Multiple Specialization through Local Analysis

Michel Ferreira   Luís Damas

Technical Report Series: DCC-98-2

Departamento de Ciência de Computadores – Faculdade de Ciências

&

Laboratório de Inteligência Artificial e Ciência de Computadores

Universidade do Porto

Rua do Campo Alegre, 823 4150 Porto, Portugal

Tel: +351+2+6078830 – Fax: +351+2+6003654

http://www.ncc.up.pt/fcup/DCC/Pubs/treports.html

Multiple Specialization through Local Analysis

Michel Ferreira Luís Damas
DCC-FC & LIACC, University of Porto,
R. do Campo Alegre, 823 4150 Porto, Portugal
{michel,luis}@ncc.up.pt

Abstract

Program specialization is normally supported by global analysis of the program. Compilers use the information deduced to generate more efficient, specialized implementations of the program. This specialization can be *single* or *multiple*, depending if each procedure of the program is specialized into one or more versions. We present a Prolog compiler that does *multiple specialization*, using an algorithm that works over the WAM code, deducing the reachable procedure activations based on local analysis. The idea is to use the analysis that is done in the Prolog to WAM compilation, and that generates local specialized instructions, and to propagate this information through the entire program. The creation of multiple versions of predicates allows the propagation to be inter-procedural, and to achieve global program specialization. Results from an implementation generating C code as target language are presented, showing a performance comparable to native code Prolog compilers.

Keywords: Prolog Compilation, Code Specialization, Program Analysis.

1 Introduction

The efficiency of a Prolog compiler is normally associated with two aspects: a program analyzer and the generation of native code. Parma [Tay91] and Aquarius [Van90] are two of the most efficient Prolog compilers and both generate native code and have a program analysis phase.

The analysis phase is normally done by applying a technique named *abstract interpretation* [CC92] to the source program. This analysis generates information about modes and types of arguments [DW88, BJ88], variable aliasing [JL89], determinism [Mel86], etc. This information is then used to generate specialized code, normally in the form of some low level abstract machine, like the BAM (Berkeley Abstract Machine) [Van90] in Aquarius. The WAM (Warren Abstract Machine) [War83] is much too high level to use the information of the analysis.

The code specialization results from knowing that the activation patterns from procedure calls have certain properties, like, for instance, having dereferenced arguments, non-instantiated arguments or of integer type. A new specialized version of the procedure is then generated, making use of the properties deduced. If all the activation patterns of a certain procedure that occur in a program are similar, then the specialized code of that procedure can be importantly optimized. If however there are very distinct (or unknown) activation patterns to a procedure, then the optimizations are limited. An example of unknown activation patterns occurs on modular programming, when not all modules are available for analysis. This kind of code specialization is known as *single specialization*, because it is generated just one specialized version for each procedure. Parma and Aquarius are, basically, single specializations compilers.

Winsborough proposed in [Win89] another kind of specialization, where a different specialized version was generated to each deduced reachable activation pattern. This is known as *multiple specialization*. Some papers were written based on this idea [JLHW90, PGB90], but we do not know any compiler which implemented the algorithms described. The way to deduce the kind of activation patterns that were reachable during execution was also by using abstract interpretation over the source program.

In this paper we present an algorithm to obtain multiple specialization that is not based on abstract interpretation, showing how a local analysis on the WAM instructions of a predicate can be used to deduce the activation patterns of the calls and how the creation of multiple versions of predicates allows the propagation of such information to be inter-procedural, achieving global program specialization.

The remainder of this paper is organized as follows: in the following section we discuss some techniques of program analysis, as the tool to derive the information needed to program specialization; next we present our strategy to obtain specialized code, showing how we can simplify this analysis; we then discuss the implementation of the WAM instructions generated for the multiple specialized predicates; next we present evaluation results of the performance of our system and we end with some conclusions.

2 Abstract Interpretation and Abstract Compilation

To obtain code specialization, single or multiple, the framework of abstract interpretation has been widely used, as the tool to global data-flow analysis of a programming language based on an unified model [CC77], in order to extract information from programs, like mode information, type or variable aliasing.

The idea is to execute a program over a finite *abstract domain* rather than the possibly infinite *concrete domain*. Elements and operators from the *abstract domain* map a class of elements and operators of the *concrete domain*. The abstract interpretation proceeds until a fix-point of the program properties is reached. New definitions of unification and variable substitutions are needed in this abstract execution.

The major problem with abstract interpretation is that it is a very time consuming process. Also, the complexity of real programs makes obtaining precise data rather difficult, deriving naive results that do not pay back the time spent in the analysis phase.

Another method of analysis has been proposed that tries to overcome the large time-consumption of abstract interpretation, and is known as *abstract compilation* [BF96]. It is based on the simple idea that instead of interpreting the source program it should be compiled into a program which computes the desired analysis when it is executed. The removal of a layer of abstraction to the analysis process should result in an important improvement on execution speed.

