Partial Translation: Towards a Portable and Efficient Prolog Implementation Technology *

Paul Tarau † Koen De Bosschere ‡ Bart Demoen §

Abstract

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

Our partial translation framework compiles selected sequences of emulator instructions down to native code. 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.

In our implementation, the complex control structure, some large instructions and the management of the symbol table are left to the emulator while the translated code deals with relatively long sequences of simple instructions. After compilation, the generated code is linked with a target language representation of the emulator's byte-code and the emulator itself to form a stand-alone application. The composite runtime system's behavior can be seen as a form of 'coroutining' between emulated and native code.

The framework supports modular compilation, allows programmer control of the speed vs. size optimization, is fully portable and has a performance that ranges between the performance of emulated code and that of native code. Our design has been proven practical in the implementation of the C-code generator of a fairly complete multi-platform Prolog system (BinProlog) available by ftp at clement.info.umoncton.ca.

Keywords: Compilation, Code generation, WAM, Prolog to C translation

*Extended and revised version of the paper *The Power of Partial Translation: an Experiment with the C-ification of Binary Prolog* presented at the ACM SAC Conference, Nashville, TA, 1995

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

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

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

1. Introduction

In an attempt to create portable implementations of logic programming languages, some translation based frameworks have recently been proposed [5, 6, 9, 15, 16, 17], and have been compared [13]. A common characteristic of all these implementations is that portability is achieved through the use of C as target for the translation. By either using only standard constructs, or by generating code for a particular compiler (e.g., `gcc`), the generated code becomes platform-independent.

However, unlimited portability sometimes causes inefficiencies. Indeed, early attempts [9, 17] showed that a straightforward translation into (standard) C is not particularly efficient (long code and slow execution), with one important exception: the implementation of committed choices languages such as KL1 [5] and Janus [7, 11, 15]. The major reason (besides the use of clever optimizations like call-forwarding [7]) is that the semantic gap between committed choice logic programming languages and an imperative language such as C is relatively small.[1]

When looking at the basic cause of inefficiency, it turns out that it is mostly due to a limited number of inefficiencies in the generated C-code such as the fact that there are only two ways to jump to an address that is not known at compile time: either by means of a function call, which is not particularly efficient, and also requires a function return, or by means of a huge switch statement, which is not efficient either. The latter solution is fully portable, but has two disadvantages: since the whole program has to be compiled as a single C-function, it does prohibit the implementation of modular compilation, and compilers sometimes get confused by the enormous C-function and therefore generate sub-optimal code. Furthermore, the time to compile a non-trivial Prolog application becomes unreasonably long.

When using one function per clause, or one function per Prolog procedure, the overhead caused by the function call and return is a major source of inefficiency [4, 9] and makes the implementation of last call optimization difficult. In WAMCC [6], a combination of assembler directives and tricks to mislead the compiler and to bypass the function calls, is used to obtain a high performance implementation of Prolog by means of translation. Unfortunately, this solution needs minor modifications when being ported, and still suffers from code explosion.

This paper advocates a completely new approach based on two observations:

1. C compilers produce their best code when compiling small function containing little or no control constructs. On the other hand, the implementation of head unification and backtracking requires much testing and jumping.

2. Prolog emulators are highly optimized toward the efficient implementation of the basic execution model. Many Prolog emulators do a much better job than the average C compiler would do for controlling the execution of a Prolog program.

Therefore, instead of fully translating Prolog into C, we propose to only partially translate it. The parts that would give rise to complex or long code sequences are

[1] There is no support for backtracking needed, and even unification often boils down to assignment. In contrast to committed-choice languages, in the presence of non-determinism, Prolog's relatively high backtracking costs can easily dilute the speed-up with respect to emulated code, while adversely affecting locality .

not translated. The compiler can decide at compile time whether or not to translate depending on the expected speedup of the translation.

An additional and important benefit of our approach is that there is no code explosion because the code that tends to explode most when translating to an imperative language is precisely the control part, not the data manipulation part.

The paper starts by explaining the basics of partial translation, followed by a fully worked out example (a recursive BinProlog clause on a Sparc architecture). The paper is concluded with some performance data and a comparison with related work and some directions for future work.

2. Partial Translation

Basic Principle. Figure 1 illustrates the basic principle of partial translation. We start from the (optimized) byte code stream generated for the emulator. In this byte code stream, we search for a contiguous sequence of instructions that is worth being translated. Subsequently, that instruction sequence is translated into a C-function and the original instruction sequence is replaced by a function call (new WAM instruction) to the newly generated C-function. The function is called with the following arguments: heap pointer `H`, the register set `regs`, and a pointer `P` to an argument list containing the symbol table entries of the symbols used by the C-function; as the symbol table is still managed by the emulator, the symbol table entries are not known at compile time and must therefore be passed explicitly to the C-function at run-time.

