-down was expected, although it was clear from the start that, at least, the implementation will be much simpler. It turned out, after building the BinProlog system around our engine, that it is fairly equivalent if not better than well engineered implementations of the standard WAM. This paper explains in detail the reasons of the BinWAM's efficiency and focuses on a novel term-compression technique which addresses its main weakness: the higher heap consumption.

Contents. In Section 2 we present binary Prolog a subset of full Prolog that is 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 discusses BinProlog, the actual implementation of the BinWAM, and compares its performance to existing WAM implementations.

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 a set of in-line built-ins ([5]), is executed on the WAM engine without using environments. The first step towards a practical implementation of full Prolog with Binary Prolog is the efficient and simple transformation called *binarization* which preserves indexing [10].

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 [10] 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)).

Remark that the first step of the transformation wraps metavariables inside a new predicate true/1. The second step adds continuations as last arguments of each predicate and a new predicate true/1 to deal with unit clauses. The final

result contains two new predicates true/1 and call/2 [2]. Although we can add clauses that describe how true/1 and call/2 work on the set of all the functors occurring in the program, in an emulator like BinProlog it is more efficient to treat them as built-in predicates see [8]. BinProlog performs their execution inline while affording to look up dynamically a hash-table to transform the term to which a meta-variable points, to its corresponding predicate-entry. As this happens only for facts in the original program, it has relatively little impact on performance [3].

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 a specialized engine like the BinWAM. We discuss in this sections the simplified design of the BinWAM with respect to a full WAM engine.

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, three pointers (next clause, heap and trail) are stored by the BinWAM. Since binary clauses contain the continuation as an additional argument, the arity of binary clauses is one argument larger than the corresponding Prolog clause. 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.

[2] Which can be seen as continuation passing variants of Prolog's true/0 and call/1

[3] Our instruction-level profiling in BinProlog 2.10 indicates that the engine spends on average between 5-7 % of its execution time in setting up a goal from a continuation predicate.

3.2 Simplified instruction set

All instructions related to the handling of WAM Y-variables are not needed because the environment stack is absent. ALLOCATE, DEALLOCATE, CALL are also absent for the same reason. However, the BinWAM goes further with the reduction of the basic instruction set. First, specialized list instructions have been omitted, initially to keep the design simple and later because term-compression has made them less useful. Second, special instructions for void variables or constants have been omitted initially to keep the design simple and later because profiling indicated that their impact was empirically unimportant. Third, in contrast to conventional WAM's emphasis on tag analysis the BinWAM implements a simplified functor-based indexing mechanism made possible by a simpler low-level data representation. The reduced instruction set allows us to add compressed versions of the most frequent instruction sequences while increasing the size of the emulator only by a small amount (3K). This improves execution speed close to that of threaded-code emulators.

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 [13] as an alternative to environment stacking.

First, dynamic space requirements are discussed, then the code to be generated for both approaches is compared. We illustrate the differences using the non binary clause h $\leftarrow$ g_0, g_1, ..., g_n with $n > 0$. 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 functor 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 in the WAM occurs when all arguments of the goals are variables and when these variables occur once in h or g_0 and once in a goal g_i, $i > 0$. The WAM is able to reduce its space requirements for argument constants and variables occurring more than once in g_i, $i > 0$. While the BinWAM requires more space for most environments, deallocation of already used and determinate frames is for free[4]. The WAM needs environment trimming to reclaim space while a clause is active. The variables DYNSPACE below describe the space requirements for both machines.

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

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

[4]The BinWAM requires garbage collection or failure for this to actually happen while a goal stacking machine would reclaim space immediately.

