GNU Prolog: beyond compiling Prolog to C

Daniel Diaz[1] and Philippe Codognet[2]

[1] University of Paris 1, CRI, bureau C1407
90 rue de Tolbiac, 75013 Paris, FRANCE
Daniel.Diaz@univ-paris1.fr
[2] University of Paris 6, LIP6, case 169
8 rue du Capitaine Scott, 75015 Paris, FRANCE
Philippe.Codognet@lip6.fr

Abstract

We describe in this paper the compilation scheme of the GNU Prolog system. This system is built on our previous experience of compiling Prolog to C in `wamcc`. The compilation scheme has been however redesigned to overcome drawbacks of the compilation to C. In particular, GNU-Prolog is based on a low-level mini-assembly platform-independent language that makes it possible to avoid the phase of compiling C code, and thus speeds up drastically compilation time. It also makes it possible to produce small stand alone executable files as the result of the compilation process. Interestingly, GNU Prolog is now compliant to the ISO standard and includes several extensions (constraint solving, OS interface, sockets, global variables, etc). The overall system is efficient and comparable in performances with commercial systems.

1 Introduction

GNU Prolog is a free Prolog compiler supported by the GNU organization (`http://www.gnu.org/software/prolog`). The development of GNU Prolog started in January 1996 under the name Calypso. Discussions with the GNU organization in late 1998 makes it released as a GNU product in April 1999: GNU Prolog.

GNU Prolog is based on our experience with `wamcc`. The main novelty of `wamcc` [4] was to translate Prolog to C but with also the idea of translating a WAM branching into a native code jump in order to reduce the overhead of calling a C function, see [4] for details. `wamcc` shown that this approach was an attractive alternative to the classical solution consisting in a WAM emulator. Indeed, the performances of this unoptimized system were close to commercial systems based on a highly optimized emulator. Moreover, many ideas of `wamcc` have inspired the designers of some Prolog systems since its diffusion in 1992. From the user point of view the main advantage of `wamcc` was its ability to produce stand alone executables while most other Prolog systems need the presence of the emulator at run-time. There is however a serious drawback when compiling Prolog to C, which is the size of the C file generated and the time taken to

compile such a big program by standard C compilers (e.g. `gcc`). Indeed, a Prolog program compiles to many WAM instructions (e.g. the 3000 lines of the GNU Prolog compiler give rise to 12000 WAM instructions) and trying to inline each WAM instruction could lead to a very big C file that cannot be handled by the C compiler. In order to cope with big Prolog sources we decided, in `wamcc`, to translate most WAM instructions to a call to a C function performing the treatment. Obviously the execution is a bit slowed down but the compilation is much faster (and the executable is smaller). However, even with this solution, the C compiler took too much time for big sources, especially in trying to optimize the code produced.

The novelty of the GNU Prolog compilation scheme is to translate a WAM file into a mini-assembly (MA) file. This language has been specifically designed for GNU Prolog. The idea of the MA is then to have a machine-independent intermediate language in which the WAM is translated. The corresponding MA code is mapped to the assembly language of the target machine. In order to simplify the writing (i.e. porting) of such mappers the instruction set of MA must be simple, in the spirit of the LLM3 abstract machine for Lisp [3], as opposed to the complex instruction set of the BAM designed for Aquarius Prolog [8]. Actually, the MA language is based on 11 instructions, mostly to handle the control of Prolog and to call a C function (performing most of the treatment). This new compilation process is between 5-10 times faster than `wamcc`+`gcc`. The rest of this paper is devoted to a detailed explanation of this compilation scheme.

Moreover, `wamcc` had been designed as an experimental platform to provide a sound basis for various extensions (like Constraint Logic Programming). Due to this choice, several functionalities were missing in `wamcc` (e.g. no floating arithmetic, no stream support,...) and the whole system was not robust enough. The main goal of GNU Prolog was then to provide a free, open, robust, extensible and complete Prolog system. Like `wamcc` it should be able to produce efficient stand alone executables but overcoming all limitations cited above.

