wamcc: Compiling Prolog to C

Philippe Codognet and **Daniel Diaz**
INRIA-Rocquencourt
Domaine de Voluceau
78153 Le Chesnay
FRANCE
{Philippe.Codognet, Daniel.Diaz}@inria.fr

Abstract

We present the wamcc system, a Prolog compiler that translates Prolog to C via the WAM. This approach has some interesting consequences: simplicity, efficiency, portability, extensibility and modularity. Indeed the system does not incorporate complex optimizations but is as efficient as Quintus Prolog 2.5 (based on an emulator written in assembly language) and only 30 % slower than Sicstus Prolog (compiling to native code). It is instantly portable to any machine with a C compiler and easily expandable with various extensions, such as constraints, as shown by the clp(FD) system which is based on wamcc. It also incorporates a simple but efficient handling of modularity thanks to the use of C modularity.

1 Introduction

Compiling Prolog is now a well-known task and no more as challenging as ten years ago, when the design of high-speed Prolog systems was not completely assessed. One of the major breakthroughs of the last decade in Logic Programming has arguably been the definition of the Warren Abstract Machine (WAM) [14] which became a de facto standard for the compilation of Prolog and has helped many researchers to gain a better understanding of Prolog's execution and to develop efficient LP systems. Moreover the WAM proved to be flexible enough to remain the backbone of various extensions such as Higher-Order, constraints, parallel or concurrent LP. However the design of simple and yet efficient Prolog systems is still an active topic.

We started to develop our own logical engine in 1991, in order to provide a sound basis for various extensions and in particular Constraint Logic Programming [6]. The requirements for this new system were as follows:

extensibility: the system should be used as an experimental platform. This implies simplicity in the design of the Prolog engine without complex optimizations that would excessively enlarge the size of the system.

portability: in order to achieve a wide diffusion and avoid the obsoleteness inherent to the fondness for a particular architecture.

efficiency: to avoid handicapping future extensions because of insufficient performances of the underlying Prolog Systems. A new language is indeed often first judged on the (counter) performances it has on classical benchmarks and only second on the novelty and expressiveness it offers. Therefore the objective for our logical engine was to be as efficient as the emulated version of Sicstus Prolog, which is based on a highly optimized C emulator.

modularity: to decompose a Prolog application into several modules. This also make it possible in a first step to forget about built-in predicates, that will be written in Prolog later in separate modules.

Classical techniques consist in compiling Prolog to a WAM-like instruction set and then either executing directly the WAM code with an emulator written in C (original version of Sicstus Prolog) or assembler (Quintus Prolog) or compiling to native code (Prolog by BIM, latest version of Sicstus Prolog, Aquarius Prolog).

Nevertheless, neither emulation nor compilation to native code satisfy our initial requirements: emulation has poor performances if unoptimized or great complexity if optimized while the production of native code is certainly not a simple task and is not portable. We thus decided to investigate a new alternative: the translation of Prolog to C. Indeed, this solution merges, in theory at least, the advantages of both approaches. Portability is ensured by the worldwide hegemony of the C language and native code will be produced at the end of the compilation process (by the C compiler). This avoids the overhead of emulatros and therefore should compensate from the lack of optimizations. The modularity of the C language makes it possible to compile Prolog modules as separate C modules that can be linked together by the linker (predicate visibility being coped through appropriate C declarations). Compiling to C also makes it possible for a simple interface with this (and any other compiled) language and produces real executable files as outputs of the logical system. The resulting system, named `wamcc`, proves that this approach is sound and that a system translating Prolog to C can be as efficient as an optimized commercial Prolog.

The paper is organized as follows. Section 2 presents basic ideas about the execution of WAM code. Section 3 describes how other existing systems compile WAM code to C while section 4 presents the `wamcc` approach. Section 5 details how the memory is managed in `wamcc` and, finally, section 6 presents performance evaluations. A short conclusion ends the paper.

2 Below the WAM

For a few decades compilers for imperative, functional or logical languages decompose the compilation process in several steps and in particular use an abstract machine as an intermediate level between the high-level source code and the target low-level executable code. Indeed, since Pascal and the P-code, abstract machines have been highlighted as the backbone of the compilation process. Logical languages are no exception here and the compilation of Prolog to WAM instructions is a de facto standard and well-known process. However, WAM code cannot be executed directly on mainstream computers and need therefore some treatment in order to be made executable. There exists classically two methods to execute WAM code: a WAM emulator or a translator to native code.