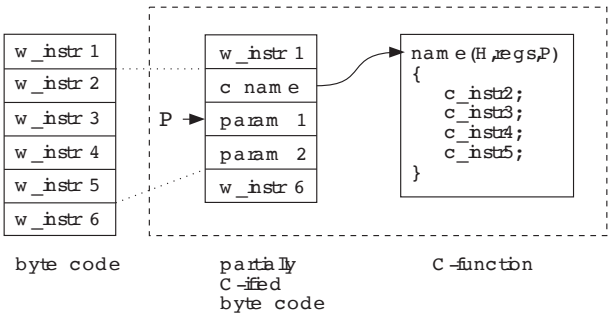


Figure 1: Basic idea of partial C-ification

An important characteristic of the partial translation scheme we propose is that the translated system has a strong operational equivalence with the emulated code, as both share exactly the same observables in the run-time system thanks to the principle of *instruction-level compositionality*: if every translated instruction has the same observable effect on a (small) subset of the program state (registers and a few data areas) in emulated and translated mode, then arbitrary sequences of emulated and translated instructions are operationally equivalent.

Speedup over emulated code. For partial translation to produce a speedup, the execution time of the byte code instruction sequence must be greater than the execution time for the C-function plus the extra overhead of the function call. Since

the real execution times of the emulated and the translated code are generally not known, the compiler could rely on programmer declarations specifying for a given code slice (module, file, set of predicates, etc.) a threshold such that no sequence of instructions shorter then the threshold is C-ified. Alternatively, execution profiling can decide for each sequence of abstract machine instructions whether they should be C-ified. Therefore, the performance of a partially translated Prolog system will always range between that of purely emulated systems, and native code implementations. The final performance will depend on the amount of translation, and the resulting speedup[2].

Additional optimizations. The more efficient the C-function is, and the more interpretative overhead there is in the byte code, the bigger the speedup resulting from partial translation will be.

C-functions are executed most efficiently if they do not contain function calls. In that case they are compiled into efficient *leaf routines* that do not consume stack space. Hence, it is interesting not to C-ify instructions that do require function calls.

On the other hand, some emulator instructions benefit more from C-ification than others. The finer the grain of the instruction, the higher the interpretative overhead, and the higher the speedups that can be obtained. As mentioned earlier, instructions that control the execution of the program will not be C-ified as their expected speedup is too limited, and the generated code is too long and too complex.

Putting it all together. In order to produce a working system, the generated C-code must be combined with the partially C-ified byte code. Therefore, the function addresses in the byte code must be resolved. This is easily done by translating the byte code into a huge C data structure[3] containing the symbolic names of the functions. Compilation re-generates the byte code file, and linking will resolve the function addresses (and generate error messages when functions are missing).

This executable can subsequently be linked either statically or dynamically with the emulator in order to produce a stand-alone executable (see Figure 2), another goody of translation into C. In the case of dynamic linking, the resulting executable is small as it does not contain the full emulator.

Major benefits of partial translation. The solution we propose effectively solves the problems mentioned in the introduction.

- The modularity problem is solved. Modules can be compiled separately, the common data areas such as the symbol table are being managed by the shared emulator.

[2]Note that in practice precise static estimates of execution times are not always enough, due to intricate caching and paging related factors. However, as the decision to C-ify or not a given sequence is ultimately under the programmer's control (as it is the case with loop-unrolling or inline expansion in C) it is always possible to fine-tune the amount of partial translation to avoid a degradation of overall performance.

[3]C-compilers react moderately well on ingesting our C-ified byte-code, due to it's very simple and uniform nature. For the C-ified BinProlog system itself (a 400K file) it took 27s to lcc and 76s to gcc, assembler included, on a 25 MHz Sparcstation 10-20 to generate an object file. The full Prolog-to-C translation of the BinProlog 4.00 compiler and various run-time modules to C on top of the emulator took 54s on the same machine. No compiler overflow has been experienced with any of our applications on about 10 different UNIX and Intel platforms.

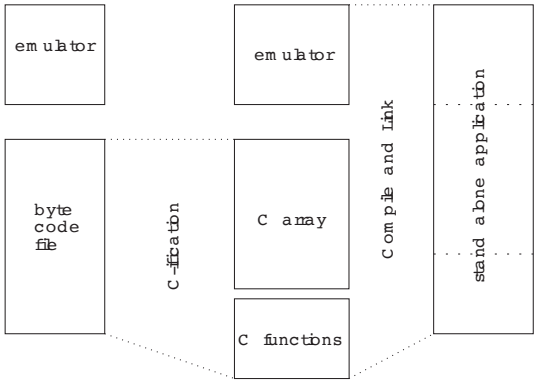


Figure 2: Preparing the byte code file to be linked by the C linker.