GNU Prolog is a complete system including: floating point numbers, streams, dynamic code, DCG, operating system interface, sockets, a Prolog debugger, a low-level WAM debugger, a line editing facility with completion on atoms, etc. GNU Prolog offers more than 300 Prolog built-in predicates and is compliant to the ISO standard for Prolog [7] (GNU Prolog is today the only free Prolog system really compliant to this standard). There is also a powerful bidirectional interface between Prolog and C, featuring implicit Prolog $\leftrightarrow$ C type conversion, transparent I/O argument handling, non-deterministic C code, ISO error support, etc. This is a key point to allow users to write their own extensions. Finally, GNU Prolog includes a very efficient constraint solver over finite domains inspired from `clp(FD)` [5,6] containing many predefined constraints: arithmetic constraints, boolean constraints, symbolic constraints, reified constraints; there are more than 50 FD built-in constraints/predicates, and several predefined labeling heuristics. Moreover the solver is extensible, and new high-level constraints can be easily defined by the user and defined in terms of simple primitives.

The rest of this paper is organized as follows. Section 2 introduces the MA language while section 3 describes how this language can be mapped to a specific architecture. Section 4 is devoted to the link phase. Performance evaluation is detailed in Section 5, and a short conclusion ends the paper.

2 The Mini-assembly Language

2.1 Overview

We here describe the mini-assembly (MA) language. The idea of the MA language is to have a machine-independent intermediate language in which the WAM is translated. The design of MA comes from the study of the C code produced by `wamcc`. Indeed, in the `wamcc` system, most WAM instructions given rise to a call to a C function performing the treatment (e.g. unification, argument loading, environment and choice-point management). The only exceptions were obviously instructions to manage the control of Prolog and some short instructions that were inlined. The MA language has been designed to avoid the use of the C stage and thus has instructions to handle the Prolog control, to call a C function and to test/use its returned value. The MA file is then mapped to the assembly of the target machine (see section 3) from which an object is produced. Thus, the `wamcc` sequence: *WAM* $\rightarrow$ *C* ($\rightarrow$ *assembly*) $\rightarrow$ *object* becomes in GNU Prolog: *WAM* $\rightarrow$ *MA* $\rightarrow$ *assembly* $\rightarrow$ *object*.

In order to simplify the writing of translators of the MA to a given architecture (i.e. the mappers), the MA instruction set must be simple: it only contains 11 instructions.

2.2 The MA Instruction Set

Here is a description of each MA instruction:

`pl_jump` *`pl_label`*`:` branch the execution to the predicate whose corresponding symbol is *`pl_label`*. This symbol is an identifier whose construction is explained later in section 2.3. This instruction corresponds to the WAM instruction `execute`.

`pl_call` *`pl_label`*`:` branch the execution to the predicate whose corresponding symbol is *`pl_label`* after initializing the continuation register `CP` to the address of the very next instruction. This instruction corresponds to the WAM instruction `call`.

`pl_ret:` branch the execution to the address given by the continuation pointer `CP`. This instruction corresponds to the WAM instruction `proceed`.

`pl_fail:` branch the execution to the address given by the last alternative (`ALT` cell of the last choice point pointed by the WAM `B` register). This instruction corresponds to the WAM instruction `fail`.

`jump` *`label`*`:` branch the execution to the symbol *`label`*. This instruction is used when translating indexing WAM instructions to perform local control transfer (e.g. `try`, `retry` or `trust`). This instruction has been distinguished from

`pl_jump` (even if both can be implemented/translated in a same manner) since, on some machines, local jumps can be optimized.

`call_c` *`fct_name`*`(`*`arg`*`,...):` call the C function *`fct_name`* passing the arguments *`arg`*,... Each argument can be an integer, a float (C `double`), a string, the address of a label, the address or the content of a memory location, the address or the content of a WAM `X` or `Y` register. This instruction is used to translate most of the WAM instructions.