Code generation. Before executing g_i, $i > 0$, the WAM has to ensure initialization of all arguments. The layout of WAM code is:

- head-unification, head built-ins, and initialization for the first goal
- instructions initialize the environment
- call g_0
- a_1 instructions initialize g_1
- call g_1
  ...
- a_n instructions initialize g_i and save unsafe variables
- execute g_n

In contrast the layout in the BinWAM is as follows:

- head-unification, head built-ins, and initialization for the first goal
- $\text{STATSPACE}_{\text{BinWAM}}$ put- and write-instructions
- execute g_0

Since the instructions needed for head-unification, head built-ins and the initialization of the first goal are the same, we consider only the code size for the code required for the goals g_1 to g_n. BPERM refers to the number of permanent variables in the WAM that only g_1 to g_n.

$$\text{STATSPACE}_{\text{WAM}} = \text{DYNSPACE}_{\text{WAM}} - \text{BPERM} + n + \sum_{i=1}^{n} a_i$$

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

$$\text{STATSPACE}_{\text{WAM}} - \text{STATSPACE}_{\text{BinWAM}} = \text{DYNSPACE}_{\text{WAM}} - \text{BPERM} - 2 \geq 0$$

The instruction overhead of the WAM corresponds therefore to the number of permanent WAM variables that are already initialized before the first goal. 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.

Native code generation. Initialization code does not contain conditional instructions (like unify etc.) and therefore constitutes a basic block. The BinWAM in native code has therefore a single long basic block to execute that comprises the code for initializing all goals. The WAM comprises $n + 1$ smaller basic blocks, often called chunks. On current modern processors exploiting ILP (instruction level parallelism) large basic blocks improve instruction scheduling and reduce branch penalty.

3.4 Stack versus heap allocation

Due to the absence of an auxiliary stack be it an environment or goal stack heap consumption is increased in the BinWAM. Therefore, simplicity in its design is paid by higher memory requirements. In the area of functional languages similar design decisions were chosen in implementations based on continuation passing style [7, 12].

Note that the situation in Prolog is different from the situation in functional languages. While functional languages reclaim all memory allocated by 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. 4 a copy-once implementation of findall/3 is described which is so efficient that it is practical to use for this purpose.

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 he BinWAM consumes more heap than the traditional WAM with an environment or a goal stack. A compact representation of heap terms is therefore of particular interest for the BinWAM.

Similar to the simplification of the instruction set, term representation has been simplified in the BinWAM. The term representation presented here is of general interest for other structure copying Prolog implementations.

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 to represent variables and variable sharing
2. pointer to a structure
3. pointer to a list, as an optimization for structure ./2

A 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 has to be added to the address.

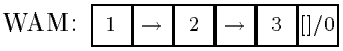
We note that the coding effort for procedures that manipulate terms like copy_term/2, assert/1 or garbage collection increases with the number of different pointer types. In the classical WAM the situation is even worse because different pointer types can point to the same memory cell (lists and references).

Tag-on-data representation. The BinWAM uses a single pointer type with tag=00 on 32 bit machines and tag=000 on 64 bit machines. Pointers are therefore always word aligned. The only other tagged data types of BinProlog are integers, and functors. Lists are treated as any other structure of arity 2. Symbolic constants are implemented as functors of arity 0. 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. Our general term-compression technique discussed in the next sections made our list-representation as efficient as a special list data type.

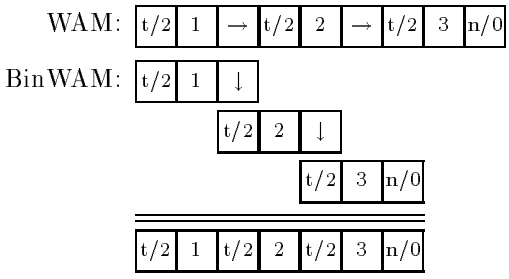
Dereferencing. Dereferencing is performed similar to the WAM with two variables containing the pointer and its value. After dereferencing, the functor of a structure is present in the value register. If the functor's value is of interest, as in the case of indexing, no further loading is required.

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. The more complex binding scheme has no impact on the efficiency of unification in the BinWAM because the BinWAM omits a comparison during trailing. In exchange, an additional comparison for variable binding is performed.