Emulating the WAM code is usually the first solution that comes to mind. In this approach, WAM instructions (byte-code) are simply considered as data that will be treated by some emulator program, usually written in a language such as C or assembler. The advantages of an emulator lie in its simplicity in writing (around 3000 lines of C code for an unoptimized version), its portability when it is written in a general language such such as C, and its ability to create and dynamically modify WAM code. These advantages make it the main trend among Prolog developers. However this approach has a major drawback in the overhead of the emulation, that is in the cycle: *fetching, decoding* and *executing.* Another drawback is that it is not possible to produce an autonomous executable program as output, because the emulator has to be present.

Producing native code has characteristics that are roughly opposite to that of emulation. Writing a translator from abstract code to machine code is a difficult task requiring a deep knowledge of the target machine. Good performances are at this price, especially on RISC architectures where optimized code is essential. Such compilers are not easy to port on new architectures: some aspect for which a machine is particularly good at could possibly be the Achilles' heel of another machine. On the other side, developing a native code compiler consists of many classical parts, maybe painful to develop but for which adequate techniques are well-known, e.g. register allocation.

Since none of these methods satisfied our requirements we decided to try another approach consisting in a translation of Prolog to C. The idea was to compensate the lack of optimizations (required for simplicity) by the absence of emulation overhead phases (fetching and decoding) since, finally, the C compiler would produce native code. Doing this we would combine the advantages of emulation and compilation to native code. However no such experience had been done when we decided to develop our system[1], back in 1991. Even today, three years later, no implementation of Prolog based on the compilation to C is available outside our own `wamcc` system. However in the meanwhile several implementations of deterministic committed-choice languages have opted for this approach: Janus [7], KL1 [3] and Erlang [8]. They are all based on a different compilation scheme, and it is interesting to compare them with `wamcc` for deterministic Prolog.

3 Existing Logic Programming translators

We will detail in this section how Janus, KL1, Erlang and `wamcc` handle the control flow. This presentation is inspired from [5] which uses a goal stacking model. However, we do not follow the same abstraction to be closer to what is really implemented. As a consequence of this choice, the C code associated to WAM instructions is explicitly described. Due to space limitations we only adress here the problem of the control. This is motivated first by the fact the WAM used by `wamcc` is a conventional WAM without any optimizations and thus the code for other instructions is now well-known [1]. Second, doding efficiently the control is crucial in the translation to C because the WAM code is flat and execution transfers are done by branchings. This is different from C which is more suited for high-level control structures such as functions and does not provide much for low-level control. Therefore the main problem will be to find an adequate solution for translating WAM branchings. Our presentation will be based on the following example, consisting in only one clause and one fact:

```
p:  allocate     /* p:- q, r.  */
    call(q)
    deallocate
    execute(r)

q:  proceed      /* q.  */
```

[1] although there has been an unsuccessful experiment at ECRC at the end of the 80s.

This simple example nevertheless shows all the instructions used for the control of Prolog in the deterministic case. The manner to translate `call` and `execute` will in particular highlights how to manage direct branchings (i.e. when the target address is a known label), whereas the translation of the `proceed` instruction has to solve the problem of indirect branchings (i.e. when the target address is the content of some variable, the `CP` register in this case).

3.1 Janus

The implementation of Janus is based on the straightforward idea of translating a WAM branching by a C branching, i.e. a `goto` instruction. A similar method has been used in the Prolog compiler described in [11]. However problems arise because indirect branching is not available in standard (ANSI) C (and must therefore be simulated) and also because `goto` instructions can only address code within the same function. The solution therefore leads to a C program consisting of a unique function with a `switch` instruction to simulate indirect `goto`s. Following this method, our previous example will be translated to:

```
fct_switch()
{
 label_switch:
   switch(PC) {
      case p:                 /* p:- q,r .  */
      label_p:
         push(CP);            /* allocate */
         CP=p1;               /* call(q) */
         goto label_q;        /*   :  *.

      case p1:
         pop(CP);             /* deallocate */
         goto label_r;        /* execute(r) */

      case q:                 /* q.  */
      label_q:
         PC=CP;               /* proceed */
         goto label_switch;   /*   :  */
      ⋮
   }
}
```

This method is expensive on RISC machines since a switch-statement costs about 10 machine instructions (including bound checks). However, the major drawback of this approach is that a program gives rise to a single function. Hence, except for toy examples, it generates an enormous function that the C compiler is not able to treat in a reasonable time. Coping with modularity is not easy in this setting, as it requires consulting a dynamic table for each predicate call in order to pass the control to the `switch` function of this module. Moreover, to support full Prolog, one has also to take care to correctly handle backtracking in case of context changes. Supporting modularity is therefore penalizing, and an extra-module call will be much more costly than an intra-module call.