`fail_ret:` perform a Prolog fail (like `pl_fail`) if the value returned by the previous C function call is 0. This instruction is used after a C function call returning a boolean to indicate its result (e.g. functions performing the unification).

`jump_ret:` branch the execution to the address returned by the previous C function call. This instruction makes it possible to use C functions to determine where to transfer the control. For instance, the WAM indexing instruction `switch_on_term` is implemented via a C function accepting several addresses and returning the address of the selected code.

`move_ret` *`target`*`:` copy the value returned by the previous C function call to *`target`* which can be either a memory location or a WAM `X` or `Y` register.

`c_ret:` C return. This instruction is used at then end of the initialization function (see below) to give back the control to the caller.

`move` *`reg1`*`,`*`reg2`*`:` copy the content of the WAM `X` or `Y` register *`reg1`* to the register *`reg2`*.

It is worth noticing the minimality of the language which is based on a very restricted instruction set. Note however the presence of the `move` instruction to perform a copy of WAM X or Y registers. We could instead invoke a C function to perform such a move (using `call_c`). However, those moves between registers are rather frequent and the invocation of a C function would be costly. There is thus a compromise to find between the minimality of the instruction set and the performance. Obviously, it is possible to extend this instruction set (e.g. adding arithmetic instructions) but this will complicate much more the writing of the mappers to assembly. Performance evaluation will show that this instruction set gives good results.

Beside these instructions, the MA language include several declarations which are presented now. The keyword `local` specifies a local symbol (only visible in the current object) while `global` allows other object to see that symbol.

`pl_code local/global` *`pl_label`*`:` define a Prolog predicate whose corresponding symbol is *`pl_label`*. For the moment all predicates are `global` (i.e. visible by all other Prolog objects). But `local` will be used when implementing a module system.

`c_code local/global/initializer` *`label`*`:` define a function that can be externally called by a C function. The use of `initializer` ensures that this function will be executed first, when the Prolog engine is started. Only one function per file can be declared as `initializer`.

`long local/global` *`ident`* `=` *`value`*`:` allocate the space for a `long` variable whose name is *`ident`* and initializes it with the integer *`value`*. The initialization is optional (i.e. the `=` *`value`* part can be omitted).

`long local/global` *`ident`*`(`*`Size`*`):` allocate the space for an array of *`Size`* `long`s whose name is *`ident`*.

The *WAM* → *MA* translation can be performed in linear time w.r.t. the size of the WAM file (the translation is performed on the fly while the WAM file is read).

2.3 Associating an Identifier to a Predicate Name

Since the MA language is later mapped to the assembly of the target machine only classical identifiers can be used (a letter followed by letters, digits or the underscore character). In particular, it is necessary to associate such an identifier (referenced as *`pl_label`* in section 2.2) to each predicate. Since the syntax of identifiers is more restrictive than the syntax of Prolog atoms (which can include any character using quotes) GNU Prolog uses an hexadecimal representation where each predicate name is translated to a symbol beginning with an `X` followed by the hexadecimal notation of the code of each character of the name followed by an underscore and the arity. For instance `append/3` is coded by the symbol `X617070656E64_3` (`61` is the hexadecimal representation of the code of `a`, `70` is associated to `p`, ...). The linker is then responsible for resolving external references (e.g. call to built-in predicates or to user predicates defined in an other object). The output of the linker is filtered by GNU Prolog to decode eventual hexadecimal notations in case of errors (e.g. undefined predicate, multiple definitions for a predicate).

