The Power of Partial Translation: an Experiment with the C-ification of Binary Prolog

Paul Tarau
Département d'Informatique
Université de Moncton
E-mail: *tarau@info.umoncton.ca*

Bart Demoen
Departement of Computer Science
Katholieke Universiteit Leuven
E-mail: *bimbart@cs.kuleuven.ac.be*

Koen De Bosschere
Vakgroep Elektronica en Informatiesystemen
Universiteit Gent
E-mail: *kdb@elis.rug.ac.be*

Correspondence should be sent to:

Paul Tarau
Département d'Informatique
Faculté des sciences
Université de Moncton
Moncton N.B.
CANADA E1A-3E9

The Power of Partial Translation: an Experiment with the C-ification of Binary Prolog

Paul Tarau * Bart Demoen † Koen De Bosschere ‡

Abstract

We describe a new language translation framework (*partial translation*) and the implementation of one of its instances: the *C-ification* of Binary Prolog.

The basic idea of partial translation is to speed up the execution of emulated code by compiling sequences of emulator instructions down to native code (on top of a C or other native-code compiler) while keeping the advantages of the emulator such as compactness and flexibility. In the case of a logic programming languages, their complex control structure, some large instructions, the management of the symbol table are left to the emulator while the native code chunks will deal with relatively long sequences of simple instructions.

The technique can be seen as an automatic *specialization* with respect to a given program of the traditional *instruction folding techniques* used to speed-up emulators.

When the target language of the host compiler is the same as the implementation language of the emulator (say C), the emulator, the representation of the code generator's byte code as a C data structure, some other the C-ified library routines and hand-written C-code can all be compiled and linked together to form a stand-alone application.

We give full details and a performance analysis of the C-ification of a continuation passing Binary Prolog engine for which large write-mode sequences and the absence of a call-return mechanism make the framework particularly well suited.

The framework is shown as practical and easy to implement and it compares favorably with instruction folding techniques used traditionally to speedup emulators.

Keywords: Programming language translation techniques, Compilation of binary Prolog, WAM, BinWAM, Prolog to C translation

*Département d'Informatique, Université de Moncton, E-mail: *tarau@info.umoncton.ca*

†Departement of Computer Science, Katholieke Universiteit Leuven, E-mail: *bimbart@cs.kuleuven.ac.be*

‡Vakgroep Elektronica en Informatiesystemen, Universiteit Gent, E-mail: *kdb@elis.rug.ac.be*

1 Introduction

Partial C-ification is a translation framework which 'does less instead of doing more' to improve performance of emulators close to native code systems.

The basic idea is the following: starting from an emulator for a language L written in C[1], and a subset of its instruction set (usually frequent and fine-grained instructions which are executed in contiguous sequences) we translate them to C and then simply use a compiler for C to generate a unique executable program[2].

Communication between the run-time system (still under the control of the emulator) and the C-ified chunks is handled as follows. The full emulator is mapped to a C data structure which allows exchange of symbol table information at link time. The C-ified emulator is then compiled to a stand alone executable with performance in the range between pure emulators and native code implementations.

The method ensures a strong operational equivalence between emulated and translated code which share exactly the same observables in the run-time system.

An important characteristic is easy debugging of the resulting compiler, coming from the full sharing of the run-time system between emulated and compiled code and the following property we call *instruction-level compositionality*: if every translated instruction has the same observable effect on a (small) subset of the program's state (registers and a few data areas) in emulated and translated mode, then arbitrary sequences of emulated and translated instructions are operationally equivalent.

2 Partial C-ification

The main reason for translating Prolog to C are

1. gain of speed;
2. portability to multiple architectures;
3. stand alone executables.

One might expect full compilation to C to lead to code which is even better, but this not necessarily true due to

- modularity problems;

[1]The choice of C is not essential here, of course any reasonably efficient target language compilable to native code can be used instead.