3.2 KL1

As the compilation to a single C function is unrealistic, the C program representing the WAM code has to be sliced into several functions. Translating each Prolog predicate into a C function seems therefore natural. WAM branchings will give rise to function calls. Such a function will call another nested function (branching) before returning, and so on; in such a way in fact that it will never return before the end of the program. The data accumulated in the C control stack are therefore useless and can lead to a memory overflow. The solution is thus to return from any function before executing a branching, and to have a *supervisor* process in charge of branching to the adequate continuation. This leads to the following code for our continuing example:

```
fct_supervisor()
{
   while(PC)
      (*PC)();
}

void fct_p()       /* p:- q,r.  */
{
   push(CP);       /* allocate */
   CP=fct_p1;      /* call(q) */
   PC=fct_q;       /*   :  */
}

void fct_p1()
{
   pop(CP);        /* deallocate */
   PC=fct_r;       /* execute(r) */
}

void fct_q()       /* q.  */
{
   PC=CP;          /* proceed */
}
```

The code depicted above can be optimized by suppressing the PC register since its information can be returned by the functions. Thus, each function realizes in-line computations and terminates returning the address where the control has to be passed when a branching is required. The analysis of this method shows that a WAM branching is implemented by a return to the supervisor followed by a function call. This is obviously much more costly than the simple jump which would be generated by a native code compiler. However extra-module calls are now possible without extra cost. The first implementation of wamcc used this technique and was about twice as slow as emulated Sicstus. KL1 opted for a trade-off in order to reduce function calls and returns: all predicates within the same module are translated into a single function. Hence when only one module is used, KL1 behaves like Janus. The supervisor function is only needed for context switching on extra-module calls, which are therefore more costly than intra-module calls.

Let us conclude by remarking that this method (with or without the improvement proposed in KL1) is the most suited for a 100% ANSI C solution.

3.3 Erlang

In Erlang also a predicate is translated into a C function. However, in order to avoid the overhead of function calls and returns, Erlang takes advantage of the new possibilities offered by the GNU C compiler (gcc). Indeed gcc considers (jump) labels as first class objects and makes it possible to store a label in a pointer variable and to make in the subsequent execution an indirect jump to the value pointed by such a variable. The idea thus consists in translating a WAM branching to an indirect jump going *inside* a C function, in order to avoid the extra cost of the call. A global table storing all addresses is then needed, which must be initialized by a first call to each function. Back to our unavoidable example, this produces:

```
void fct_p()                /* p:- q,r.  */
{
   jmp_tbl[p]=&&label_p;    /* (initialization) */
   jmp_tbl[p1]=&&label_p1;
   return;

 label_p:
   push(CP);                /* allocate */
   CP=&&label_p1;           /* call(q) */
   goto *jmp_tbl[q];        /*   :  */

 label_p1:
   pop(CP);                 /* deallocate */
   goto *jmp_tbl[r];        /* execute(r) */
}

void fct_q()                /* q.  */
{
   jmp_tbl[q]=&&label_q;    /* (initialization) */
   return;

 label_q:
   goto *CP;                /* proceed */
}
```

All branchings are done by indirect `goto` through a global address table. In order to eliminate the cost of this indirection for direct jumps, Erlang operates as KL1 or Janus and compiles into a single function all the predicates of a given module. Hence only extra-module calls need the consultation of the global address table, and would therefore be more costly that intra-module calls.

Observe that branching directly inside a function and avoiding the *prologue* makes it impossible to use local variables (no room is reserved in the C stack) and thus implies to only use global variables. Note also that any instruction must not be moved before the entry labels, and this is quite difficult to guarantee. Let us consider the access to an element of the global table. This is compiled into a load of the address of the table followed by an instruction for accessing the given element. The compiler can take the liberty to optimize the table accesses and to place the loading of the table address at the very beginning of the function, where it assumes that it will always be executed. This would cause a problem when a jump inside the function will happen and will try to use the uninitialized register.

4 The wamcc approach

The three methods proposed above have all in common to behave similarly within a single module, giving rise to a single large function that the C compiler has much pain to compile. Extra-module calls are, if they are possible, more costly than intra-module calls. Hence the manner in which a program is decomposed in modules influences not only compilation time but also execution time, in inverse proportions evidently.