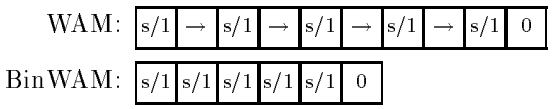
Last argument overlapping. While the BinWAM does not provide a specialized representation for lists, it allows a more general optimized representation that can be used for any structure. References to structures in the last argument of another structure can be replaced by the structure itself. The WAM 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.



Last argument overlapping is possible for any structure. 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. Thus the term s^5 is represented in WAM and BinWAM as follows:



As a positive impact to programming style our new data representation makes the 'efficient' usage of dotted pairs obsolete.

3.6 Adaptations for last argument overlapping

Several built-ins had do be modified in order to read the new compact representation. Instructions creating structures were modified to create overlapping structures if possible. If last argument overlapping was not possible during creation, procedures that copy or reorganize terms should generate more compact representations.

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 overhead of these modifications is empirically un-noticeable.

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 can over write the variable with the new functor. No reference needs to be created. The GET_STRUCTURE instruction is therefore 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.

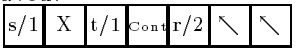
Source clause:

```
p(X) ←
    q,
    r(s(X)),
    t.
```

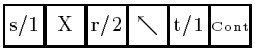
Binary clause:

```
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 before the creation of r/2. Last argument overlapping allows to create 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 asm/0 is generated by an experimental separate pass after WAM-code generation. While our first measurements with the emulator indicate very small savings (less than 5%) in runtime and dynamic memory, our compilation scheme reduces at least the size of larger clauses. Note that 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. PUT_STRUCTURE-instructions can be safely replaced by WRITE_CONSTANT-instructions for all structures that occur only as a last argument. It should be underlined that our new compilation scheme for PUT-instructions does not require any modification or extension to the existing instruction set.

3.7 A Cheney-style iterative compressing copy_term

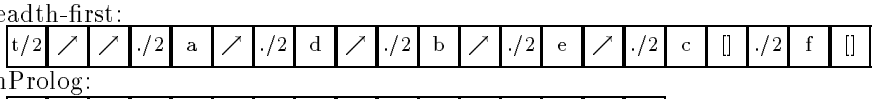
The classical algorithm of Cheney [3] copies a term in a breadth-first manner using forwarding pointers for already moved cells. This algorithm can be directly taken to implement copy_term. The single difference is that the forwarding

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 by '↗'.

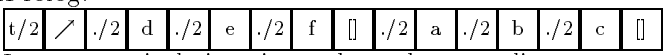
We modified this algorithm to enforce last argument overlapping as follows. When a structure is copied, its last argument and all its subterms that are again found as a last argument are copied first. In this manner, most possibilities for last argument overlapping are exploited. While the classical method is a pure breadth-first algorithm, our adaptation introduces a depth-first component. It is known from the LISP-literature that references to structures that have been copied depth-first lead to higher locality of memory references than structures copied breadth-first.

We illustrate BinProlog's algorithm with the term t([a,b,c],[d,e,f]). A breadth-first algorithm 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.

Breadth-first:



BinProlog:



In most typical situations where there are linear structures the copy of a term is more compact than the original term. It should be noted, however that in the worst case our algorithm may expand an existing term when shared subterms are present. The following example shows this worst case.