[2]A translation threshold allows the programmer to empirically fine-tune the C-ification process by choosing the length of the basic WAM instruction sequence, starting from which, translation is enabled. The process uses a reasonable default value and can be easily controlled also with pragmas for each predicate.

- code growth;
- difficult mapping between Prolog predicates and C-functions (e.g., backtracking);
- expensive (general) C-function calls.

For simpler languages like Janus or KL1 full compilation has been proven quite practical. The major difference with Prolog is that these languages do not feature backtracking, making the run-time system much simpler.

The solution we propose has the following features:

- the modularity problem is solved by the shared emulator;
- the code growth is limited by partial compilation;
- the semantic gap between Prolog and C is bridged by the emulator;
- doing less, not more: we compile relatively small sequences all inside a clause;
- cheap leaf-routines are used.

2.1 Key Idea

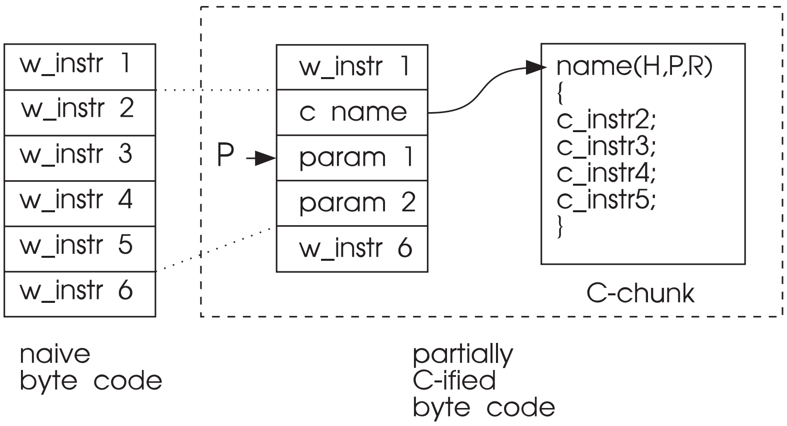

Figure 1: Basic idea of partial C-ification

The figure 1 illustrates the principle of partial C-ification. By a straightforward modification of the emulator written in C we replace some contiguous sequences of interpreted WAM instructions by (usually) small, possibly *leaf* C-routines (*C-chunks*) which get called from a slightly differently compiled version of the emulator, linked together with them.

We can see this process as a partial evaluation of the known *instruction-folding* technique often used to speed-up emulators [3] with respect to a known program.

2.2 Applicability

Long sequences of put instructions used to create heap-based continuations and the absence of call/return makes Binary Prolog (the subset of Prolog obtained through a transformation called binarization) particularly well suited for this framework. We refer the reader to [11] for a more formal description of the binarization transformation and to [4] for its relationship to the WAM. Basically, by a form of AND-continuation passing, with continuations reified as an extra argument of each goal, we manage to have only one call on the right side. This allows the use of a simplified, yet powerful reduced instruction set WAM-like execution model (*BinWAM*), as described in [8, 9, 10]. The main difference at code generation level between the WAM and the BinWAM is the call-return mechanism for the former versus the no-return, continuation passing execution for the later. In the frequent case when a lot of state information is passed between the head and the last call, an instruction in the BinWAM actually replaces two instructions in the WAM (see [12]), which has to save and restore Y variables upon entering and returning from a call, as the reader can see on an (extreme) example like:

```
a(X1,...Xn):-b,c,d,e,f,g(X1,...,Xn).
```

where `X1...Xn` have to be saved in the WAM when calling `b,c,d,e,f`.

Not only code-sequences become shorter but also more uniform after binarization. Last but not least, binarized clause bodies have also the nice property, which will be illustrated in the following example, of generating *leaf-routines* i.e. small C-functions which will use no C-stack space. We have therefore chosen the code sequence generated by binarized clause bodies as the target of our empirical study of the partial C-ification process.

3 Example

Let us follow the translation process on a typical example.