In [BF96] the authors show how the control flow analysis of higher-order functional programs can be compiled.

The name of *abstract compilation* in the context of logic programs analysis has been first used by Tan and Lin in [TL92]. However, their approach is basically identical to the abstract interpretation approaches. There is no notion of analysis program as in [BF96], as what they do is an *abstract interpretation of the WAM compiled code*.

This approach is however very interesting and has a similar motivation of the idea we describe in this paper: in the WAM code there are important specializations, for instance in specialized unification subroutines, defined for each type of term (`_variable`, `_value`, `_const`, `_list`, `_struct`), that in [TL92] are used to simplify the analysis and that we use with a propagation technique to generate multi-specialized code.

3 Unfolding WAM Code

The multi-directionality of Prolog is well appreciated by the programmer, but is a problem to the compiler designer. When no modes declarations are given or deduced, the code that is generated to a predicate will answer calls from different "directions", handicapping the efficiency of the code with several run-time tests, to implement distinct cases in one general procedure.

Recursive predicates are extremely used in Prolog, and recursive calls show how those run-time tests are redundantly executed. If a specialized predicate was created for each basic direction of execution then those run-time tests could be importantly minimized.

In Figure 1 we present some specialized versions of the `append/3` predicate. Those specialized versions are the versions that our algorithm of multiple specialization generates to the `append/3` predicate. In the next sections we will show how they are generated.

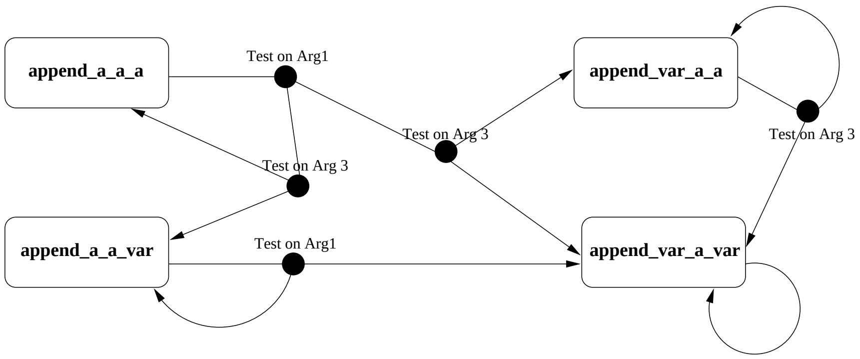

Figure 1: Call graph on unfolded append program

The `append_a_a_a`[1] predicate is the general `append/3` predicate. In a single version and non-specialized implementation this will be the only predicate called. Two important run-time tests are executed, the first one on argument 1, corresponding to the `get_list(1)` instruction and the second one on argument 3, corresponding to the `get_list(3)` instruction. The WAM code for the recursive clause of the `append/3` predicate is presented in Figure 2.

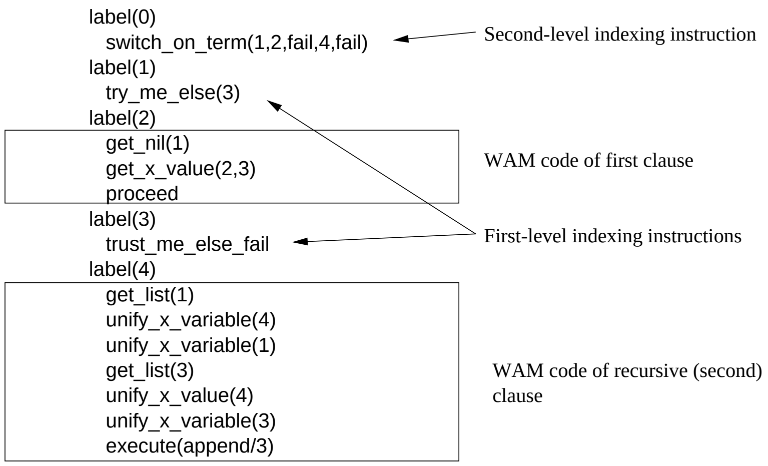

Figure 2: WAM code of `append/3`

If the result from these run-time tests was propagated to the calling instructions, leading the calls to specialized versions as is represented in the graph on Figure 1, then in subsequent calls those run-time tests could be partially or totally eliminated.

3.1 Propagation of information to inter-procedural specialization

The `get_list` instruction is a specialization of the `get_structure` instruction, where the functor is `./2`. This allows the implementation of such unification to be simplified. The kind of specialization present in WAM instructions is local to instructions, or in the best is intra-procedural, but an inter-procedural specialization is non-existent. To obtain inter-procedural specialization analysis is needed. Our idea is to propagate the local specialization information of WAM instructions through the

[1] `a` stands for type *any*, describing the type of the first, second and third arguments, respectively.

entire code of the program, avoiding a global analysis phase. In calling instructions the information propagated until the call point will describe the activation pattern of that call, and will lead to the creation of the correspondent specialized version, allowing the propagation of information to be inter-procedural. A similar work that also tries to relate procedure calls directly with heads unification is present in the VAM [KN90], where two program pointers are used, one to the calling goal and one to the head of the unify-able clause.