- Code growth is limited by partial translation. Parts that can cause excessive code growth are not translated.
- Portability is obtained by generating standard C. The portability of a Prolog application further depends on the portability of the emulator. If it is written in standard C, practically unlimited portability is achieved.
- Good performance is guaranteed by the execution time estimates that make that the partially translated code is always faster than the non-translated code. So there is never a slowdown.

3. Example

In order to fully exploit the capabilities of partial translation, we need long sequences of fine grained instructions, preferably not containing control instructions. In WAM-based Prolog implementations, except for long sequences of built-in operation as for instance complex numeric operations, the only sequences of instructions that can be C-ified are the instructions between two predicate calls, possibly including the inline predicates if they do not contain function calls.

However, there is a subset of Prolog obtained through a transformation called binarization[4], namely Binary Prolog. It often contains long sequences of (fine grained) `put`-instructions to create heap-based continuations. Besides this property, the absence of call/return makes Binary Prolog particularly well suited for partial translation. As binarization can be applied as a preprocessing step to arbitrary Prolog code [12] this restriction does not affect the generality of the approach. Arguably, some other program transformations (for instance *unfolding*) can also be used at source level to produce clauses with longer sequences of control-free instructions.

[4]We refer the reader to [24] for a more formal description of the binarization transformation and to [12] for its relationship to the Warren Abstract Machine.

In order to make things clear, the binarization transformation is explained by means of an example program. The well-known Prolog procedure

```
nrev([],[]).
nrev([X|L],R) :- nrev(L,L1), append(L1,[X],R).
```

is binarized into

```
nrev([],[],Cont) :- call(Cont).
nrev([X|L],R,Cont) :- nrev(L,L1,append(L1,[X],R,Cont)).
```

Every clause has now one extra argument, the so-called *AND-continuation*. The continuation is passed from the clause head to the clause body, except in the case of a fact where it is executed. Binarized clauses have only one call in their body, namely the first goal of the original body. The remaining goals are passed as continuation. The continuation `append(L1,[X],R,Cont)` is created on the heap before the body goal is called. This is achieved by a sequence of fine grained `put`-instructions.

The abstract machine (*BinWAM*) used to execute binarized clauses is simpler than the standard WAM, and is described in [20, 21, 22, 25]. The main difference at code generation level between the WAM and the BinWAM is that in the latter (i) there is no return from predicate calls, and (ii) continuations have to be created explicitly. The creation of continuations is well suited to be C-ified as it consists of long sequences of `put`-instructions. These sequences do neither contain control instructions, nor function calls. So they will be compiled into efficient *leaf routines* which do not consume stack space.

The remainder of this section is devoted to the full description of the C-ification of the `treesize/3` predicate.

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))).
```

Hence, the clause gets an extra argument that carries the continuation, i.e., the part of the program that is still to be executed. Before `add/4` can be started, its continuation consisting of the nested `treesize/4` terms is created on the heap.

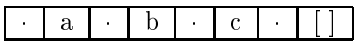
3.3. Basic BinWAM-code

From this binarized form we generate the following WAM-like code:

```
 [1] clause_? treesize 4
 [2] firstarg_? tree/2 11
 [3] get_structure tree/2 var(1)        % tree/2
 [4] unify_variable  var(5)             % Left
 [5] unify_variable  var(6)             % Right
 [6] begin_c_chunk var(7)
 [7] get_variable arg(2) var(1)         % S0
 [8] put_structure treesize/4 var(8)    % make continuation
 [9] write_value  var(5)                % Left
[10] write_variable  var(5)             % S1
[11] write_variable  var(9)             % S2
[12] push_constant treesize/4 var(10)   % term compression
[13] write_value  var(6)                % Right
[14] write_value  var(9)                % S2
[15] write_value  var(3)                % S
[16] write_value  var(4)                % C
[17] put_constant arg(2) 1              % 1
[18] put_value arg(3) var(5)            % S1
[19] put_value arg(4) var(8)            % move continuation
[20] end_c_chunk var(7)                 %   to its register
[21] execute_? add 4                    % call body
```

Note the presence of `begin_c_chunk` and `end_c_chunk` in the WAM-code. The `var(7)` comes from a compile-time logical variable which allows them to exchange information without an additional pass. In the generated code, `end_c_chunk` will know the heap offset (computed at compile time) it has to add to the heap pointer after the chunk returns. The C-function will be generated from the sequence of instructions between them, but only if this will give rise to a speedup, based on the length of the C-ified sequence.

Also note the term compression [25] that is used for the creation of the continuation. If a term has a last argument containing a functor, the tag-on-data[5] representation used in BinProlog can avoid the extra pointer from the last argument to the functor cell and simply make them collapse. Obviously the unification algorithm must take care of this case, but the space savings are important, especially in the case of lists which become contiguous vectors with their N-th element directly addressable at offset `2*sizeof(term)*N+1` bytes from the beginning of the list:



As a result of term compression the functors of the last arguments are created with `write_constant` instead of creating a pointer to the next contiguous cell.

[5] Note that as atomic values (symbolic constants and integers) fit in a cell, they are 'stored in the pointer' as in the tag-on-pointer approach. Arguably, this would justify calling our scheme 'a mixture of object- and pointer-tag' representation. Note that such a representation is quite common (especially in object-oriented languages).

3.4. Generating the C-function

Based upon the estimated execution times, the translator decides whether or not to translate the instructions between the `begin_c_chunk` and `end_c_chunk` to generate a new function (say `xx_1`). In the current implementation, the execution time is estimated by means of a number of instructions. Below a given number of instructions, there is no C-ification.

The original byte code then becomes.