[3] Which consists of creating a plethora of combinations of frequently occurring operation-code pairs or triplets with their associated C-code, all dispatched through a big switch statement and/or some other (for instance first-order label based) direct jumping techniques.

3.1 Prolog version

We have chosen the following clause containing some duplicate symbols and numbers:

```
treesize(tree(Left,Right),S0,S) :-
   add(S0,1,S1),
   treesize(Left,S1,S2),
   treesize(Right,S2,S).
```

3.2 Binarized version

After binarization, our example becomes:

```
treesize(tree(Left,Right),S0,S,C) :-
   add(S0,1,S1,
      treesize(Left,S1,S2,
         treesize(Right,S2,S,C))).
```

3.3 Basic WAM-code

From this binary form we obtain the following[4] WAM-code:

```
[1] clause_? treesize 4
[2] firstarg_? tree/2 11
[3] get_structure tree/2 var(1) % var(1-1/2,2/2)
[4] unify_variable  var(5) % var(5-3/10,1/2)
[5] unify_variable  var(6) % var(6-4/15,1/2)
[6] get_variable arg(2) var(1) % var(1-5/19,1/2)
[7] c_chunk_begin 15 var(7) % var(7-8/23,1/2)
[8] put_structure treesize/4 var(8) % var(8-9/22,1/2)
[9] write_value  var(5) % var(5-3/10,2/2)
[10] write_variable  var(5) % var(5-11/21,1/2)
[11] write_variable  var(9) % var(9-12/16,1/2)
[12] write_variable  var(10) % var(10-13/14,1/2)
[13] push_structure treesize/4 var(10) % var(10-13/14,2/2)
[14] write_value  var(6) % var(6-4/15,2/2)
[15] write_value  var(9) % var(9-12/16,2/2)
[16] write_value  var(3) % var(3-6/17,2/2)
[17] write_value  var(4) % var(4-7/18,2/2)
[18] put_constant arg(2) 1
[19] put_value arg(3) var(5) % var(5-11/21,2/2)
[20] put_value arg(4) var(8) % var(8-9/22,2/2)
[21] c_chunk_end 15 var(7) % var(7-8/23,2/2)
[22] execute_? add 4
```

[4] Actual code, integrated in BinProlog 3.00 available by ftp from clement.info.umoncton.ca has been used for generating this example.

Note the presence of `c_chunk_begin` and `c_chunk_end` in the WAM-code. The leaf C-routine will be generated from the sequence of instructions between them.

3.4 C-ifing the emulator

To be able to call the C-routine from the emulator we have to know its address. A simple and fully portable technique is to C-ify the byte-code of the emulator as an array of structures:

```
EMULATOR DATA STRUCTURE:
......
{63,7,0,(void *)xx_399},
{6,0,4,"treesize"},
{64,0,1,"?"},
{17,0,4,"add_node"},
.......
```

3.5 Generating the C-chunk

The C-chunk itself follows as a sequence of C-macros. Notice the presence of the `xx_399` function name which will actually appear only when the macro `c_chunk_variable(xx_399)` will get expanded to a C-function by the preprocessor.

```
c_chunk_variable(xx_399)
put_structure(0,0,8)
write_value(1,5)
write_variable(2,5)
write_variable(3,9)
write_constant(4,0)
write_value(5,6)
write_value(6,9)
write_value(7,3)
write_value(8,4)
put_integer(1,2)
move_reg(3,5)
move_reg(4,8)
c_chunk_value(9,1).
```