The objective of the second version of our wamcc system was to translate a WAM branching into a native code jump. As the decomposition into several functions is mandatory, these jumps should reach blocks of code inside a function. To produce direct branchings we have to be able to determine labels *statically* (at compile time) rather than *dynamically* (at execution time). The pair "compiler + linker" is well suited to do this for the addresses of functions. The solution adopted in wamcc is then to insert a label at the entry of each function thanks to asm(...) directives. To manipulate the addresses of those labels, say L, one just need to fool the compiler and let him believe that L is an external function by declaring a prototype for the function L and by using the symbol L (in C the name of a function is its address). The compiler then generates an instruction with a hole that will be filled in by the linker with the knowledge of all inserted labels (internal and external). The cost of an extra-module call is then exactly the same as that of an intra-module call. To finish with our favorite example, the code produced will be:

```
void label_p();               /* prototypes */
void label_p1();
void label_q();
void label_r();

#define Direct_Goto(lab)      lab()
#define Indirect_Goto(p_lab)  (*p_lab)()

void fct_p()                  /* p:- q,r.  */
{
 asm("label_p:");
   push(CP);                  /* allocate */
   CP=label_p1;               /* call(q) */
   Direct_Goto(label_q);      /*   :  */
}

void fct_p1()
{
 asm("label_p1:");
   pop(CP);                   /* deallocate */
   Direct_Goto(label_r);      /* execute(r) */
}

void fct_q()                  /* q.  */
{
 asm("label_q:");
   Indirect_Goto(CP);         /* proceed */
}
```

Only two macros are needed to realize branching, direct or indirect, and they depend on the machine architecture. For instance on a RISC machine, we have:

- `Direct_Goto(lab)` for simply invoquing the `lab` function.
- `Indirect_Goto(p_lab)` for invoquing a function whose name (address) would be stored in `p_lab`.

Indeed on a RISC machine the instruction for a function call passes the control to the given address (as a jump) and initializes the continuation pointer of the processor. Because of the RISC architecture, this instruction is as fast as a simple jump. As nothing is stacked, it can be used for branching (the fact that the continuation pointer is updated is without important because we know that it is not a real function call but a mere jump). Doing this way avoids the need for inserting jump instructions in the assembly code. Moreover, RISC branch instructions can only access code relatively close to the current instruction whereas the function call instruction does not follow this limitation and this is indeed needed for accessing code potentially far because of the module decomposition. Let us finally note that leaving the generation of the function call to the C compiler allows it to optimize the *delay slot* to take advantage of the instruction pipeline[2]. To sum up:

- direct jumps are executed as fast as possible because they are translated into native code jumps (or in case of RISC architectures into function calls of equivalent cost).
- extra-module calls are not more costly than intra-module calls.
- compared to previous approaches where all the predicates of a single module were compiled into a single function, this method gives rise to as many functions as they are goals in the body of a clause (head and first goal counting one). The code produced is therefore compiled faster (see later).
- each function has only one direct entry point at the very beginning, thus only the prologue is jumped over. To allow for local variables, some (big enough) space is reserved in the C control stack before starting the computation by defining an array in an intermediate function. Thus the C stack pointer `sp` points to the end of the array. Local variables will be allocated within this array (see below).
- the only assumption underlying this approach is therefore that the prologue does nothing but decrement `sp`. This is generally the case except for a few machines where the C compiler does not reference local variables through `sp` but through another `fp` register (frame pointer) that will be set at the function entry to `sp`. This is intended to help the debugger and it is usually possible to deactivate this operation by a compiler option. In the case this is not possible one can always generate an `asm` directive to initialize this `fp` register.
- it is possible to have real function calls inside these pseudo-functions. In particular most of the macros associated to WAM instructions are expanded into call to the `wamcc` library. This makes it possible to favorize code size (and compilation speed) to the (small) detriment of execution speed.

[2] on some RISC processors the instruction immediately following a jump or function call (*delay slot*) is always executed because it is already in the pipeline. Compilers try to use this particularity by moving a pertinent instruction after the branching. When this is not possible, a `nop` instruction is generated.