```
[1] clause_? treesize 4
[2] firstarg_? tree/2 11
[3] get_structure tree/2 var(1)        % tree/2
[4] unify_variable  var(5)             % Left
[5] unify_variable  var(6)             % Right
[6] call_chunk xx_1                    % call function
[7] symbol treesize/4                  % chunk parameter
[8] execute_? add 4                    % call body
```

As the C-function has unlimited access to the WAM registers, and as the function is currently never re-used (although it could in some cases), all information that is available at compile time can be used immediately in the C-function. Only run-time information such as symbol table entries[6] must be passed explicitly as parameters (e.g., `treesize/4`). The `P` register points to the argument vector when the C-function is executed. Hence, the C-function can easily access the arguments. Multiply occurring arguments are merged to save memory space and integers are directly embedded in the C-code.

The C-function itself (`xx_1`) is generated from a sequence of C-macros. Notice the presence of the `xx_1` function name which will actually appear only when the macro `begin_c_chunk(xx_1)` will get expanded to a C-function by the C-preprocessor. We had to choose synthetic function names as not all Prolog names boil down to valid C names. The `put_structure` and `put_constant` instruction get their source operand from the argument list of the `call_chunk` instruction (`treesize/4`).

```
begin_c_chunk(xx_1)
move_reg(1,2)              % S0
put_structure(0,0,8)       % treesize/4
write_value(1,5)           % Left
write_variable(2,5)        % S1
write_variable(3,9)        % S2
write_constant(4,0)        % treesize/4
write_value(5,6)           % Right
write_value(6,9)           % S2
write_value(7,3)           % S
write_value(8,4)           % C
put_integer(1,2)           % 1
move_reg(3,5)              % S1
move_reg(4,8)              % continuation
end_c_chunk(9,1)   % update heap (H=H+9) and code (P=P+1) pointer
```

[6] More precisely, functor cells also containing the arity and the tag.

After expansion of the macros, the C-function looks as follows:

```
term  xx_1(H,regs,P)
   register term H,regs;
   register instr P;
{
  regs[1] = regs[2];
  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); /*new code pointer */
  return H+9;              /*new heap pointer */
}
```

Prolog symbols that are used twice by the C-function 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 function 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 term should be re-used.

BinProlog uses a *tag-on-data* representation instead of the usual tagged pointer representation of classical WAMs. For instance, a functor is represented as:

ARITY	SYMBOL-NUMBER	TAG

Copying a functor from the code area into the heap area can be done in one instruction `H[Offset1] = P[Offset2]`, where `P[Offset2]` contains the full representation of the functor, and with `Offset1` and `Offset2` known at compile time. The fact that symbol table information, tag and arity are melted into the same word makes the copying of a functor from the code area into the heap area very efficient. By choosing TAG=0 for variables and having only 2-bit tags[7], every memory address (C pointer) looks like a logical variable. This gives a low overhead and less error-prone integration of C code in the engine and it has a positive impact on performance on architectures where base addressing is not for free.

The code could still be improved by either not using WAM-registers to store temporary data in the C-function but by defining local C-variables for which the compiler might allocate hardware registers or by rescheduling the instruction sequence in such a way that the temporary registers become superfluous. Registers `regs[8]`, `regs[5]`, and `regs[9]` are typical examples. The improved code looks then as:

[7] Tags become 3 bits on 64 bits machines with the result that code will adjust to the larger word size automatically, i.e., this gives portability to 64 bits without any conditional code