3.6 Expanding the chunk to a function

We have encapsulated the actual code through a C-macro based second interface to keep the compiler simple (about one hundred extra lines added to the emulated code generator). This also made the process of transforming BinProlog into a compiler with basic C-ification capability an about 1-week, 1-person task. Here is the actual C function, after preprocessing by `cpp`, except for the low level INPUT_INT macro we have edited back for readability.

```
term  xx_399 (H,regs,P,ires)
   register term H,regs;
   register instr P; long ires;
{
 regs[8] = (cell)(H+ 0); H[0] = P[0];
 H[1] = regs[5];
 regs[5] = H[2] = (cell)(H+2);
 regs[9] = H[3] = (cell)(H+3);
 H[4] = P[0];
 H[5] = regs[6];
 H[6] = regs[9];
 H[7] = regs[3];
 H[8] = regs[4];
 regs[2] = INPUT_INT(1);
 regs[3] = regs[5];
 regs[4] = regs[8];
 regs[-1]  = (cell)(P+ 1); /* returned  P, possibly affected by CUT */
 return H+ 9 ;                /* returned  new heap pointer */
}
```

3.7 Assembly code

It is interesting to take a look to the actual assembly (SparcCenter 1000, gcc -O2) listing, which shows clearly that our objective to have high quality code has been attained with minimal effort.

```
EMULATOR DATA STRUCTURE:
....
.word xx_399
.byte 6
.byte 0
.half 4
.word .LLC993
.byte 64
.byte 0
.half 1
...

C_CHUNK:

xx_399:
!#PROLOGUE# 0
!#PROLOGUE# 1
st %o0,[%o1+32]
```

```
ld [%o2],%g2
st %g2,[%o0]
ld [%o1+20],%g3
add %o0,8,%g2
st %g2,[%o0+8]
st %g3,[%o0+4]
st %g2,[%o1+20]
add %o0,12,%g2
st %g2,[%o0+12]
st %g2,[%o1+36]
ld [%o2],%g2
st %g2,[%o0+16]
ld [%o1+24],%g2
st %g2,[%o0+20]
ld [%o1+36],%g2
st %g2,[%o0+24]
ld [%o1+12],%g2
st %g2,[%o0+28]
ld [%o1+16],%g2
add %o2,4,%o2
st %g2,[%o0+32]
mov 5,%g2
st %g2,[%o1+8]
st %o2,[%o1-4]
ld [%o1+20],%g3
add %o0,36,%o0
ld [%o1+32],%g2
st %g3,[%o1+12]
retl
st %g2,[%o1+16]
.LLfe399:
.size  xx_399,.LLfe399-xx_399
```

3.8 Discussion

All Prolog symbols are internalized by the emulator. The Prolog symbols that are needed by the C-chunk are put in the argument list of the C-call on a constant offset from the current P-pointer. So, the chunk can access them with a single load operation[5]. There is no need for the C-chunk to directly access the Prolog symbol table. The final argument list of the C-call is built once by the loader of the emulator.

An alternative is to compile the symbol table down to unique C-objects that are given unique addresses by the C-compiler and linker. Given the fact that these objects will have full word-wide addresses, and given the penalty to load immediates of word length for RISC architectures, this scheme does not give a

[5] This is similar to arguments of instructions in threaded code [1]

speedup (4 memory accesses for both approaches) and is therefore not worth considering.

Hence, it turns out that the emulator is a good tool for sharing symbol-tables between multiple C-modules and that compiling the symbol table down to C is not a good idea. This is yet another illustration of the fact that it is often better *not* to translate everything into C.

The C-symbols (i.e., the names of the functions that contain the C-chunks) are resolved by the C linker in the standard way. Since we cannot link a file containing byte code by means of the standard linker, the byte code file is first converted into a C file containing a huge array with the same information, and also the names of the C-chunks. After compilation and linking with the emulator, the addresses of the chunks will be automatically resolved and the result will be a stand alone Prolog application.