2.4 An Example

We here present the MA code associated to the simple clause `p(T,g(U),V):-q(a,T,V,U)`. Associated WAM instructions are shown as comment.

```
% gplc -M t.pl
% more t.ma
pl_code global X70_3                    ; define predicate p/3
   call_c   Get_Structure(at(2),1,X(1)) ; get_structure(g/1,1)
   fail_ret
   call_c   Unify_Variable()            ; unify_variable(x(3))
   move_ret X(3)
   move     X(0),X(1)                   ; put_value(x(0),1)
   call_c   Put_Atom(at(3))             ; put_atom(a,0)
   move_ret X(0)
   pl_jump  X71_4                       ; execute(q/4)

long local at(4)                        ; table for 4 atoms
```

```
c_code initializer Object_Initializer   ; object initializer
   call_c   Create_Atom("t.pl")         ; atom #0 is 't.pl'
   move_ret at(0)
   call_c   Create_Atom("a")            ; atom #3 is 'a'
   move_ret at(3)
   call_c   Create_Atom("g")            ; atom #2 is 'g'
   move_ret at(2)
   call_c   Create_Atom("p")            ; atom #1 is 'p'
   move_ret at(1)
   call_c   Create_Pred(at(1),3,at(0),1,1,&X70_3)
   c_ret                                ; define predicate p/3
```

It is easy to see that most WAM instructions give rise to a C function call (e.g. `call_c Get_Structure()`). Calls to functions that can fail (unification) are followed by a `fail_ret` that performs a Prolog fail if the returned value is 0. Note the presence of the MA instruction `move` to perform a copy of WAM registers (associated to the WAM instruction `put_value(x(0),1)`).

According to the encoding presented in section 2.3, the symbol `X70_3` is associated to `p/3` (and `X71_4` to `q/4`) .

It is worth noting how atoms are managed. All atoms are classically stored in a hash-table. To cope with separate linking the hash-table must be built at run-time (while it is possible to compute hash-values at compile-time in the presence of a single file). For that the function `Object_Initializer` is first invoked. It is responsible for updating the atom table with atoms needed by the object. The hash value of each atom is then stored in a local array (`at(`*`atom_number`*`)`) and is used by instructions handling atoms (e.g. `put_atom`) or functors (e.g. `get_structure`). The initialization function also updates the predicate table with predicates defined in the object. Both properties (public/private, static/dynamic, user/built-in) and the address of the code of each predicate are added. The knowledge of the address of the code is only necessary for meta-call (e.g. to implement `call/1`) since all other references are resolved by the linker. The way the initializer function is automatically invoked at run-time is explained later in section 4.

3 Mapping the Mini-assembly to a Target Machine

The next stage of the compilation consists in mapping the MA file to the assembly of the target machine. Since MA is based on a reduced instruction set, the writing of such translators is simplified. However, producing machine instructions is not an easy task. The first translator was written with the help of a C file produced by `wamcc`. Indeed, compiling this file to assembly with `gcc` gave us a first solution for the translation (since the MA instructions corresponds to a subset of that C code). We have then generalized this by defining a C file (now independently from `wamcc`). Each portion of this C code corresponds to a MA instruction and the study of the assembly code produced by `gcc` is a good starting point. This gives a first information about register conventions, C calling

conventions,... However, to further optimize the assembly code it is necessary to refer to the technical documentation of the processor together with the ABI (Application Binary Interface) used by the operating system. Our experience is that such a backend for a new architecture can be produced within a week.

Here is the interesting portion of the linux/ix86 assembly code corresponding to the definition of `p/3` (the associated MA code is shown as comment):

```
% gplc -S t.pl
% more t.s
fail:
   movl   1028(%ebx),%eax # fail
   jmp    *-4(%eax)

   .globl X70_3           # pl_code  global X70_3
X70_3:
   movl   at+8,%eax       # call_c   Get_Structure(at(2),1,X(1))
   movl   %eax,0(%esp)    #    arg   at(2)
   movl   $1,4(%esp)      #    arg   1 ($1=immediate value)
   movl   4(%ebx),%eax
   movl   %eax,8(%esp)    #    arg   X(1)
   call   Get_Structure
   testl  %eax,%eax       # fail_ret
   je     fail
   call   Unify_Variable  # call_c   Unify_Variable()
   movl   %eax,12(%ebx)   # move_ret X(3)
   movl   0(%ebx),%eax    # move     X(0),X(1)
   movl   %eax,4(%ebx)
   movl   at+12,%eax      # call_c   Put_Atom(at(3))
   movl   %eax,0(%esp)
   call   Put_Atom
   movl   %eax,0(%ebx)    # move_ret X(0)
   jmp    X71_4           # pl_jump  X71_4

.data
   .local at              # long local at(4)
   .comm  at,16,4
```