```
term  xx_1(H,regs,P)
   register term H,regs;
   register instr P;
{
  regs[1] = regs[2];
  H[0] = H[4] = P[0];
  H[1] = regs[5];
  H[7] = regs[3];
  regs[3] = H[2] = (cell)(H+2);
  H[6] = H[3] = (cell)(H+3);
  H[5] = regs[6];
  H[8] = regs[4];
  regs[2] = INPUT_INT(1);
  regs[4] = (cell)(H);
  regs[-1]  = (cell)(P+1); /*new code pointer */
  return H+9 ;             /*new heap pointer */
}
```

This optimization has not yet been carried out in the current implementation.

3.5. C-ifying the emulator

To be able to call a C-function from the emulator we have to know its address. Unfortunately, the final address can only be determined by the linker. A simple and fully portable technique to plug the address of a C-function into the byte code is to C-ify the byte-code of the emulator into a huge C data structure, containing the names of the C-functions. After compilation the byte code file is re-generated, and after linking with the emulator, all the missing addresses will be resolved. The result will be a stand-alone Prolog application (see also Figure 2).

```
EMULATOR DATA STRUCTURE:

struct bp_instr {
   unsigned char op, reg;
   unsigned short arity;
   char *name;
}
user_bp[] =
{
  ......
  {63,7,0,(void *)xx_1}, /* Start of C function (symbolic address).      */
  {6,0,4,"treesize"},    /* String constant at P[0] to be passed to xx_1 */
  .....                  /* after being replaced with actual symbol      */
                         /* by the emulator's symbol table manager.      */
  {64,0,2,"?"},          /* End of C function                            */
  ......
}
```

The array of structures which represents the byte code starts with an entry containing the address of the C-function, followed by the symbol table entries needed by the C-function. Various emulated instructions are represented as a structure, containing:

- an opcode
- a first register number
- a second register number or an arity
- a string or a function pointer

This regular format allows loading one word at a time and extracting various fields with fast register-only operations [8].

During the execution of the function, the P-pointer is pointing to the first argument. As the argument offsets w.r.t. the P-pointer are known at compile time, the function can access them with a single load operation at no extra cost, and an optimizing compiler can take full advantage of this information and generate the most efficient code for a particular sequence. Obviously, integers are passed as immediate operands, directly in the generated C-macro. There is no need for the C-function to directly access the Prolog symbol table. The argument list of the C-call is built only once by the loader of the emulator. Note that our symbol passing scheme from the emulator to the C-code is similar to argument passing in threaded code [1].

3.6. Assembly code

It is interesting to take a look at the actual assembly (Sparc, Solaris 2.x, gcc 2.6.3 -O2) listing, which shows clearly that our objective to have high quality code has been attained with minimal effort (H is in %o0, regs is in %o1, and P is in %o2).

```
EMULATOR DATA STRUCTURE:
....
      .word     xx_1
      .byte     6
      .byte     0
      .half     4
      .word     .LLC993
...

C_CHUNK:

xx_1:
      ld [%o1+8],%g2
      st %o0,[%o1+32]
      st %g2,[%o1+4]
      ld [%o2],%g2
      st %g2,[%o0]
      ld [%o1+20],%g2
      st %g2,[%o0+4]
```

[8] The main reason for our fixed-size format is that modern hardware is strongly word (32 or 64-bit) oriented and (usually) anything smaller implies some form of extra work, even when, for instance, byte-addressing is supported. With careful packaging of the instructions and their folded variants (see [25]), we are able to generate very compact code, as shown by code-size figures in section performance evaluation.