Let us now detail the code needed for starting the computation as described above. Suppose that the first predicate (usually a top-level) is at address `p_lab`:

```
#include <setjmp.h>

jmp_buf jumper;

void Label_Success();
void Label_Fail();

Bool Call_Prolog(WamCont p_lab)
{
 Create_Choice_Point();
 ALTB(B)=Label_Fail;
 CP=Label_Success;

 ret_val=setjmp(jumper);
 if (ret_val==0)
     Call_Next(p_lab);

 Delete_Choice_Point();
 return ret_val==2;
}
```

```
void Call_Next(WamCont p_lab)
{
 int t[1024];

 Indirect_Goto(p_lab);
}

void Call_Prolog_Success(void)
{
 asm("Label_Success:");

 longjmp(jumper,2);
}

void Call_Prolog_Fail(void)
{
 asm("Label_Fail:");

 longjmp(jumper,3);
}
```

The `Call_Prolog` function has to execute the predicate whose address is `p_lab`. It starts creating a choice point in order to record the address to branch to in case of failure (`Label_Fail`). The `CP`, indicating which code to execute after the predicate success, is initialized with `Label_Success`. Finally a `setjmp` is executed in order to allow later return to the instruction after the `setjmp`. The `Call_Next` function is called for reserving enough space in the C stack for possible local variables (c.f. the declaration of array `t`). The control is then given to the predicate, that will execute as detailled previously. In case of success (resp. failure), the control is transferred to `Label_Success` (resp. `Label_Failure`) that will simply return to the `Call_Prolog` function by a `longjmp` with the second parameter set to the value 2 (resp. 3).

5 Memory management

We just recall here that the WAM memory management consists in using three stacks: the Local Stack for control blocks and local variable, the Heap for data structures, and the Trail for storing bindings to undo upon backtracking.

It is mandatory to control the growth of stacks and to alert the user in case of overflow. This is usually done by incorporating software tests either at each memory allocation (potentially several times for each clause for the Heap) or at each clause entry (checking all stacks) or thanks to new WAM-like instructions. In any case this control is costly, all the more because basically current machine architectures allow for hardware tests. Indeed machines use virtual memory, meaning that the user does not have to bother about physical addresses and real memory size, and provide, logically if not physically, very big linear memory (e.g. 4 GBytes on 32 bits architectures). When some data must be accessed, the memory manager detects if the memory *page* to which it actually belongs is physically present in memory or not (*page default*). In the later case, the memory manager loads it in memory after

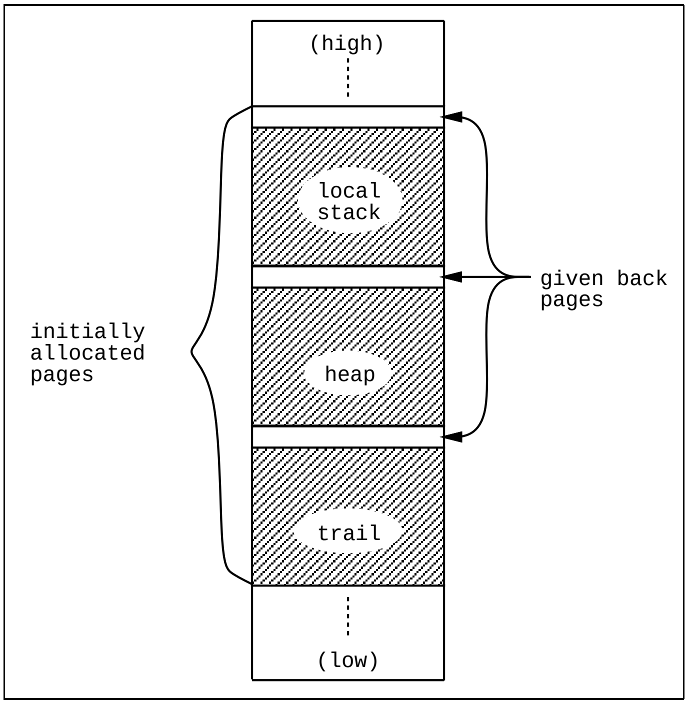

Figure 1: memory allocation

swapping another page on disk if necessary. Interestingly, the memory manager raises an *exception signal* when a page default refers to an un-allocated (i.e. free) page. The idea is therefore to have such a signal raised in case of stack overflow. To ensure this we only have to free (i.e. to give back) each page following a stack (see figure 1). When an attempt to read/write in this page occurs the signal triggered is caught by a C function (*handler*) responsible of diagnosing the overflow (checking top stack pointers) and of generating the adequate error message. The easiest way to implement this scheme is to use the Unix `mmap` function which makes it possible to map a file to a part of memory. All pages of this part are initially marked as “swapped” on the corresponding page of the file. Then, readings and writings on this file are then done by simply reading and writing the memory. There usually exists a special device (`/dev/zero`) returning zero on initial readings and on which writings are not reflected. This device is then well suited for our stacks since only memory operations are performed. Thanks to the `munmap` function, each page following a stack is given back to the memory manager. In case of the absence of `mmap` functions on a particular machine, it is possible to use those handling the sharing of memory between processes (`shmget`, etc), as they also make it possible to get and free back memory. Finally, for machines without even those functions, `wamcc` uses the standard C memory allocator (`malloc`) and performs software tests to check overflows.