Here again, a crucial point is that the mapping *MA* → *assembly* is executed in linear time w.r.t. the size of the MA file (the translation is done on the fly while the MA file is read). Obviously the translation to the assembly of the target machine makes room for several optimizations. For instance the ix86 mapper uses the `ebx` register as a global register to store the address of the bank of WAM registers (consisting in 256 `X` registers followed by control registers: `H`, `B`, ...). Maintaining this address in `ebx` makes it possible to load/store (into/from a processor register) any WAM register with only one machine instruction. More generally, it is possible to use machine registers if it is ensured that they are saved and restored by functions using them (the ABI gives this information).

Note the definition of a `fail` label which performs a WAM `fail`. The associated code first loads the value of the `B` register (pointer to the last choice-point) and then branches to the value of the `ALT` cell of that choice-point (stored at the offset -1(*4 bytes) from `B`).

Another optimization used in this translation consists in using an original argument passing scheme. Under ix86, arguments are passed into the stack (as usually for CISC processors). The classical way to call a C function is then to use `push` instructions to initialize arguments, to call the function and, after the return of the function, either to use several `pop` instructions or to perform a stack pointer adjustment (adding a positive number to it). Many optimizing C compilers try to group these stack adjustments delaying them as long as possible to only perform one addition. GNU Prolog does better by avoiding all adjustements. This is done by reserving at the start of the Prolog engine enough space in the stack [1] and then copying arguments on that space (cf. `movl ...,`*`offset`*`(%esp)` instructions with *`offset`* a positive integer). This could not be done when compiling to C in `wamcc` (like many other optimizations included in the GNU Prolog mappers).

4 Linking

All objects are linked together at link-time with the GNU Prolog libraries: Prolog built-in predicate library, FD built-in constraint/predicate library and run-time library. This last library contains in particular functions implementing WAM instructions (e.g. `Get_Structure()`,...). Linked objects come from: Prolog sources, user C foreign code or FD constraint definition. This stage resolves external symbols (e.g. a call to a predicate defined in another module).

Since a Prolog source gives rise to a classical object, several objects can be grouped in a library (e.g. using `ar` under Unix). The Prolog and FD built-in libraries are created using this way (the user can also define his own libraries). Defining a library allows the linker to extract from it only the needed objects, i.e. those containing statically referenced functions/data. For this reason, GNU Prolog offers an option to generate small executables by avoiding the inclusion of most unused built-in predicates. To cope with meta-calls in that case, GNU Prolog provides a directive to force the linker to include a given predicate. To further reduce this size of the executables the linker should exclude *all* (instead of *most*) unused predicates. To do this we should define a built-in predicate per Prolog file (similarly to what is done for the C standard library) since the object is the unit of inclusion of the linker (i.e. when a symbol is referenced the whole object where this symbol is found is linked). For the moment built-in predicates are grouped by theme, for instance, a program using `write/1` will give rise to an executable also containing the code of `writeq/1`, `display/1`,... In the future we will define only one predicate per file. In the same spirit we will also define only one C function associated to a WAM instruction per file (e.g. to avoid to link the code of `Put_Structure()` if this instruction is not used).

[1] enough to store the maximal number of arguments of library functions.

In section 2.4 we have mentioned the role of the initializer function (called `Object_Initializer` in our example). It is worth explaining how this function is invoked. Indeed, the Prolog engine must be able to find dynamically at run-time all objects selected by the linker and execute their initializer function. The solution retained in GNU Prolog consists in marking all objects with magic numbers together with the address of the initializer function. At run-time, a pertinent portion of the data segment is scanned to detect each linked object (thanks to the magic numbers) and invoke its initializer.