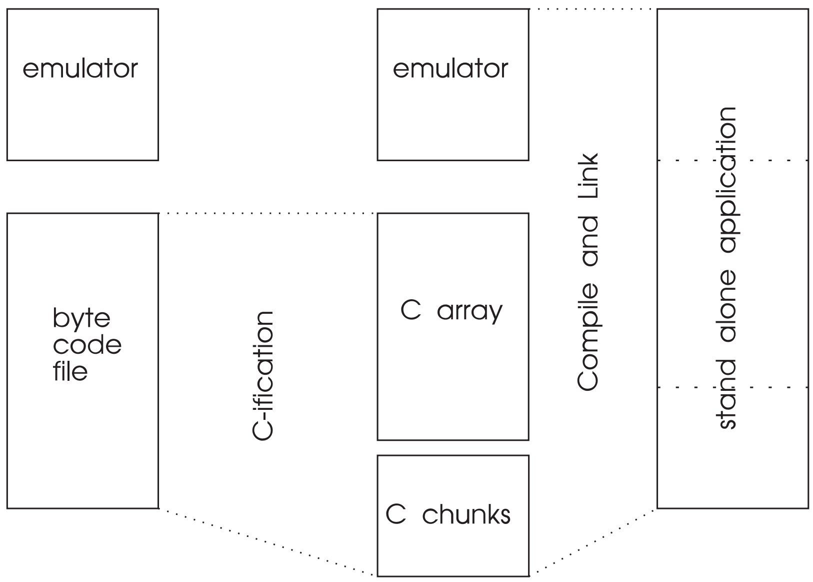

Figure 2: Preparing the byte code file to be linked by the C linker.

Prolog symbols that are used twice by the C-chunk are only generated once as argument in the P-code section. Re-using arguments saves program code (improves locality), and speeds up the execution of the chunk by loading a common symbol only once from the argument list, and writing it as many times as needed onto the heap. The multiple assignment of C (`a=b=c`) is used to express that a particular word is load once and stored many times.

BinProlog's tag-on-data representations makes the copying of a symbol from the code area onto the heap very efficient. Pushing a functor on the heap can be done as `H[Offset1]=P[Offset2]` with `Offset1` and `Offset2` known at compile time where `P[Offset2]` contains the full representation of the functor i.e. `<Arity,SymbolNumber,Tag>` which fits in one word.

Untagged C-pointers being the same as Prolog's logical variables, the C-equivalent of emulated code becomes simpler. Small integers are passed as immediate operands.

Partial C-ification becomes very interesting in the presence of

- partial evaluation which creates long bodies, i.e. it ensures a better hit on long sequences;
- optimal term creation;
- term compression which also reduces the size of the chunks and simplifies their data movements;
- a top-down compilation scheme, which takes advantage of free indirect addressing with small offsets;
- register allocation as some pseudo-registers in the emulator become real ones in C.

Since most of the instructions in the C-chunks have arguments that are at a constant offset, an optimizing compiler can take full advantage of this information and generate the most efficient code for a particular sequence. It turns out that on modern RISCs, the address calculation of adding a small offset to an address that resides in a register is for free. Furthermore, by not using WAM-registers but C-variables to store temporary data in the chunk, the compiler might use hardware registers to keep intermediate values.

Since the C-chunks do normally not contain calls to other procedures or functions, they will get compiled into (more efficient) leaf routines. Leaf routines use a limited (but sufficient) subset of the hardware registers, so that save/restore of the caller's registers is avoided. Calls and returns are therefore jump instructions with the return address kept in a fixed register (%o7 on Sparc).

Since the code of the C-chunks is simple and very regular, a fairly reduced amount of resources (swapping space, time) is needed to allow the C-compiler to generate good code for the chunks.

4 Performance evaluation

Performance clearly depends on the amount of C-ification and on the statistical importance of C-ified code in the execution profile of a program.

We have noticed between 10-20% speed increase for programs which take advantage of C-ified code moderately, like `tak`, `meta-interpretation of nrev`, `an intuitionistic implication benchmark`, the bootstrapping of our own `C-code generator` etc. As these programs spend only 20-30% of their time in C-ified sequences performances are expected to scale correspondingly when we extend this approach to 90% of the BinProlog instruction-set (almost everything except some heavy choice-point manipulation and control instructions).