When an important run-time test is done, like the `get_list` in `append/3`, and information about the argument tested is not present, the code will be unfolded to propagate the succeeding possible types through the distinct branches. This happens in figure 1 where tests on arguments which have type *any* cause code to be split in two, with different `execute` instructions that call different predicates.

The process of collecting information from WAM instructions and propagating it is very simple and we will describe it using the `append/3` example.

The information we need to propagate and maintain is the following:

- type of temporary and global variables
- mode of structure register
- possible sharing of local and global variables

Figure 2 presents the WAM code of the `append/3` predicate, with labels and indexing instructions. To a detailed explanation of WAM instructions see [AK91].

In this example the first run-time test is done by the second-level indexing instruction `switch_on_term(1,2,fail,4,fa` It tests the tag of argument 1 and jumps to label 1 if it is *var*, to label 2 if it is *const* or to label 4 if it is *list*. If it is *integer* or *struct* execution fails.

As nothing is known about the type of argument 1 (we are applying our algorithm to the general `append/3` WAM code), we will propagate the succeeding possible types (*var*,*const* or *list*) through the correspondent distinct branches. The propagation of different information to each distinct branch leads to different implementations of the WAM instructions, and no longer makes possible to interlace the code of the various cases, as is done in Figure 2. The code of clauses now as to be repeated to the *var*, *const* and *list* cases.

To distinguish the implementation of WAM instructions in each of these cases, based on the information propagated, we complement the instructions with the relevant information to their simplification. A `get_list(1)` instruction where it has been propagated that register argument 1 has type *var* becomes `get_list_var(1)`. This specialized instruction propagates the unification mode to be *write*, which will complement the subsequent `unify` instructions.

Some important run-time tests, like the `get_list(3)` in our example, where no information is present about the argument, will be replaced by an instruction which splits the code in distinct branches and jumps to the correspondent branch accordingly to the test made. A `switch_on_list(3,3,4)` will replace a `get_list(3)` instruction where argument 3 has type *any*, testing the type of this argument and jumping to label 3 if it has type *list* or to label 4 if it has type *var*.

The calling instructions (`execute` in our example) are complemented with the propagated type of their arguments, calling a correspondent specialized predicate that will be created.

In Figure 3 we present the WAM unfolded code of `append_a_a_a/3` after applying our algorithm of specialization.

The difference between the code of Figure 2 and the code of Figure 3 is that in this last code we have propagated the information of the run-time tests that are made, causing blocks of code to be repeated because of implementation differences, and obtaining 4 distinct predicate calls.

3.2 Creation of specialized predicates

The inter-procedural specialization is obtained through the propagation of the activation pattern of the calling goal into the predicate which will handle the call. In our algorithm this is automatically achieved, because each activation pattern of the calling goal that is deduced through the process of propagation, will lead to the creation of a predicate that is specialized in that activation pattern.