```
add %o0,8,%g2
st %g2,[%o0+8]
st %g2,[%o1+20]
....
add %o2,4,%o2
st %o2,[%o1-4]
retl
add %o0,36,%o0
```

It can be seen that the mapping to a sequence of load-store instructions with precomputed offsets gives efficient code, which can be reorganized quite freely by super-scalar schedulers. Since the C-functions do normally not contain calls to other procedures or functions, they will get compiled into efficient *leaf routines* which use a limited 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 for use by the specialized **retl** instruction.

3.7. Lessons learned

The effect of partial C-ification as presented in this section is especially interesting in the presence of

- long sequences of fine grained WAM instructions. The interpretative overhead can be largely reduced at the expense of the generation of a limited amount of extra code. Partial evaluation of logic programs will especially benefit from it as it creates long bodies;
- a top-down compilation scheme, which takes advantage of base addressing with small immediate offsets that are for free on most RISC-architectures;
- optimal register allocation as some pseudo-registers in the emulator become real ones in C.

Note that term compression [25] actually shortens instruction sequences in the body and as such it does not increase the *relative* speed-up of C-ified code w.r.t emulated code, while contributing to an increased performance for both, as it simplifies data movements in the functions and reduces memory references. This implies that an even larger speed-up is possible for emulators using a conventional data representation.

3.8. Built-ins and anti-calls

We have shown in the previous example the C-ification of `put`-sequences. This has been extended to frequently used inline operations which can be processed in Binary Prolog before calling the 'real goal' in the body as described in [12]. Chunks containing small built-ins that do not require a procedure call will still generate leaf routines.

On the other hand large built-ins implemented as macros in the emulator would make code size explode. Implementing them as functions to be called from the C-function would require code duplication and it would destroy the leaf routine discipline which is particularly rewarding on Sparcs. We have chosen to implement

them through an abstraction with a coroutining flavor: *anti-calls*. Note that calling a built-in from a C-function is operationally equivalent to the following sequence:

- return from the function,
- execute the built-in in the emulator (usually a macro),
- call a new leaf routine to resume the work left from the previous leaf routine.

Overall, anti-calls can be seen as a form of coroutining (jumping back and forth) between native and emulated code. Anti-calls can be implemented with the direct-jump technique used in WAMCC [6] even more efficiently, although for portability reasons we have chosen a conventional return/call sequence, which is still fairly efficient as a return/call costs the same as a call/return. Moreover, this allows the functions to remain leaf routines, while delegating overflow and signal handling to the emulator. Note that excessively small functions created as result of anti-calls are removed by an optimizing step of the compiler with the net result that such code will be completely left to the emulator. This is of course *more compact* and provable to be *not slower* than its fully C-expanded alternative.

4. Performance evaluation

The speed-up clearly depends on the amount of C-ification and on the statistical importance of C-ified code in the execution profile of a program (see Table 1). The benchmarks have been executed under Solaris 2.3 with the same C-compiler (`gcc 2.6.3`) and the same (`-O2`) level of optimization. For BinProlog, we have measured execution times for:

- basic emulator with no instruction folding[9] and no C-ification (emBP)
- optimized emulator with instruction folding (emBPo)
- basic emulator with C-ification (C-BP)
- optimized emulator with C-ification (C-BPo)

C-BPo gives the best overall performance, due to the synergy between the two optimizations and has been retained as the default mode of execution for BinProlog 3.xx and 4.xx. A similar speed-up (43%) is observed over the basic emulator (see C-BP vs. emBP) as over the emulator with instruction folding (see C-BPo vs. emBPo). Partial translation is clearly superior to instruction folding (see C-BP vs. emBPo) but their combined effect (C-BPo) is close to full C-translation (wamcc), although it is, as one might expect, slower than native code (natSP).

The main difference is however, that our approach is still fully portable, whereas the native code generation for SICStus Prolog is bound to a particular machine architecture. It is also quite remarkable that, on the average, our partial translation framework in synergy with emulator optimizations (C-BPo), is only 12% slower than WAMCC (full C-translation) which is an integer-only system and uses platform specific "asm" directives.

[9] Instruction folding consists of creating a large number of combinations of frequently occurring operation-codes with their associated C-code, all dispatched through a big switch statement or some direct branching techniques like gcc's first-order labels.

Bmark/Version	emBP	emBPo	C-BP	C-BPo	emSP	natSP	wamcc
NREV (KLIPS)	194	406	233	407	368	862	250
NREV (ms)	23370	11190	19490	11150	12340	5270	18180
CAL (ms)	600	510	350	290	760	310	191
FIBO (ms)	2380	2010	1500	1440	1970	910	714
TAK (ms)	1100	1140	830	760	580	170	315
SEMI3 (ms)	2280	2170	1710	1560	2270	1600	5791
QUEENS (ms)	5200	3510	3670	2270	3520	1080	1627
FKNIGHT (ms)	23490	19150	15070	12310	18150	4870	9040
geom. mean (ms)	3739	2941	2600	2054	2857	1113	1821
speedup	1.00	1.27	1.43	1.82	1.30	3.35	2.05