6 Performance Evaluation

Let us now detail the performances of the wamcc system and compare them with that of other Prolog systems either academic or commercial.

6.1 Benchmark programs

Table 1 presents the performances of wamcc on classical set of benchmarks. Timings are in seconds measured on a Sparc 2 (28.5 Mips) using gcc 2.5.8 with the -O2 option. For each program, one can find: the number of lines of the Prolog source program, the total compilation time (Prolog to C, gcc, linker), the size of the object code and of the final executable (KBytes) and the execution time.

Program	Lines	compilation time	object size	executable size	execution time
boyer	395	64.0	48	240	3.450
browse	111	21.0	12	208	4.020
cal	202	18.0	12	208	0.300
chat_parser	1184	290.0	132	328	0.980
crypt	96	16.0	10	200	0.016
ham	90	14.0	10	200	4.330
meta_qsort	146	18.0	10	200	0.045
nand	574	202.0	76	264	0.120
nrev	105	9.0	5	200	0.600
poly_10	112	16.0	11	200	0.300
queens (16)	95	7.0	3	192	2.440
queens_n (8)	79	9.0	5	200	0.700
queens_n (10)	79	9.0	5	200	13.680
reducer	388	50.0	37	232	0.270
sdda	327	32.0	23	216	0.015
sendmore	66	12.0	8	200	0.230
tak	35	5.0	2	192	0.550
zebra	57	9.0	6	200	0.260

Table 1: Performances of wamcc

6.2 wamcc versus academic Prolog systems

Let us in a first time compare wamcc with other systems in the same category: academic Prolog systems, usually developed for research purposes by a single person and freely distributed by ftp. Among the number of available such systems, we have chosen only the most popular ones. We thus have:

BinProlog 3.0: this implementation is based on the "binarization" of clauses, which roughly consists in making the continuations explicit. The WAM is specialized and the abstract code is emulated (by an emulator written in C). Recently the author has also investigated a translation to C [12].

XSB-Prolog 1.4.0: this language is the follower to the popular SB-Prolog. It also uses an emulator written in C but integrates partial evaluation techniques for specializing partly instantiated calls. It can also detect some cases of determinacy and efficiently compile control structures such as Prolog `if-then-else`. The compilation phase can therefore be quite long.

SWI-Prolog 1.8.11: the characteristics of this system are its compilation speed and the variety of built-in predicates that it provides. It is up to now one of the most frequently used academic Prolog systems.

Table 2 shows the execution times of those various systems and the average speedups of `wamcc`. An *overflow* entry in the table means that memory was exhausted before completion of the program with maximal stack sizes.

	`wamcc`	BinProlog	XSB-Prolog	SWI-Prolog
Program	2.21	3.0	1.4.0	1.8.11
`boyer`	3.450	6.700	11.450	21.200
`browse`	4.020	7.930	11.850	18.180
`cal`	0.300	0.920	1.420	5.120
`chat_parser`	0.980	1.200	1.790	2.050
`crypt`	0.016	0.017	0.040	0.100
`ham`	4.330	5.280	8.840	12.650
`meta_qsort`	0.045	0.100	0.140	0.130
`nand`	0.120	0.320	*overflow*	0.420
`nrev`	0.600	0.520	1.040	3.350
`poly_10`	0.300	0.420	0.720	1.200
`queens (16)`	2.440	4.670	6.480	31.220
`queens_n (8)`	0.700	0.920	1.560	3.450
`queens_n (10)`	13.680	16.030	28.541	56.180
`reducer`	0.270	0.550	*overflow*	0.930
`sdda`	0.015	0.030	0.050	0.030
`sendmore`	0.230	1.100	0.670	2.580
`tak`	0.550	1.400	1.430	651.000
`zebra`	0.260	0.400	0.530	0.580
average speedup of `wamcc`		2.0	2.7	5.3

Table 2: `wamcc` versus other academic Prologs

On average `wamcc` is twice as fast as BinProlog on average. However, on the `nrev` benchmark, BinProlog performs faster than `wamcc`. This is due to the very simple WAM used in `wamcc` which does not include any well-know optimization (like separate unification streams, shallow backtracking, unification reordering,...). We can also remarks that `wamcc` is 2.7 times faster than XSB-Prolog, and 5.6 times faster than SWI-Prolog (without taking into account the `tak` benchmark).