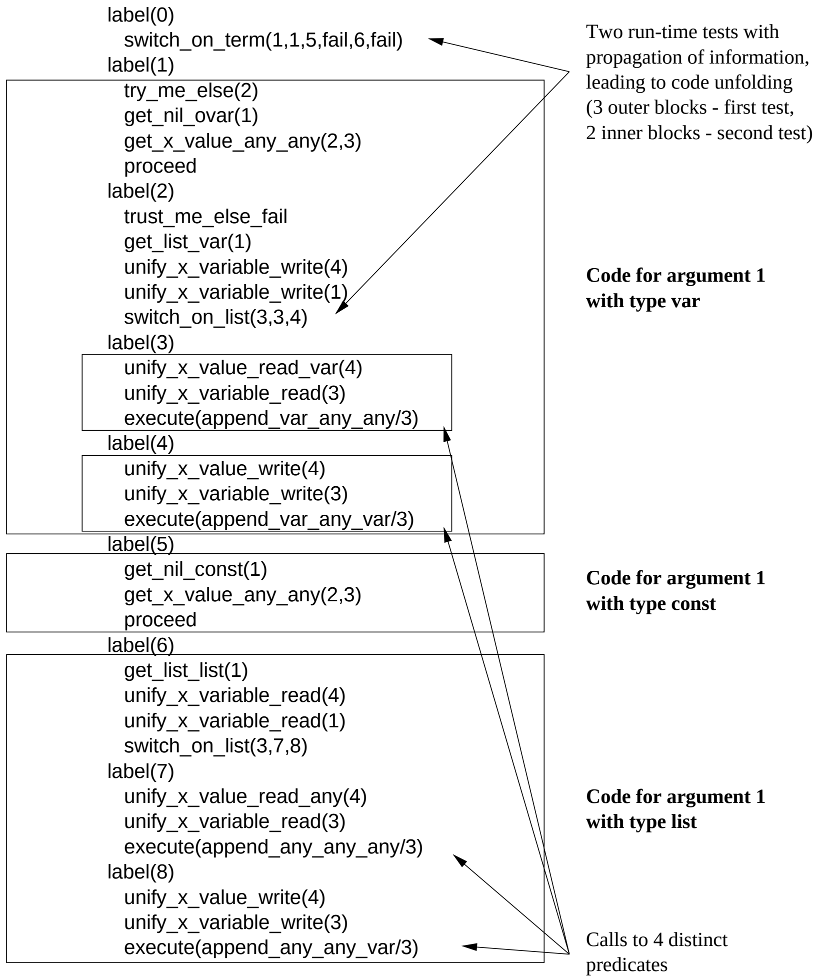

Figure 3: The unfolded WAM code for `append_a_a_a/3`

For each distinct reachable activation pattern it will be created a specialized predicate. In our `append/3` example there are four distinct activation patterns in the calls to `append/3`: `append_a_a_a`, `append_var_a_a`, `append_var_a_var` and `append_a_a_var`.

Whenever a calling instruction is found when applying our specialization algorithm, the new predicate will be created and the algorithm of specialization will be applied to it. The information propagated through the arguments types will be used to simplify the codification. In the `append_var_a_a`, for instance, no `switch_on_term` instruction has to be generated, as the tag of argument 1 is already known to be *var*. If when applying the algorithm to the created predicates, new activation patterns are derived, they will also give rise to predicates and the process will also be applied on them, until no more distinct activation patterns are derived.

Note that splitting a predicate into several predicates makes execution more deterministic. Clause selection is done based on the unification of activation patterns and clauses heads. Indexation is also more powerful as it takes advantage from the knowledge of the types of the arguments, allowing to index on the argument that creates less choice-points[2].

Figure 4 shows the code for the specialized versions generated.

3.3 Some problems with our algorithm

Consider the following Prolog program:

[2]the first argument on the `switch_on_term` instruction of Figure 3 identifies the argument of indexing

append_var_a_a	append_var_a_var	append_a_a_var
label(0) try_me_else(1) get_nil_ovar(1) get_x_value_any_any(2,3) proceed label(1) trust_me_else_fail get_list_var(1) unify_x_variable_write(4) unify_x_variable_write(1) switch_on_list(3,2,3) label(2) unify_x_value_read_var(4) unify_x_variable_read(3) execute(append_var_a_a/3) label(3) unify_x_value_write(4) unify_x_variable_write(3) execute(append_var_a_var/3)	label(0) try_me_else(1) get_nil_ovar(1) get_x_value_any_ovar(2,3) proceed label(1) trust_me_else_fail get_list_var(1) unify_x_variable_write(4) unify_x_variable_write(1) get_list_var(3) unify_x_value_write(4) unify_x_variable_write(3) execute(append_var_a_var/3)	label(0) switch_on_term(1,1,3,fail,4,fail) label(1) try_me_else(2) get_nil_ovar(1) get_x_value_any_ovar(2,3) proceed label(2) trust_me_else_fail get_list_var(1) unify_x_variable_write(4) unify_x_variable_write(1) get_list_var(3) unify_x_value_write(4) unify_x_variable_write(3) execute(append_var_a_var/3) label(3) get_nil_const(1) get_x_value_any_var(2,3) proceed label(4) get_list_list(1) unify_x_variable_read(4) unify_x_variable_read(1) get_list_var(3) unify_x_value_write(4) unify_x_variable_write(3) execute(append_a_a_var/3)

Figure 4: WAM code for the specialized versions generated

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

q(a).
```

The WAM code for this program is:

```
p/1:  allocate(1)                q/1:  get_constant(a,1)
      put_y_variable(1,1)              proceed
      call(q/1)
      put_y_value(1,1)
      deallocate
      execute(p/1)
```

Applying our process of specialization to this code, will deduce that the only activation pattern to `q/1` will be an unbound argument in register 1, allowing the `get_constant` instruction to be implemented by a store instruction. However our algorithm does not conclude anything about the activation patterns of the predicate `p/1`, although it is not difficult to see that it will always be just one, with constant `a` in argument 1.

The reason for this unconclusion is the `call` instruction. This instruction has the inconvenient effect of promoting the unbound variables present in its arguments to the type *any*. As a result an important loss of precision in the information propagated occurs. Clearly this promotion has to be done, as we do not collect and propagate exit information, and there is no way to know what the called predicate will do to unbound variables.

To avoid this loss of precision a simple analysis algorithm is used to derive the predicates argument types on exit. After a call instruction, instead of promoting unbound variables to the type *any*, we update their type based on the information about the exit types derived by the analysis algorithm.

This analysis algorithm is extremely simple, working with a bottom-up strategy, starting with facts and iteratively closing goals in clauses, until the clauses can be closed and information about the head of the clause derived.

Note that the unfolding algorithm could work without this analysis, but more specialized predicates can be created if the analysis on the exit types of predicates is performed.

Another problem with our algorithm of specialization is the code expansion. Our specialization process generates, in the worst case, a new predicate to every combination of types in the arguments.

This can lead to a major code growth, that may not be supported in programs of large size. Also, a large number of predicates in a program have no advantages in being unfolded, i.e. multi-specialized, as they are not relevant to performance. It is not unusual that a 1000 lines program spends 90% of the time doing append. Our system allows the user to specify a list of the predicates to unfold, granting him with the choice of the ratio performance/code size[3]. If the user doesn't want to choose which predicates to unfold, the compiler chooses them, by detecting predicates which define cycles of a predefined order, being by that reason, good candidates to unfold. In the benchmark programs we have compiled this cycle strategy has given reasonable results. A more accurate strategy would be based on profiling as do some compilers of other programming languages.

Some combinations of argument types give rise to a specialized code that is identical to the code of another combination of types that generalizes the one just created. In this cases we eliminate this predicate and replace the calls to it by calls to the predicate that generalizes it.

Another problem with our algorithm is that it has some limitations in the types deduced, because information is collected from WAM instructions. Our domain of types is any, var, ovar[4], struct, list, const and integer, which basically reflects the specializations present in WAM instructions. There is no type to inform on dereferencing length as in Parma or Aquarius. This kind of information has to be deduced based on analysis. To extend the types of our domain we propose to follow the strategy described in the next subsection.

3.4 From WAM unfolded code back to Prolog

Prolog compilation has improved largely since the WAM development [Van94], and it is, at least, at the same level of technology of imperative languages compilation. The speed of execution, however, does not follow this equality. This is due to the programming style of declarative languages, were the absence of "how to get it" concerns, leads programmers to write "what they want" with a generality that is not correct in the context of the particular program execution.

Our multiple specialization algorithm solves this problem, as it creates execution driven specialized procedures.

The Prolog code of a program can be easily reconstructed from the WAM code, and some compilers allow this to be done. It seemed interesting to us to see the Prolog programming style of a program reconstructed from our WAM unfolded code. From this source program it should not be difficult to a no-analysis compiler to achieve an efficient code.

The reconstructed Prolog code of the predicates `append_a_a_a/3` (WAM code in Figure 3) and `append_var_a_var/3` is presented in Figure 5.

The `$ovar`, `$var`, `$list` and `$const` are run-time testing instructions. In the predicate `append_var_a_var/3` these run-time tests are eliminated. Note that the unification (=) will have different implementations, depending on the information available about the terms to unify. When the left side is known to be *var* this unification is implemented by a store instruction. When the two terms to unify are both lists then a specialized unification will be performed.

The Prolog code obtained from this reconstruction is clearly difficult to generate by a programmer. The generation of such code by the compiler has, however, some interesting aspects. The Prolog obtained is a flat and clear code, suitable to the application of analysis algorithms, similar to the Kernel Prolog used by Aquarius analysis. Much more important is the presence of the `$ovar`, `$var`, `$list` and `$const` instructions which gives information about the type of local variables, thus simplifying analysis. Also very important is the fact that the execution path is unfolded, deriving many and more precise (from the analysis point of view) execution paths. Therefore we claim that running abstract interpretation algorithms over this code, will significantly improve both precision and execution time of analysis.

[3] Although this is not a monotonic ratio, due to cache problems

[4] var that needs trailing

```
append_a_a_a(A,B,C) :-
        $ovar(A),
        A= [ ],
        C=B.

append_a_a_a(A,B,C) :-
        $var(A),
        A= [D|E],
        $list(C),
        C=[D|F],
        append_var_a_a(E,B,F).

append_a_a_a(A,B,C) :-
        $var(A),
        A= [D|E],
        $var(C),
        C=[D|F],
        append_var_a_var(E,B,F).

append_a_a_a(A,B,C) :-
        $const(A),
        A= [ ],
        C=B.

append_a_a_a(A,B,C) :-
        $list(A),
        A= [D|E],
        $list(C),
        C=[D|F],
        append_a_a_a(E,B,F).

append_a_a_a(A,B,C) :-
        $list(A),
        A= [D|E],
        $var(C),
        C=[D|F],
        append_a_a_var(E,B,F).

append_var_a_var(A,B,C) :-
        A= [ ],
        C=B.