5 Prolog Performance Evaluation

5.1 Compilation

Program	Lines	object		executable	
		time	size	time	size
`boyer`	362	0.520	44	0.650	154
`browse`	88	0.200	11	0.420	129
`cal`	131	0.190	11	0.320	118
`chat_parser`	905	1.430	113	1.620	221
`ham`	48	0.120	8	0.240	110
`nand`	518	0.850	61	1.020	310
`nrev`	52	0.100	4	0.280	112
`poly_10`	86	0.160	10	0.360	120
`queens (16)`	60	0.080	3	0.170	111
`queens_n (10)`	37	0.070	4	0.270	112
`reducer`	307	0.420	30	0.570	148
`sendmore`	52	0.110	7	0.360	114
`tak`	24	0.060	2	0.200	110
`zebra`	42	0.090	5	0.190	107
`pl2wam`	3000	3.430	286	3.690	557

Table 1. GNU Prolog compilation evaluation

Table 1 presents the performances of the GNU Prolog compilation scheme on a classical set of benchmarks, times are in seconds and sizes in KBytes. We have also added the GNU Prolog `pl2wam` sub-compiler since it is a more representative example. For each program, one can find: the number of lines of the Prolog source program [2], the compilation time needed to produce the object, the size of the object code (stripped), the total compilation time (including the link) and the final executable size (stripped). Timings are measured on a Pentium II 400 Mhz with 256 MBytes of memory running Linux RedHat 5.2.

[2] neither blank lines nor comments are counted.

The size of (stripped) objects show that this approach really generates small code (less than 10 KBytes for many benchmarks). The size of the whole executable shows the interest of excluding most of unused built-in predicates. Indeed, when all built-in predicates (Prolog+FD) are present the size is at least 596 KBytes (this is the size of the GNU Prolog interactive top-level). Let us recall that we can even further reduce this size with a little reorganization of GNU Prolog libraries (see section 4). The ability of GNU Prolog to produce stand alone small executables is an important feature that makes it possible to use them in many occasions (tools, web CGIs,...). Other Prolog systems cannot produce such standalone executables since they always need the presence of the emulator at run-time (500 KBytes to 2 MBytes).

Compilation timings are rather good and we have reached our initial goal since GNU Prolog compiles 5-10 times faster than `wamcc+gcc`. Obviously this factor is not constant and the gain is more effective on large programs (and thus it is difficult to give an upper bound of the speedup factor). This is due to the fact that the translation from the WAM to an object is done in linear time (each translation only needs one pass) while a C compiler can need a quadratic time (and even worse) for its optimizations. Table 2 illustrates this comparing compilation times for both systems on some representative benchmarks.

Program	Lines	GNU Prolog	`wamcc`	Speedup
`cal`	131	0.320	1.210	3.7
`boyer`	362	0.650	3.050	4.6
`chat_parser`	905	1.620	10.120	6.3
`pl2wam`	3000	3.690	34.210	9.2

Table 2. Compilation speed - GNU Prolog versus `wamcc`

5.2 Benchmarking Prolog

In this section we compare GNU Prolog with one commercial system: SICStus Prolog emulated and five academic systems: Yap Prolog, `wamcc`, BinProlog (the last version is now commercialized), XSB-Prolog and SWI-Prolog. Table 3 presents execution times for those systems and the average speedup (or slowdown when preceded by a $\downarrow$ sign) of GNU Prolog (the `nand` program could not be run with XSB-Prolog). For each benchmark, the execution time is the average of 10 consecutive executions.