The final version of this paper will contain extensive benchmarking information based both on the already implemented C-ification of PUT-sequences and the more comprehensive C-ification covering builtins, arithmetics, self-recursive predicates and head-instructions which are currently being implemented.

5 Related work

Simpler logic languages like Janus [6], KL1 [2] have been successfully compiled to C with speed which competes successfully with native code compilers.

On the other hand, most of the attempts for the C-translation of full Prolog have managed to get performance only slightly better or comparable to well-engineered emulators like Quintus or SICStus Prolog [7].

In [7], the Prolog to C compiler is built on top of a traditional WAM compiler. The WAM instructions are expanded in-line or they become function calls. The call sequence of the predicates is controlled by a dispatching loop.

A smart (but machine specific) technique is used in the WAMCC system based on [5] to avoid the dispatching loop by implementing global labels and direct jumps inside C-functions, by using **asm** directives. Speed is in the middle between emulated and native code for most programs, exceptionally good on arithmetics (with data validity checking off) although on some programs like `naive reverse` the overall performances (127KLIPS on a SparcELC) are significantly lower than emulated BinProlog (220KILPS on a SparcELC).

The approach described here is significantly different from previous work. We have started with the clear objective of *partial translation* from one of the fastest existing C-emulated Prologs (BinProlog) and to obtain overall performances similar to the best full-translation schemes. Therefore, in practice, our method (which needed about 1 programmer-week of work for minimal modifications to the emulator) looks highly practical and general enough to be applied to other emulator based implementations of high level languages.

Moreover our scheme is by its nature fine-tunable in terms of the amount of translation, giving to the programmer the opportunity to configure it for a large spectrum of code-size/speed ratios.

One of the main decisions one has to make when compiling Prolog to C, is how to map the execution of Prolog to the runtime stack of C: it is tempting to map e.g., the WAM environment stack to the C stack. This is however a mistake for several reasons: the WAM environment stack is not a proper stack as soon as backtracking comes into the picture: this imposes some painful and inefficient work arounds and it is difficult for a garbage collector to find active environment variables in older register windows.

Most important however: the penalty of window overflows on the Sparc architecture is very high; in [12], it is reported that Aquarius is faster for the `tak` benchmark than C; this is due to the fact that in Aquarius no register window overflows occur during execution of the benchmark: Aquarius does indeed not map the environments to the C stack.

So without provisions to avoid register window overflows, mapping the environment stack to the C stack is a bad idea. Also the choicepoint stack could be mapped to the C stack: this will pose problems for implementing a garbage collector and cut.

These problems are naturally avoided in our approach, where the emulator handles the run-time environment and manages the stacks.

Experiments performed on limited scale in [3] show that high speed is attainable, with full expansion of BinWAM code. This is appropriate especially for small self-recursive and arithmetic-intensive predicates and is complementary to the approach described here.

6 Future Work

6.1 Modules

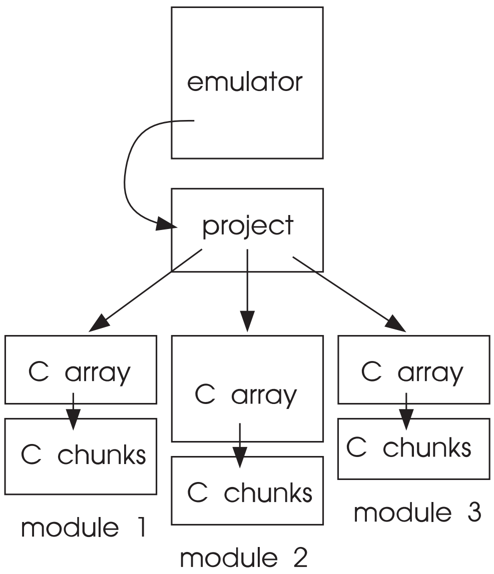

Figure 3: Module system.