6.3 `wamcc` versus commercial Prolog systems

Let us now compare `wamcc` with commercial Prolog systems, usually developed by several people over several years. The Prolog systems to which `wamcc` is compared are:

Sicstus: this system is very popular because it has been one of the pioneering efficient systems available for a small fee. It has become a de facto reference for performances of Prolog systems. The current version can produced byte-code (emulated) as well as native code (for Sparc). We here compare wamcc to both versions.

Quintus: it has been for a long time the most efficient system. It is based on an optimized emulator written in assembly language. Note however that we only have version 2.5.1 which is not the most recent one.

Aquarius: this is currently the most efficient Prolog system. It is very complex and produces native code via an original abstract machine (the BAM) more low-level than the WAM. The compiler performs many optimizations (data-flow analysis, abstract interpretation, determinism detection,...). The translation of BAM code to native code is also quite sophisticated and integrates for instance instruction reordering for Sparc machines.

	wamcc	Sicstus	Sicstus	Quintus	Aquarius
	2.21	2.1	2.1	2.5.1	
Program		(emulated)	(native)		
boyer	3.450	4.940	2.350	2.850	2.750
browse	4.020	6.630	2.020	3.340	1.380
cal	0.300	0.890	0.540	0.500	0.290
chat_parser	0.980	1.130	0.500	0.650	0.350
crypt	0.016	0.027	0.013	0.017	0.010
ham	4.330	5.050	2.090	3.000	0.950
meta_qsort	0.045	0.048	0.021	0.050	0.015
nand	0.120	0.200	0.084	0.130	0.040
nrev	0.600	0.630	0.190	0.250	0.160
poly_10	0.300	0.320	0.150	0.250	0.070
queens (16)	2.440	4.930	1.280	2.820	0.610
queens_n (8)	0.700	0.980	0.370	0.580	0.130
queens_n (10)	13.680	18.200	7.250	10.780	2.250
reducer	0.270	0.270	0.120	0.270	0.100
sdda	0.015	0.023	0.016	0.017	0.010
sendmore	0.230	0.630	0.170	0.280	0.080
tak	0.550	1.020	0.390	1.620	0.060
zebra	0.260	0.300	0.230	0.230	0.160
average speedup of wamcc		1.6	↓ 1.7	1.0	↓ 3.7

Table 3: wamcc versus commercial Prologs

Table 3 presents execution times for those systems and the average speedup (or slowdown when preceded by a ↓ sign) of wamcc. Let us note that the initial objective is achieved, as wamcc is about 1.6 faster than emulated Sicstus. It is also 1.7 times slower than native code Sicstus and equivalent in performances to Quintus. wamcc is however more than 3.5 times slower than Aquarius on average, but this is mainly due to the very good performance of Aquarius on the tak benchmark (9 times faster than wamcc), because it can detect determinacy and optimize the terminal call for integers. Let us note that the Aquarius program is twice as fast as a corresponding C program ! However, on real Prolog programs (using unification and backtracking) the difference is smaller, as for instance on the zebra puzzle Aquarius is only 1.3 times faster and on the boyer program Aquarius is less than twice as fast. Let us remark that this system does not handle modules and thus can extract more information at compile-time allowing it to produce better code (e.g. symbols can be pre-hash-coded). This is no longer possible when dealing with several modules.

Observe that wamcc has been developed by a single person in only few months and it therefore quite honorably compares with sophisticated systems written over several years by teams of several implementors. Another important issue is the fact that wamcc is much simpler. Comparing the code complexity of the core of (native code) Sicstus (35000 lines of C) and of wamcc (6000 lines), we are far from the performance advantage of Sicstus (factor 1.6). Also The Sicstus compiler consists of 9000 lines of Prolog whereas the wamcc compiler is only 3000 lines long. About Quintus, the heavy use of assembly language gives a system which is more difficult to maintain and extend, while performances are not better than that of wamcc. Aquarius has remarkable performances, but mainly on deterministic and arithmetic programs, which are not the most typical Prolog applications one can hope to write. On the contrary, the fact that Aquarius requires about 38 minutes to compile a 400 lines long program (reducer) seems unrealistic. wamcc only takes 5 minutes to produce an executable program which is maybe 2.7 times slower but anyway takes less than one second. Another drawback of the Aquarius system is the size of produced code which is around 4 times bigger than this produced by wamcc.