Table 1: Performance of emulated (emBP,emBPo) and partially C-ified BinProlog 4.05 (C-BP,C-BPo) compared to emulated (emSP), native (natSP) SICStus 2.1_9, and WAMCC 2.2 on a Sparc 10/20).

threshold:	0	4	8	12	20	30	1000
size SEMI3 (bytes)	32116	31204	27876	25092	14884	11764	11764
speed SEMI3 (ms)	1650	1580	1630	1650	1560	2030	2040
size CAL (bytes)	30444	14644	13356	12476	11828	11828	8756
speed CAL (ms)	300	390	290	380	400	410	480

Table 2: Size and speed of C-ified code w.r.t. threshold.

compiler	emBPo	C-BPo	emSP	natSP	wamcc
size SEMI3 (bytes)	6730	14884	22000	31900	44400
speed SEMI3 (ms)	2170	1560	2270	1600	5791
size CAL (bytes)	4880	13356	17100	21100	36500
speed CAL (ms)	510	290	760	310	191

Table 3: Size and speed w.r.t. compiler.

4.1. Fine-tuning the speed-size ratio

By allowing the programmer to specify that only sequences longer than a given *threshold* will get C-ified, the speed/size ratio of the resulting code can be empirically fine-tuned. For various parts of a project various thresholds can be applied. A maximum threshold is also available to avoid C-ifying large and seldomly executed blocks.

Table 2 shows some code-size/execution-speed variations with respect to the threshold for the `SEMI3` and `CAL` benchmarks. Clearly, excessively small functions can influence adversely not only on size but also on speed. Something like threshold=20, looks like a practical optimum for this program.

Table 3 shows that code-sizes for C-ified BinProlog executables (generated with a threshold of 20 for SEMI3 and 8 for CAL and dynamically linked on Sparcs with Solaris 2.3) are usually even smaller than 'compact' SICStus code which uses classical instruction folding (a few hundreds of opcodes) to speed-up the emulator, and considerably smaller than in the case of WAMCC.[10]

5. Further optimizations and extensions

5.1. 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 left to the emulator.

Applying this for the built-in `append/3` of BinProlog gave performances about 2 times faster than native SICStus Prolog 2.1_9 on the *naive reverse benchmark*. 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.

5.2. Decision graph indexing and two stream head compilation

Optimization of indexing using decision graph for unification instruction is an important optimization we plan to implement. Separate read and write streams are not yet implemented for head-unification (as their benefits are not very important for emulated code). However C-ified code as any native code compilation scheme would benefit from them.

5.3. Modules

5.3.1. C-based modules The Figure 1 shows a modular compilation proposal taking into account our C-ification process. A modular compilation scheme for C-translated Prolog is proposed in [9]. This can be applied quite naturally to

[10] For emulated BinProlog and emulated and native SICStus, sizes are as reported by the compilers. For C-ified BinProlog we have taken the size of the actual dynamically linked executable (a few K larger than the object file) and for WAMCC we have taken the size of the object file. Actual WAMCC files are about 200K larger as they are statically linked with the library.

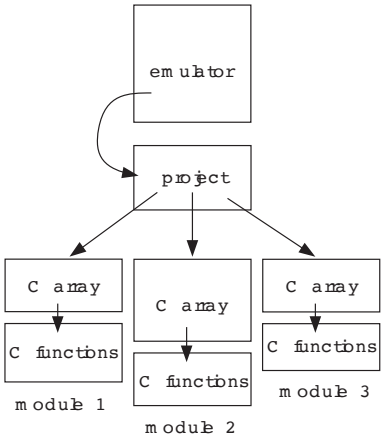


Figure 1: Module system.

our translation mechanism as the mapping to C described in [9] is also based on binary prolog. Modules are linked together in a *project* data structure accessible from the emulator at compile-time. Each module consists of an array of emulator instructions and a set of C-functions defined as local functions (i.e. declared *static* in C).

5.3.2. Source-level modules Modularity can also easily obtained by making emulated code to support modular compilation. BinProlog 3.30 implements a simple, program transformation based *source-level* module system [23]. Tight integration with the emulator ensures that source-level modules are automatically inherited by C-ified code. Future work is needed to add parametric modules and objects with a mapping to `C++`.

6. Related work

Simpler logic languages like Janus [15], KLIC [5] have been successfully compiled to C with speed which competes successfully with native code compilers.

The major difference with Prolog is that these languages do not feature backtracking, making the run-time system much simpler.

Surprisingly, for very high level logic languages like λ-Prolog good results have been obtained through compilation to C, [2] compared with previously used interpretative techniques. The resulting λ-Prolog MALI system [3, 19] supports C-based modules and typing. Although in terms of absolute performance the resulting code is still a few times slower than Prolog, due to the complexity of interpreted higher-order unifications and the lack of indexing, using compilation to C has been shown as an important step forward, by making λ-Prolog a practical language.

On the other hand, most of the attempts for the C-translation of full Prolog have managed to get performance often slower than well-engineered emulators like Quintus or SICStus Prolog, while paying the price of very large code-size or machine-specific non-portability [4, 17]. In [17], 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* and *gcc* specific) technique is used in the WAMCC system based on [6, 14] 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 or semi-ring` the overall performances are significantly lower than emulated BinProlog. Arithmetics is usually much faster but this comes in part from the fact that WAMCC does not yet support floating point operations which in our case need extra function calls. The WAMCC approach needs new assembler directives for each machine. Its use of explicitly allocated global registers conflicts with multi-threaded operating systems[11] like Solaris 2.x or Windows NT where performance scales for free through the use of multiple-CPU machines. For us this is a strong requirement with the multi-threaded execution model currently supported by BinProlog's Linda extension [8, 10].

