Translating Prolog to C: a WAM-based approach

M.R. Levy and R.N. Horspool
Department of Computer Science, University of Victoria
P.O. Box 3055, Victoria, BC, Canada V8W 3P6

E-mail: mlevy@csr.uvic.ca, nigelh@csr.uvic.ca

1 Introduction

A translator from Prolog to C (or C++) can be used as a very effective tool for performing multi-paradigm programming. In [4], we have argued in favour of translation-based multi-paradigm programming. We illustrated how C++ and Prolog could be used advantageously together to build a demonstration compiler. We are now directing our efforts to improving the quality of the translation.

Many Prolog systems do allow the use of "foreign" code. For example, SICSTUS Prolog allows C modules to be dynamically loaded and then executed. However the translator based approach offers certain advantages, including portability plus a greater degree of control of the overall system. The question we wanted to address was this: Is C an appropriate target for Prolog compilation? By appropriate, we mean that the resultant code is competitive in speed and space with the best-known Prolog compilation strategies. Furthermore, because the objective is to use the translated code in larger systems, we certainly require the ability to create separate modules, each containing the C code equivalent of one or more Prolog predicates.

Although C is sometimes regarded as a high-level assembler language, its implicit run-time structure is not compatible with efficient implementation of Prolog's backtracking. In this paper, we report on experiments we have conducted on translating Prolog to C using a WAM-based approach[5, 1]. There are a few advantages to using WAM, the main one being that we can benefit from a fairly extensive amount of research that has been done on WAM-based Prolog optimizations, both for storage and execution speed.

We will show that it is possible to produce C code whose execution time lies somewhere between byte-code compiled Prolog and native-code compiled Prolog. (Byte-code is something of a misnomer: it refers to the approach that encodes the WAM version of a program in a data array, and then interpretively executes the contents of the array. A native-code compiler translates predicates into the machine instructions of a target computer.)

2 Implementing the WAM in C

Warren's abstract machine (the WAM), was first described in [5]. That report was not designed to be tutorial, and several expository articles have subsequently appeared. We used Aït-Kaci's excellent tutorial [1]. One pleasing aspect of this book is that it presents the implementation of each of the WAM operations in the form of a Pascal-like procedure. For example, put-variable is defined as follows:

```
HEAP[H] <- <REF,H>;
Xn <- HEAP[H];
Ai <- HEAP[H];
H <- H+1;
P <- P + instruction_size(P);
```

At first glance this would suggest that a Prolog predicate could be translated by compiling it to WAM, and then representing each WAM operation as a call to the corresponding C function. Consider, for example, the Prolog predicate

```
p(X,Y) :- q(X),r(Y).
```

The predicate compiles to the following WAM code (assuming that this is the only clause):

```
p/2:    get_variable A2,X3
        call q/1
        put_value X3,A1
        execute r/1
```

(The obvious optimization has been done to eliminate a redundant get/put sequence for A1). It would be tempting to translate this to the following C code:

```
void p_2(void){
        get_variable(A2,X3);
        q_1();
        put_value(X3,A1);
        r_1();
}
```

The problem is, what if either of the calls to q_1 or r_1 fails? In WAM, detection of failure invokes a call to the backtrack operation, which selects the next control point from a stack of choice points. If q_1 fails, return will not necessarily be to p_2's caller, so it is not sufficient, for example, to insert a test for failure after the call to q_1 in the C code. In Aït-Kaci's book, detection of failure results in a call to a procedure called backtrack. The code for backtrack modifies the program counter P. However the only suitable mechanism that C provides takes the form of the setjmp/longjmp library functions – but these are too inefficient for use in the Prolog system.

One possible strategy is to use continuations [2, 6]. That is, each function takes as its last argument the pointer to a continuation that must be called as the last action of the function. Although this can be used as the basis for translation, it leads to unacceptable stack growth. The growth is especially severe when recursive calls are executed, mainly because last-call optimization is impossible to implement as C statements if more than one predicate is involved. Furthermore, the C programming language (unlike Scheme, for example) provides no efficient mechanism for creating continuations.

We have used an approach that is an extension of the (incorrect) approach that translates WAM operations to calls to C functions. We use a dispatching loop to monitor the call sequence, and we have modified the WAM design (as presented in Aït-Kaci) as follows: Firstly, P and CP are defined as pointers to C functions. Secondly, we have added two registers which we have called W and CW. P and CP are used to select C functions, whereas W and CW are used within C functions (that is, to select code within a function). The complete details will appear in a full version of this paper.

Our translator is built on top of a traditional WAM compiler. A Prolog program translates Prolog to WAM, performs various optimizations (such as the register optimization given above), and generates a WAM program in a list. This is then translated (by another Prolog predicate) into C code. Our translator can produce fast or compact code. Fast code is produced by generating macro calls wheras compact code generates function calls for the WAM operations. (The fast code is larger than the compact code because each call is expanded to possibly several C statements). The WAM data structures (Heap, Stack and Trail) are implemented using C arrays.