append_var_a_var(A,B,C) :-
        A=[D|E],
        C=[D|F],
        append_var_a_var(E,B,F).
```

Figure 5: Prolog code reconstructed from WAM unfolded code

4 WAM instructions implementation

The implementation of WAM instructions has two main approaches: constructing an emulator or generating native code. The first option has a more simple implementation, and is portable to different machines. The disadvantage is poor efficiency because of the cycle *fetch*, *decode* and *execute*. The second option is clearly used when efficiency is the main issue, sacrificing portability and simplicity of implementation.

Another approach has been implemented in the `wamcc` [CD95] compiler. The idea is to introduce a new intermediate step between the WAM code generation and the native code. This intermediate step consists in translating the WAM code to C code. The native code generation is then obtained thanks to the C compiler.

This approach has clearly some advantages. In the end we have native code, so performance can be high. Portability is achieved as a C compiler is available to every machine. Also, it is much more simpler to generate C code than to generate native code.

The disadvantage with this approach is that the fact of having native code in the end, does not imply high performance by itself. The native code generation just exposes the bare machine, but is up to the compiler to take advantage from the low level availability.

In the introduction we referred that high performance Prolog implementation was based on two aspects: program analysis and native code generation. We have efficiency as our main goal, and we showed in the above sections how specialization could be achieved without requiring global program analysis. We opted to generate C code in the `wamcc` style, to show that we could also eliminate native code generation and still achieve the performance of native code systems. The portability and modularity also contributed to this option. Remember that with our multiple specialization modular programming is well accepted, as an external unknown activation pattern will be handled by the always present general case, that leads subsequent calls to the correct specialized version.

Another important reason to the C generation option was the simplification of our work. We have generated C code using the `wamcc` framework, taking advantage of an intelligent control flow codification [CD95]. The remaining codification solutions of `wamcc` are quite conventional and high-level, and we have lost performance by reusing them, even with our more low-level complemented WAM instructions.

The C generation is an alternative to the emulation or native code generation, but to an implementation aiming maximum efficiency it is not a good option as it disallows many important low-level optimizations.

A portable assembler language, C–, with a syntax similar to C but with added low-level features, like for instance a no-alias directive, which defines the independence of two variables, has been proposed in [PJNO97], and seems like a good alternative to achieve performance without having to generate native code.

4.1 Generating C code

The major problem to solve in coding WAM to C is the codification of the control execution. As the WAM code is flat, the execution control is done by jumps, which can be direct or indirect (the destination is the contents of some variable). The C language does not offer much to express such jumps, specially in a low level manner. Other compilers of committed choice languages addressed the problem of translating the execution control, like Janus [GDBD92], KL1 [CFS94] or Erlang [Hau94]. Their methods either generated just one C function in order to be able to implement jumps as gotos (with different solutions to implement non-ANSI C indirect gotos), or implemented jumps as functions calls that could return to a supervisor function.

The `wamcc` system had the objective of translating a WAM jump by a native code jump. Because the code has to be decomposed in several functions, these jumps should reach blocks of code inside a function. The `wamcc` solution was then to insert a label at the entry of each function, thanks to `asm(...)` directives. To manipulate the addresses of those labels `wamcc` declared a prototype of an external function with the same name (address) of the label. The compiler then generates an instruction with an hole that will be filled by the linker with the knowledge of all inserted labels.

Besides the codification of the control execution, there is no relevant implementation issue in the way we generate C code. Basically WAM instructions are macro expanded to their C code implementation. In this implementation the important aspect is the simplifications allowed by the different information that complements standard WAM instructions. In Figure 6 we present the simplified implementations of the distinct possible cases of the `get_list` instruction.

Note the number of run-time tests that are eliminated in the specialized versions. Note also how a `TrailBind` instruction is implemented by a simple store in the `get_list_var` instruction.

4.2 Specializing the execution model

Besides the specialization of WAM instructions, the WAM execution model can also be specialized to take advantage of the information available. For instance, we have created specialized structures for choice-points that do not need local variables. Also, argument registers known to be integers use the native integer representation, eliminating tagging and untagging operations. With this native representation we have gained almost 10% in the execution time of tak and fib benchmarks.

The multiple specialization approach makes a little harder to do this kind of model specialization, in relation with a single specialization approach, because different versions of predicates have different argument types, and not all of them may use the native represention. The solution is to tag a native representation before a generic predicate is called, and to convert to a native representation a tagged value when a specialized version is called. Inside the specialized predicate the operations will be done over the native representation, and will be tagged before returning.

5 Performance evaluation

We will now evaluate the performance of our system, comparing it with the `wamcc` basic system and with the Sicstus compiler, emulated and native. We also present results for the code expansion of our multiple specialization strategy, comparing the sizes of the files generated by `wamcc` and by our system (ms-wam).

Table 1 shows the execution times of ms-wam 1.0, `wamcc 2.22`, emulated Sicstus 3.0 and native Sicstus 3.0 on a set of benchmarks. Timings are in seconds measured on a Sun Ultra Sparc 248 Mhz, using gcc 2.8.1 with the `-O2` option, both for ms-wam and for `wamcc`.

get_list(A)	if (!Get_List(A(a))) goto lab_fail;
get_list_var(A)	*(UnTag_REF(A(a)))=Tag_Value(LST,H); S=WRITE_MODE;
get_list_ovar(A)	TrailBind_UV(UnTag_REF(A(a)),Tag_Value(LST,H)) S=WRITE_MODE;
get_list_list(A)	S=(WamWord *) UnTag_LST(A(a)) + OFFSET_CAR;

Get_List(A) function:

```
Bool Get_List(WamWord start_word)
{
 WamWord word, tag, *adr;
 Deref(start_word,word,tag,adr);
 switch(tag)
    {
      case REF:
            TrailBind_UV(adr,Tag_Value(LST,H))
            S=WRITE_MODE;
            return TRUE;

      case LST:
            S=(WamWord *) UnTag_LST(word) + OFFSET_CAR;
            return TRUE;
    }
 return FALSE;
}
```

Figure 6: WAM code for the specialized versions generated

The most important and relevant comparison is between `wamcc` and ms-wam. Our system was constructed using the C implementation of the WAM from `wamcc`. Therefore the average speed-up of 2.6 indicates the performance gain of the multiple specialization strategy described in this paper.

Also important is the comparison with the de facto reference for performance of Prolog systems, the Sicstus compiler. On average, ms-wam is 10% faster than Sicstus 3.0 generating native code. Yet, the C generation of ms-wam has the advantage of running on multiple architectures as emulated Sicstus, over which it has an average speed-up of 3 times. Abandoning the C generation and directly generating native code will bring a considerable improvement on the performance of ms-wam, achieving the performance of the Aquarius system.

In table 2 we present results for the code growth of our multiple specialization technique. We present the size in Kbytes of the object file generated by `gcc` and the executable file after linking, both for `wamcc` and for ms-wam.

The code expansion of our multiple specialization technique is quite modest, representing less than 5% in the executable files sizes and about a 1.5 expansion factor in the object files. This is due to the smaller implementation of the specialized predicates, which balances the larger number of predicates. In some cases, where the number of versions created is small, the multi-specialized object file is even smaller than the `wamcc` file (fib and query).

6 Conclusions

Program specialization will always depend on program analysis. Therefore, simplification of program specialization will always depend on analysis simplification. Local analysis techniques are much more simpler than global analysis techniques. The main effort of our work was to achieve the global analysis benefits through local analysis. Our multiple specialization strategy is the way to propagate local analysis information through procedure calls, allowing global program optimization.

Program	**ms-wam 1.0**	**wamcc 2.22**	**Sicstus 3.0 (emulated)**	**Sicstus 3.0 (native)**
browse	0.190	0.550	0.760	0.240
fib	0.100	0.230	0.360	0.120
ham	0.240	0.590	0.550	0.230
nrev	0.820	3.370	2.480	0.520
qsort	0.300	0.610	0.630	0.240
queens(16)	0.150	0.320	0.530	0.140
queens_n(8)	0.040	0.100	0.100	0.040
queens_n(10)	0.660	1.980	1.730	0.600
query	0.140	0.390	0.560	0.340
tak	0.030	0.060	0.100	0.030
zebra	0.020	0.040	0.040	0.020
average speed up of ms-wam		2.6	3.0	1.1

Table 1: ms-wam versus `wamcc` and Sicstus

Program	**ms-wam 1.0 object file**	**wamcc 2.22 object file**	**ms-wam 1.0 executable file**	**wamcc 2.22 executable file**
browse	60.1	33.7	358.9	339.7
fib	6.5	7.8	324.7	305.8
ham	43.8	31.3	348.3	335.9
nrev	23.1	16.6	320.0	312.5
qsort	27.8	14.5	342.7	310.1
queens(16)	17.6	12.1	316.1	308.7
queens_n(8)	29.8	17.1	326.4	310.7
queens_n(10)	29.8	17.1	326.4	310.7
query	26.1	26.5	341.1	337.1
tak	12.7	8.0	312.0	305.7
zebra	43.1	19.6	347.0	328.9
Code growth	1.55		1.04	

Table 2: Code growth

The local analysis we perform is much easier to do at the WAM level. In fact, part of this analysis is already done in the Prolog to WAM compilation, giving rise to specialized instructions. We reuse this analysis promoting its instruction scope to an inter-procedural and global analysis.

The complexity of our ms-wam system is not higher than the complexity of the `wamcc` basic system, a very readable and extendable compiler. In spite of this simplicity, we out-perform native Sictsus 3.0. This is even more impressive because we are generating C code as target language. Thus we are not limited to the Sparc architecture as Sicstus, running in every architecture which has a C compiler.

The purpose of this work is not to be an alternative to the abstract interpretation techniques in order to achieve code specialization. Abstract interpretation techniques are used in our system. We plan to extend the optimizations we perform by using more abstract interpretation algorithms. However, it is important to simplify such algorithms, simplifying, for instance, the analysis domains. Our analysis of exit types is simple because it is just an exit types analysis. What makes this kind of analysis algorithms complex and expensive is the association of exit and entry types analysis. The unfolded Prolog source code generated will also allow important simplification of abstract interpretation algorithms.