Consider the term f(s^1, s^2, ..., s^{n-1}, s^n, 0) where all common subterms are shared. This term can be represented with $2n+3$ 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. Unfortunately, copying this term destroys all last argument overlapping. The resulting copy requires therefore $(n+2)+2n = 3n+2$ cells.

An optimal copy_term algorithm requires a separate pass to detect all structures where last argument overlapping can be applied while also correctly dealing with shared subterms. Since the worst case seems to be rare and a separate pass would approximately double execution time we have 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 proposed for LISP-systems. In such systems, a cons cell is represented by omitting the pointer to the next cons cell. The elements of the lists contain therefore a special tag. In contrast last argument overlapping is a more general optimization that applies to any structures 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* [9] 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 half of the heap is used for executing the goal. Every solution is copied into the lower half. 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 [9] for the C-sources of findall.

4 The BinProlog implementation

In this section we describe implementation aspects that are specific to the emulated version of BinProlog.

4.1 Compact instruction set

In version 2.10 the BinProlog C-emulator uses 123 instructions which are divided as follows:

20 elementary WAM instructions

3 instructions to handle cut

17 specialized unification instructions

3 specialized execute instructions

26 arithmetical instructions

54 other built-in instructions

The largest part of the instruction set consists of built-in predicates. 43 instructions implement the core of the BinWAM. From these 43 core-instructions, 20 instructions where derived by instruction folding.

BinProlog uses word-only addressing with shifting and mask operations explicitly performed by a set of C-macros. This turned out to be highly portable and increased performance by 5-10% performance even on byte-addressing machines. Basic instructions are 1 or 2 words long while their folded variants are of length 1, 2 or 3. Note that by folding for instance 2-unify instructions they will still fit in one word. This contributes to the reduction of memory accesses and has visible effects on performances and code-size.

4.2 Reducing the interpretation overhead

Instruction-compression. It happens very often that a sequence of consecutive instructions share some WAM state information. E.g., two consecutive unify instructions have the same mode as they correspond to arguments of

the same structure. Moreover, due to our very simple instruction set, some instructions have only a few possible other instructions that can follow them, therefore relatively long high-frequency sequences tend to form easier in the BinWAM than on larger instruction-set engines like conventional WAMs. E.g., EXECUTE is specialized to KEXEC_SWITCH and EXEC_JUMP_IF. By compressing UNIFY instructions and their WRITE-mode specializations, we obtain the following 8 new instructions:

UNIFY_VARIABLE_VARIABLE,	WRITE_VARIABLE_VARIABLE,
UNIFY_VALUE_VALUE,	WRITE_VALUE_VALUE,
UNIFY_VARIABLE_VALUE,	WRITE_VARIABLE_VALUE,
UNIFY_VALUE_VARIABLE,	WRITE_VALUE_VARIABLE

BinProlog 1.71 and later also integrates the preceding GET_STRUCTURE instruction into the double UNIFY instructions and PUT_STRUCTURE into the double WRITE instructions like for instance GET_UNIFY_VAR_VAR. This gives another 8 instructions but it covers a large majority of uses of GET_STRUCTURE and PUT_STRUCTURE. As a consequence, in the frequent case of structures of arity=2 (lists included), mode-related IF-logic is also eliminated. This gives, in the case of the binarized version of the recursive clause of append/3 the following code:

```
append([A|Xs],Ys,[A|Zs],Cont) ←
    append(Xs,Ys,Zs,Cont).

TRUST_ME_ELSE */4,                  % keeps also the arity = 4
GET_UNIFY_VAR_VAR X1, ./2, X5, A1
GET_UNIFY_VAL_VAR X3, ./2, X5, A3
EXEC_JUMP_IF append/4               % address of append/4
```

Clearly, the quadratic explosion due to the elaborate case analysis (safe versus unsafe, x-variable versus y-variable) makes the job of advanced instruction compression tedious in the case of standard WAM. We found out that simplifying the unification instructions of the BinWAM allows for very 'general-purpose' instruction compression, in contrast to conventional WAMs which often limit this kind of optimization to lists[5]. As a consequence, arithmetic expressions or programs manipulating binary trees will benefit from our compression strategy while this may not be the case with conventional WAMs, unless they duplicate their (already) baroque list-instruction optimizations for arbitrary structures. *To summarize, in a drastically simplified engine like the BinWAM, a few compressed instructions ar enough to are able to hit a statistically relevant part of the code* [6].

Note that instruction compression is usually applied inside a single procedure. As BinProlog uses only the EXECUTE instruction instead CALL, ALLOCATE, DEALLOCATE, EXECUTE, PROCEED we can afford to do instruction compression across procedure boundaries with very little increase in code size due to relatively few possibilities for combination. As an interesting development, at the global level, knowledge about possible continuations can

[5] We changed the list-constructor of the NREV benchmark and found out that performances of Quintus Prolog 3.1.1 have dropped from 479 klips to 130 klips while BinProlog achieved a constant 216 klips in both cases. In engines counting less on instruction compression like C-emulated SICStus 2.1 or SB-Prolog 3.1 the drop in performance was however less dramatic (from 139 to 120 klips) and (from 103 to 89 klips) respectively.

[6] This has so little impact on code size that the Solaris 2.2 emulator for BinProlog 2.10 emulator float-point included is still only 49K.

also remove the run-time effort of address look-up for meta-predicates and useless trailing and dereferencing. Remark that instructions like EXEC_SWITCH or EXEC_JUMP_IF have actually a global compilation flavour as they exploit knowledge about the procedure which is called. Due to the very simple initial instruction set of the BinProlog engine these WAM-level code transformations are performed in C at load-time with no visible penalty on compilation time. Note also that we have often combined instruction compression and overlapping of case statements in the C-emulator to further reduce the overall code size as in the case of compressed WRITE-instructions which are just special cases of corresponding compressed UNIFY-instructions.

4.3 Performance evaluation of the BinWAM

Figure 1 compares the performances 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 a variant of Tom Frühwrith's elegant 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 'Prolog killer' 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, its libraries and the public domain reader, tokenizer, writer and DCG preprocessor. CAL is an arithmetic intensive calendar program, DIFFEREN is a symbolic differentiation program and CHOICE[7] is a backtracking intensive program from the ECRC benchmark.

5 Future work

Since binary Prolog encodes the state of the AND-control in terms, visible at the source level, optimizations usually performed on a lower level can now be expressed on the source level. Moreover, in interesting special cases, partial evaluation at source level can largely reduce heap-consumption [4, 6]), showing again that a heap-only approach is not necessarily worse. An advanced technique, EBC-transformations, was developed that is able to perform interprocedural register allocation by source-to-source transformations in programs with difference lists or accumulators [6]. 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

[7] 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 to its smaller and unlinked choice points.

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

Figure 1.

instructions that are present in the BinWAM are used. Comparable optimizations using the environment stack would require low level modifications to the WAM.

6 Conclusion

The simplicity of BinProlog's basic instruction set allowed us to apply low level optimizations easier than on standard WAM. Despite its still higher memory consumption, the BinProlog engine can now be considered a realistic alternative to standard WAM for a full Prolog implementation.

This is especially true in C-emulated WAM's where the simplification of the instruction set has a very positive impact on limiting the interpretation overhead. Absolute performance evaluation compared to a well-engineered[8] conventional WAM like SICStus Prolog 2.1 gives empirical evidence to our claim. The Aquarius system [11] based on the BAM shows that a low level refinement of the WAM can better take advantage of global optimization techniques. A BAM-like continuation passing engine would benefit from the simplicity of the BinWAM.

Acknowledgements

We are grateful for support from NSERC (grant OGP0107411), the FESR of the Université de Moncton and the Jubiläumsfond der Stadt Wien. Sugges-

[8] [2] gives details of its design and optimizations.

tions 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 Mats Carlsson and Bart Demoen 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 always 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] U. Neumerkel. A transformation based on the equality between terms. In *Logic Program Synthesis and Transformation, LOPSTR 1993*. Springer-Verlag, 1993.

[7] J. Stoy. *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory*. Cambridge, MA. The MIT Press, 1977.

[8] 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.

[9] 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.

[10] 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.

[11] P. Van Roy. *Can Logic programming Execute as Fast as Imperative Programming*. Phd thesis, University of California at Berkley, 1990.

[12] M. Wand. Continuation-based program transformation strategies. *Journal of the Association for Computing Machinery*, 27(1):164–180, 1980.

[13] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, Oct. 1983.