To achieve acceptable performance in the fast code, very careful attention must be paid to the design of the macros. For example, we have exploited a technique that we gleaned by studying SICSTUS Prolog that eliminates multiple tests of WAM's read/write flag in code sequences for building or matching structures on the heap. We have also been careful to eliminate unnecessary pointer dereferencing.

Table 1: Benchmark results

Machine/system	nrev	queens
Sun SPARC:		
SICSTUS byte code:	2.92secs(177Klips)	1.44secs
SICSTUS native code:	0.75sec(687Klips)	0.44secs
Translated to C compact:	2.77secs(187Klips)	0.77secs
Translated to C fast:	1.87secs(277Klips)	0.69secs
Macintosh IIci:		
BNR Prolog interpreted:	96.8secs (5.3Klips)	63.5secs
Open Prolog:	101secs (5Klips)	40.0secs
Translated to C compact:	33secs (15.6Klips)	13.7secs
Translated to C fast:	18.4secs (28Klips)	10.2secs

The size of the compact code can be further reduced without much effect on its speed by replacing repeated sequences of operations with calls to specially created new functions. This is obviously a form of data compression. We have implemented certain specific compressions, and are currently experimenting with a general strategy based on "tupling" [3].

3 Benchmark results

We have compared our system with SICSTUS Prolog. We used SICSTUS Prolog because it can compile either byte-code or native code, it is WAM-based and the source code is available. We have also run the (identical) C code on a Macintosh. We were unable to find a compiled Prolog on the Mac, so we used BNR Prolog and Open Prolog for comparison. As far as we can tell, both are interpreters, although the Open Prolog documentation claims that it does perform last-call optimization and indexing.

There are many well-known pitfalls to benchmarking. One problem is comparing like with like. For example, it is reputed (and not denied) that a certain well-known Prolog system detects the definition of append (or any predicate defined equivalently!) and "compiles" specially written, hand optimized code for it. Such a strategy will drastically improve the performance of one of the standard benchmarks used for comparing Prolog systems, namely the naive-reverse benchmark.

TABLE 1 shows the results of our benchmarks on a Sun 4/390 SPARC server, and on a (vanilla) Apple Macintosh IIci. Note that we are not able to run SICSTUS code on a Mac, whereas our translated code is written in ANSI C, and hence is portable to any system that provides an ANSI-C compiler. On the Mac we used SYMANTEC's THINK C, v 5.0. We ran naïve reverse 100 times using a list of length 100. As can be seen in the table, our compact code slightly outperforms SICSTUS byte code compiled programs on the SPARC architecture. Our fast code is about 50% faster, making it about 40% slower than native-code compiled SICSTUS code.

4 Factors affecting perfomance

We found that performance is very dependent on the way in which the WAM structures are defined in C. In particular, on machines with 32-bit data widths (such as the SPARC and Motorola 68020), it is important to keep the size of heap elements to one 32-bit word. Clearly, the quality of code generated by the C compiler will significantly affect performance. The Gnu C compiler, for example, compiles our C code into object code that runs 24% faster than the code generated by Sun ANSI-C compiler, acc. The use of local variables to hold copies of global variables noticeably improves performance on the SPARC architecture. This occurs because, on the SPARC architecture, it takes two instructions to fetch a value from global memory, whereas most local variables are held in registers and are accessible in one clock cycle. For the purposes of supporting WAM, we would certainly be happy if it was possible to declare in C that certain global variables should be permanently allocated to registers.

5 Conclusions

C is a reasonable target for Prolog compilation. Translated Prolog programs can easily be combined with code written in other programming languages. Furthermore, the translated code is portable (as is our translator).

Further work needs to be done to develop tools that simplify multi-paradigm programming. For example, the C++ translator described in our earlier work uses C++ class libraries to give a high-level abstract interface between C++ and Prolog. We intend to provide a similar C++ interface for the new translator.

References

[1] Hassan Aït-Kaci. *Warren's Abstract Machine.* MIT Press, 1991.

[2] J.L. Boyd and G.M. Karam. Prolog in C. Technical report, Carleton University, Ottawa, 1988.

[3] Eric C. R. Hehner. *Matching Program and Data Representations to a Computing Environment.* PhD thesis, University of Toronto, 1974.

[4] M.R. Levy and R.N. Hosrpool. Translator-based multi-paradigm programming. *Journal of Software and Systems*, 1993.

[5] David H.D. Warren. An abstract Prolog instruction set. Technical Report Technical Note 309, SRI International, 1983.

[6] J.L. Weiner and S.Ramakrishnan. A piggy-back compiler for Prolog. In *Proc. of SIGPLAN'88 Conf. on Prog. Lang. Design and Implementation*, pages 288–296, 1988. Atlanta, Georgia.