The integration of our multiple specialization technique into a Prolog compiler is simple. In fact, a relevant aspect of this specialization technique is that it is done at the implementation level. It depends on the process of compilation and in the generation of WAM code. The intermediate language that is generated is almost WAM code, complemented with some type and mode information. This allows the instructions of this intermediate language to be easily implemented in the WAM engine of the original compiler.

Acknowledgments

The first author is thankful to PRAXIS ("Sub-Programa Ciência e Tecnologia do 2o. Quadro Comunitário de Apoio") for their financial support.

References

[AK91] Hassan Aït-Kaci. *Warren's Abstract Machine — A Tutorial Reconstruction.* MIT Press, 1991.

[BF96] Dominique Boucher and Marc Feeley. Abstract compilation: A new implementation paradigm for static analysis. In Tibor Gyimothy, editor, *Proceedings of the 6th International Conference on Compiler Construction*, pages 192–207, Linköping, Sweden, 1996. Springer.

[BJ88] Maurice Bruynooghe and Gerda Jenssens. An instance of abstract interpretation integrating type and mode inferencing. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 669–683, Seattle, Washington, 1988. MIT Press.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977.

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *The Journal of Logic Programming*, 13(1, 2, 3 and 4):103–179, 1992.

[CD95] Philippe Codognet and Daniel Diaz. wamcc: Compiling Prolog to C. In *12th International Conference on Logic Programming.* MIT Press, 1995.

[CFS94] 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*, number CIS-TR-94-04. Department of Computer Information Science, Oregon, 1994.

[DW88] Saumya K. Debray and David Scott Warren. Automatic mode inference for prolog programs. *The Journal of Logic Programming*, 5(3):78–88, September 1988.

[GDBD92] 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, 1992. "MIT Press".

[Hau94] B. Haussman. Turbo Erlang: Approaching the Speed of C. In Evan Tick, editor, *Implementations of Logic Programming Systems.* Kluwer, 1994.

[JL89] Dean Jacobs and Anno Langen. Accurate and efficient approximation of variable aliasing in logic programs. In Ewing L. Lusk and Ross A. Overbeek, editors, *North American Conference on Logic Programming*, pages 154–165. "MIT Press", 1989.

[JLHW90] Dean Jacobs, Anno Langen, and William H. Winsborough. Multiple specialization of logic programs with run-time tests. In Péter Szeredi David H. D. Warren, editor, *Proceedings of the 7th International Conference on Logic Programming*, pages 717–731, Jerusalem, Israel, 1990. MIT Press.

[KN90] A. Krall and U. Neumerkel. The Vienna Abstract Machine. In *International Workshop in Programming Language Implementation and Logic Programming (PLIP'90)*, number 456 in Lecture Notes in Computer Science, pages 121–135. Sweeden, Springer-Verlag, 1990.

[Mel86] C. S. Mellish. Abstract interpretation of prolog programs. In *Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science. Springer-Verlag, 1986.

[PGB90] Jonh P. Gallagher and Maurice Bruynooghe. The derivation of an algorithm for program specialisation. In Péter Szeredi David H. D. Warren, editor, *Proceedings of the 7th International Conference on Logic Programming*, pages 732–746, Jerusalem, Israel, 1990. MIT Press.

[PJNO97] Simon Peyton Jones, Thomas Nordin, and Dino Oliva. C–: a portable assembly language. In *[To appear in the proceedings of IFL'97.]*, St. Andrews, Scotland, 1997.

[Tay91] A. Taylor. *High Performance Prolog Implementation.* PhD thesis, University of Sydney, June 1991.

[TL92] Jichang Tan and I-Peng Lin. Compiling dataflow analysis of logic programs. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 106–115, San Francisco, California, 1992. SIGPLAN.

[Van90] P. Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, University of California at Berkeley, November 1990.

[Van94] P. Van Roy. 1983-1993: The Wonder Years of Sequential Prolog Implementation. *The Journal of Logic Programming*, 19/20, May/July 1994.

[War83] David H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, 1983.

[Win89] W. H. Winsborough. Path-dependent reachability analysis for multiple specialization. In Ewing L. Lusk and Ross A. Overbeek, editors, *North American Conference on Logic Programming.* "MIT Press", 1989.