7 Conclusion

We have presented the design and implementation of wamcc, a complete Prolog system based on the idea of compiling Prolog to C. We have shown that this alternative is viable and provides various advantages. The system obtained in this way is simple, extensible, portable (wamcc works on machines as different as 32-bit Sparcs and 64-bit Alphas) efficient and provides modularity at no extra-cost. This system is faster than all other academic Prologs and compares reasonably well with commercial systems, e.g. its performances are equivalent to those of Quintus Prolog. This is not a final step and these performances could be improved significantly simply by optimizing the WAM code produced. Indeed, wamcc only uses a conventional WAM without any well-known optimizations like: shallow backtracking, unification reordering, separate read/write stream, global analysis, optimal term representation. This should fill the gap between wamcc and the best implementations. wamcc has also proved to be a sound basis for various extensions, such as in particular the Constraint Logic Programming language clp(FD).

However, the most important characteristic of wamcc perhaps lies in its minimality and its "pyramidal" architecture based on the fact that there exist efficient low-level compilers able to take care of low-level tasks such as register allocation and various code optimizations. For instance it is now possible to access machine registers with the gcc compiler. Also the last version of gcc (2.6.0) provides new super-scalar instruction scheduler for Sparc and should bring some extra 10-15 % performaces on these machines for free. In the PC world, the Symantec C++ compiler is more performant than gcc both in compilation time and in quality of produced code and thus will be used when wamcc will be ported under MSDOS. The advantage of "reusing" instead of "rewriting" is therefore enormous. It saves time for research on more adventurous topics and it avoids producing ad hoc implementations which turn to be impossible to maintain, modify or extend.

How to get wamcc

The wamcc system is available by anonymous ftp at ftp.inria.fr in the directory /INRIA/Projects/ChLoE/LOGIC_PROGRAMMING/wamcc. A README file explains the installation procedure.

References

[1] H. Aït-Kaci. *Warren's Abstract Machine, A Tutorial Reconstruction.* Logic Programming Series, MIT Press, 1991.

[2] K. De Bosschere and P. Tarau. A Continuation Based Prolog-To-C Mapping. In *ACM Symposium on Applied Computing*, Phoenix, Arizona, 1994.

[3] T. Chikayama, T. Fujise and D. Sekita. A Portable and Efficient Implementation of KL1. In ICOT/NSF *Workshop on Parallel Logic Programming and its Programming Environments*, CIS-TR-94-04, Department of Computer Information Science, Oregon, 1994.

[4] B. Demoen and A. Mariën Can Prolog Execute as Fast as Aquarius. draft, BIM Kwikstraat, Everberg, Belgium, 1994.

[5] B. Demoen and G. Maris. A Comparison of some Schemes for Translating Logic to C. In *Workshop on Implementations* of *11th International Conference of Logic Programming*, Santa Margherita, Italy, MIT Press 1994.

[6] D. Diaz and P. Codognet. A Minimal Extension of the WAM for clp(FD). In *10th International Conference of Logic Programming*, Budapest, Hungary, MIT Press 1993.

[7] D. Gudeman, K. De Bosschere and S. Debray. jc: An Efficient and Portable Sequential Implementation of Janus. In *Joint International Conference and Symposium on Logic Programming*, Washington, MIT Press, 1992.

[8] B. Haussman. Turbo Erlang: Approaching the Speed of C. In *Implementations of Logic Programming Systems*, Evan Tick (ed.), Kluwer 1994.

[9] M. R. Levy and R. N. Horspool. Translator-Based Multiparadigm Programming. *Journal of Software and Systems*, 1993.

[10] M. R. Levy and R. N. Horspool. Translating Prolog to C: a WAM Based Approach. In *2nd Compulog Network Area Meeting Programming Languages and the workshop on Logic Languages*, Pisa, Italy, 1993.

[11] T. Solla. PhD dissertation (to appear), THOMSON-CSF, France, 1994.

[12] P. Tarau, B. Demoen and K. De Bosschere. The Power of Partial Translation: An Experiment with the C-ification of Binary Prolog. *ACM Symposium on Applied Computing*, Nashville, ACM Press, 1995.

[13] P. Van Roy and A. Despain. High-Performance Logic Programming with the Aquarius Prolog Compiler. IEEE Computer, pp 54-67, 1992.

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