The figure 3 shows a module proposal taking into account our C-ification process. A module system for C-translated Prolog is proposed in [3]. This can be applied quite naturally to our translation mechanism as is the mapping to C described in [3] is also based on binary prolog. On the other hand, modularity is easily obtained in our case by simply making emulated code to support modular compilation. We have started to implement a *make* facility inside the compiler which already deals with modular compilation of emulated code.

6.2 Builtins

We have prototyped the C-ification of PUT sequences and studied the impact. However this can be extended to all the inline operations which can be processed in Binary Prolog before calling the 'real goal' in the body [4].

Chunks containing builtins that do not require a procedure call will still generate leaf-routines.

6.3 Small self-recursive predicates

Small self-recursive predicates (like `append/3`) are a good target for full C-ification for the deterministic case while the rarely used non-deterministic case can be executed by escaping through to the emulator.

Applying this technology for the builtin `append/3` of BinProlog gave performances about 2 times faster than native SICStus Prolog 2.1_9 on the *naive reverse benchmark*. Automating the process is fairly straight-forward as the property of being self-recursive is local to the predicate. Basically, a self-recursive predicate is transformed to a while loop which advances on its known input data until non-determinism or non-applicability of the self-recursive clause is detected, when through continuation manipulation the function escapes back to the emulator.

7 Conclusion

We have presented a new technique called 'partial translation' and studied its use for Prolog to C translation. The technique has allowed to control the amount of translation to C for an optimal speed/code-size ratio. Although our experiments have been described in the context of Prolog to C translation, the technique itself is general purpose. The technique gives performance in a range between emulated and native code with very little implementation effort and ensures portability through C. Compilation of head and inline builtins can be done along the same lines leading to performance close to 'real' native code systems.

Acknowledgements

Paul Tarau thanks for support from NSERC (grant OGP0107411), from the FESR of the Université de Moncton, and from K.U.Leuven through Fellowship F/93/36. Bart Demoen thanks the Belgian Ministry (DWTC) for support through project IT/IF/4. Koen De Bosschere is senior research assistant with the Belgian National Fund for Scientific Research.

References

[1] J. Bell. Threaded code. *Communications of ACM*, 16:370–372, June 1973.

[2] T. Chikayama, T. Fujise, and D. Sekita. A portable and efficient implementation of kl1. In *Proceedings of Programming Language Implementation and Logic Programming*, Lecture Notes in Computer Science. Springer, Sept. 1994. to appear.

[3] K. De Bosschere and P. Tarau. High Performance Continuation Passing Style Prolog-to-C Mapping. In E. Deaton, D. Oppenheim, J. Urban, and H. Berghel, editors, *Proceedings of the 1994 ACM Symposium on Applied Computing*, pages 383–387, Phoenix/AZ, Mar. 1994. ACM Press.

[4] 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.

[5] D. Diaz and P. Codognet. A minimal extension of the WAM for clp(FD). In D. S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, pages 774–790, Budapest, Hungary, 1993. The MIT Press.

[6] D. Gudeman, K. De Bosschere, and S. Debray. `jc`: An Efficient and Portable Sequential Implementation of Janus. In K. Apt, editor, *Joint International Conference and Symposium on Logic Programming*, pages 399–413, Washington, Nov. 1992. MIT press.

[7] R. Horspool and M. Levy. Translator-based multiparadigm programming. *Journal of Software and Systems*. to appear.

[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. 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.

[10] P. Tarau. Low level issues in implementing a high-performance continuation passing binary prolog engine. In M.-M. Corsini, editor, *Proceedings of JFPL'94*, June 1994.

[11] 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.

[12] P. Tarau and U. Neumerkel. A Novel Term Compression Scheme and Data Representation in the BinWAM. In *Proceedings of Programming Language Implementation and Logic Programming*, Lecture Notes in Computer Science. Springer, Sept. 1994. to appear.