A promising approach attributed to R. Mayer in [26] was at least partially tried out in the (yet unfinished) Half-Life compiler. The idea is to use C as a portable high level assembler (through a sequence of macros generating very simple C instructions). Although excellent for speed and portability, this is unlikely to avoid the code-size explosion problem.

Previous approaches have often circumvented the theoretically challenging problem of giving an efficient mapping from a non-deterministic logic programming language to C-like procedural languages 'as they are', with relatively slow function calls, and their own way to handle recursion with absent or only partial last call optimization.

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 have obtained overall performances comparable to the best full-translation schemes. The resulting code is portable and based on a non-trivial mapping from Prolog to C, through program transformation and continuation passing techniques. Our method, which needs very little implementation effort, 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.

Our scheme can be seen as a partial evaluation with respect to a known program, of the *instruction-folding* technique, often used to speed-up emulators.

The idea itself of partial translation has been advocated independently in [18] for purpose of hardware simulation, and prototyped for the Dhrystone program. Surprisingly, a lot of common techniques and motivations are shared between this work and ours, despite their very different objectives and implementation techniques.

[11] Global data that permanently allocated in registers become inaccessible by the other processes.

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 performances in a range between emulated and native code with little implementation effort and ensures portability through C.

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. We thank the anonymous referees for their careful reading and useful comments and suggestions.

The code shown in this paper has been generated with BinProlog 3.30, available by `ftp` from *clement.info.umoncton.ca*, together with the benchmarks and some related papers.

References

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

[2] P. Brisset and O. Ridoux. The compilation of λProlog and its execution with MALI. Technical Report 687, IRISA, 1992. ftp://ftp.irisa.fr/local/lande.

[3] P. Brisset and O. Ridoux. The architecture of an implementation of λProlog: Prolog/Mali. In K. De Bosschere, B. Demoen, and P. Tarau, editors, *ILPS'94 Workshop on Implementation Techniques for Logic Programming Languages*, 1994. ftp://ftp.irisa.fr/local/pm.

[4] A. Cheyers, Apr. 1993. Personal Communication.

[5] 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.

[6] P. Codognet and D. Diaz. Compiling Prolog to C : the WAMCC system. In *Proceedings of the 12-th International Conference on Logic Programming*, Yokohama, Japan, 1995. The MIT Press.

[7] K. De Bosschere, S. Debray, D. Gudeman, and S. Kannan. Call Forwarding: A Simple Interprocedural Optimization Technique for Dynamically Typed Languages. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 409–420, Portland/USA, Jan. 1994. ACM.

[8] K. De Bosschere and J.-M. Jacquet. Multi-Prolog: Definition, Operational Semantics and Implementation. In *Proceedings of the ICLP'93 conference*, Budapest, Hungary, 1993.

[9] 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.

[10] K. De Bosschere and P. Tarau. Blackboard-based Extensions in Prolog. *Software — Practice and Experience*, 26(1):49–69, Jan. 1996.

[11] S. Debray, K. De Bosschere, and D. Gudeman. Call Forwarding: A Simple Low-Level Code Optimization Technique. In E. Tick and G. Succi, editors, *Implementations of Logic Programming Systems*, pages 77–88. Kluwer Academic Publishers, 1994.

[12] 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.

[13] B. Demoen and G. Maris. A comparison of some schemes for translating logic to C. Technical Report 188, K.U.Leuven, Mar. 1994. presented at the Workshop on parallel and data parallel execution of logic programs, ICLP94.

[14] 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.

[15] 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.

[16] B. Hausman. Turbo Erlang: Approaching the Speed of C. In *Implementations of Logic Programming Systems*, pages 119–135. Kluwer Academic Publishers, 1994.

[17] R. Horspool and M. Levy. Translator-based multiparadigm programming. *Journal of Systems and Software*, 25(1):39–49, 1993.

[18] P. Magnusson. Partial translation. Technical Report 05, SICS, 1993.

[19] O. Ridoux. MALIv06: Tutorial and reference manual. Publication Interne 611, IRISA, 1991. ftp://ftp.irisa.fr/local/lande.

[20] 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.

[21] 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.

[22] 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.

[23] P. Tarau. BinProlog 3.30 User Guide. Technical Report 95-1, Département d'Informatique, Université de Moncton, Feb. 1995. Available by ftp from *clement.info.umoncton.ca*.

[24] 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.

[25] P. Tarau and U. Neumerkel. A Novel Term Compression Scheme and Data Representation in the BinWAM. In J. P. M. Hermeneguildo, editor, *Proceedings of Programming Language Implementation and Logic Programming*, Lecture Notes in Computer Science 844, pages 73–87. Springer, Sept. 1994.

[26] P. Van Roy. Issues in implementing logic languages. Technical report, WWW hypertext document, 1994. http://ps-www.dfki.uni-sb.de/ vanroy.