To summarize, GNU Prolog is 1.6 times slower than Yap Prolog, the fastest emulated Prolog system and also slightly slower than `wamcc`, mainly because of a richer arithmetic support. On the other hand, GNU Prolog is 1.2 times faster than SICStus emulated, 2.3 times faster than BinProlog, around 2.5 times faster than XSB-Prolog and more than 4 times faster than SWI-Prolog (without taking

Program	GNU Prolog 1.0.5	Yap Prolog 4.2.0	wamcc 2.21	SICStus Prolog 3.7.1	Bin Prolog 5.75	XSB Prolog 1.8.1	SWI Prolog 3.2.8
boyer	0.322	0.187	0.270	0.315	0.576	0.830	1.322
browse	0.410	0.189	0.316	0.409	0.798	0.804	1.217
cal	0.032	0.032	0.030	0.030	0.033	0.031	0.031
chat_parser	0.075	0.055	0.075	0.089	0.103	0.245	0.240
ham	0.293	0.170	0.336	0.329	0.393	0.582	0.723
nand	0.011	0.008	0.010	0.016	0.022	?.???	0.067
nrev	0.042	0.019	0.039	0.037	0.023	0.089	0.192
poly_10	0.023	0.013	0.020	0.024	0.031	0.056	0.097
queens (16)	0.223	0.136	0.126	0.404	0.378	0.751	2.261
queens_n (10)	1.091	0.542	1.011	1.193	1.307	2.143	4.067
reducer	0.021	0.010	0.021	0.022	0.243	0.064	0.077
sendmore	0.026	0.020	0.015	0.047	0.063	0.087	0.166
tak	0.040	0.031	0.029	0.066	0.110	0.156	28.810
zebra	0.026	0.017	0.027	0.021	0.034	0.040	0.057
GNU Prolog speedup		↓ 1.6	↓ 1.2	1.2	2.3	2.5	4.2

Table 3. GNU Prolog versus other Prolog systems

into account the `tak` benchmark). To be fair let us mention that had not enough time to exaustively compare with all the Prolog systems and their variants. For instance, SICStus Prolog can compile to native code for some architectures (e.g. under SunOS/sparc but not yet under linux/ix86) and then it will be 2.5 times faster than GNU Prolog on those platforms, BinProlog can partly compile to C and CIAO Prolog seems a bit faster than GNU Prolog,...

Alltogether, this performance evaluation shows that a Prolog system based on a simple, unoptimized WAM engine can nevertheless have good efficiency with this MA-based native compilation scheme. Obviously further improvements could be achieved by integrated all well-known WAM optimizations.

6 Conclusion

GNU Prolog is a free Prolog compiler with constraint solving over finite domains. The Prolog part of GNU Prolog conforms to the ISO standard for Prolog with also many extensions very useful in practice (global variables, OS interface, sockets,...). The finite domain constraint part of GNU Prolog contains all classical arithmetic and symbolic constraints, and integrates also an efficient treatment of reified constraint and boolean constraints. The new compilation scheme of GNU Prolog drastically speeds up compilation times in comparison to compiling to C (5-10 times faster than `wamcc`+`gcc`). The MA language can be used by other logic languages as a target language. This choice has been made for the Dyalog system (a logic programming language with tabulation). GNU Prolog produces native binaries and the executable files produced are stand alone. The size of

those executable files can be quite small since GNU Prolog can avoid to link the code of most unused built-in predicates. The performances of GNU Prolog are close to commercial systems and several times faster than other popular free systems.

References

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

2. M. Carlsson. *Design and Implementation of an Or-Parallel Prolog Engine*. PhD dissertation, SICS, Sweden, 1990.

3. J. Chailloux. La machine LLM3. Technical Report RT-055, INRIA, 1985.

4. P. Codognet and D. Diaz. `wamcc`: Compiling Prolog to C. In *12th International Conference on Logic Programming*, Tokyo, Japan, MIT Press, 1995.

5. P. Codognet and D. Diaz. A Minimal Extension of the WAM for `clp(FD)`. In *Proc. ICLP'93, 10th International Conference on Logic Programming*. Budapest, Hungary, MIT Press, 1993.

6. P. Codognet and D. Diaz. Compiling Constraint in `clp(FD)`. *Journal of Logic Programming*, Vol. 27, No. 3, June 1996.

7. Information technology - Programming languages - Prolog - Part 1: General Core. ISO/IEC 13211-1, 1995.

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

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