Controlling Code Expansion in a Multiple Specialization Implementation

Michel Ferreira* Luís Damas

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

Abstract. Program specialization can be *single* or *multiple*, depending if each procedure of the program is specialized into one or more versions. Multiple specialization has severe consequences on code growth. In order to keep a multiple specialization strategy implementable on real-size programs, large and efficient selection of the predicates to be multiply specialized must be done. We present a multiple specialization Prolog compiler that bases the control of code growth on a predicate selection driven by a WAM-level profiler. Such code expansion control is also achieved by finding the optimal set of versions for a particular predicate. In this case a non-strictly-identical collapsing of versions is performed, also at the WAM-level. The results show a time per predicate distribution in Prolog programs where partial multiple specialization can importantly speed-up execution, while keeping code size reasonable.
Keywords: Prolog Profiling, Prolog Compilation, Multiple Specialization.

* The first author is thankful to PRAXIS and Fundação para a Ciência e Tecnologia for their financial support.

1 Introduction

The main problem with code specialization is code growth. Although the idea of writing a specialized version of a predicate is to implement the predicate in a more simplified manner, and such simplification is also reflected on code size, the overall process of specialization causes code to enlarge. In our `ms-wam` [5] system, we have a large number of WAM specialized instructions, which are simplifications of the general instructions, due to the knowledge of one or more of the argument types. Such increase in WAM instructions is reflected on the size of the run-time library. In the `wamcc` [3] basic system, in which our system is based, this run-time library is of 300 Kb, being of 500 Kb in `ms-wam`. Such increase is reflected on the executable files produced by the compiler.

In *single specialization* compilers such code growth is much less relevant than in *multiple specialization* [8, 6, 13] systems. In `ms-wam`, which is a multiple specialization compiler, the major cause of code growth is not the larger run-time library, but the multiplication of a predicates implementation code by a number of specialized versions. The code of each specialized version is often much smaller than the generic implementation code of the predicate, and the multiplication is attenuated, but it still is proportional to the number of versions.

The number of versions generated for each predicate is a function of the arity of the predicate and of the analysis domain complexity. The elements of the analysis domain are of two types: elements which actually lead to optimizations and elements used to propagate analysis information. If d is the number of elements of the domain and n is the number of elements which do not lead to optimizations, then, in the worst possible case, the number of versions generated for a predicate will be:

$$P_{versions} = (d-n)^a$$

where a is the arity of the predicate. This is a wildly pessimistic number of versions as, due to the structure of the domain, only a pathological predicate could achieve such a variety of activation patterns. It is, though, an indicator of the order of growth of a multiple specialization implementation, revealing its exponential nature.

In order to make a multiple specialization strategy feasible for real-size programs it is necessary to use a strong selection criteria of the predicates to be multiply specialized. In real-size programs only the most critical predicates of a program must be multiply specialized, performing partial multiple specialization instead of total multiple specialization. Further more, not every version of a particular predicate that is generated is relevant for the overall performance of the program. A process of generating the optimal set of versions for a particular predicate is also determinant for the success of a multiple specialization implementation.

In this paper we address these two problems and the solutions we adopted in the context of our `ms-wam` system. For the first problem we base our selection on the results of a (goal-dependent) profiler implemented at the WAM level. A goal-independent selection strategy is also suggested. For the second problem, a collapsing approach of the versions generated based on the optimizations each version allows is presented. The paper is organized as follows: in the next section we present the common approaches to profiling and the particular problems of Prolog profiling; we next present our WAM-level implementation of a Prolog count profiler; profiling results for a set of benchmarks are then presented, followed by the description of a no-execution based selection strategy; the collapsing of versions of a particular predicate is addressed in the next section; we end with some conclusions.

2 Prolog profilers

To optimize partial multiple specialization we must be able to correctly choose which predicates to unfold. We want to generate specialized versions to the predicates which consume more time, where optimizations will result in important speedups in the overall program execution.

To measure the most important parts of a program, profiling techniques have been used in several programming languages to collect information during the execution of the program. The information provided by the profiler can be used for other purposes than just improving speed of execution, such as program debugging and code optimization[2]. For a developer a simple and economical way of obtaining efficient programs is to write the program, obtain execution profiles for it on representative input data, and then rewrite the most often executed parts in an optimized way. The program can be profiled again, returning new information that can be used again in an iterative optimization process. In our case, where the program is automatically transformed to become more efficient and where that transformation has important consequences on code growth, the information from execution profiles is much more useful, even essential. Totally unfolding a 1000 lines program with high-arity predicates, causes code to enlarge to unreasonable limits, eventually leading to a slowdown rather than a speedup due to caching problems. However, it happens very often that a 1000 lines Prolog program spends 90% of the time in just a few predicates. If the profiler detects those few predicates and multiple specialization is done just for those predicates, then performance can be significantly improved with very low impact on code growth.

The implementation of profilers for traditional languages is simple and well understood. There are two types of profilers: *time profilers* and *count profilers.* Time profilers give information about the time spent in each procedure or piece of code. This is the most useful information to the developer in order to improve program efficiency. It is difficult, though, to accurately time profile a program, because the overhead of the timing instructions is large. This overhead, together with the coarse grain of system clocks, can mislead the time estimate of fast procedures. Another problem with the overhead of the timing instructions is that the candidate programs to profiling and consequent optimization are programs which already take long to execute, possibly becoming prohibitive when being profiled.

A solution for those problems is based on statistical sampling of the program execution. An interrupt is generated on a regular or random time cycle, and we annotate the routine or part of the code where the program counter was at that time. This idea can be easily applied to Prolog programs, measuring the time spent inside each predicate definition. The overhead is small and there is no need to change the program code. Errors may be, however, relatively large[7].

Count profilers give information about the number of times a procedure or piece of code is executed. Execution counts do not give time information directly. However, if we know how much time an instruction takes to execute the count information can be converted to time information. The profiler we have implemented is a count profiler which converts the count information to relative time information. Count profilers have some advantages over time profilers, because they are exact and are able to sketch the execution path of the program.

2.1 The control-flow of Prolog programs

Classical languages have a simple unidirectional flow of control, and the sequence of the program instructions is identical to the sequence of execution. In Prolog, however, the execution has two possible directions, forward and backward. In this case the sequence of the program execution is not identical to the sequence of the program instructions. For example, a possible execution sequence for the following program

```
h :- p1,p2,p3.
```

can be

```
h,p1,p2,p1,p2,p1,h   (fail)
```

This more complex mechanism of flow of control introduces some interesting problems, concerning the management of timing in the presence of backtracking and failure. These problems have been addressed and solutions proposed by Debray in [4].

When time profiling a call, the first step is to record the CPU time just before executing the call ($T1$). Immediately after exiting the call, we record the CPU time at that moment ($T2$). The time spent on the call is then $T2 - T1$. For recursive predicates special care must be taken for a predicate not to be considered more than once, marking for each predicate whether or not a call for it is already being timed.

Another problem is to handle backtracking and failure correctly in the timing of a call. For the example above, the schema of the control flow of the call to `p1` is presented in Figure 1.

Assume `p1` is first called at time $T1$ exiting with success for the first time at time $T2$. Execution proceeds calling `p2`, and suppose `p2` fails. `p1` is then re-entered through backtracking at time $T3$ and succeeds for the second time at time $T4$. `p2` is called and fails again, re-entering execution in `p1` at time $t5$.

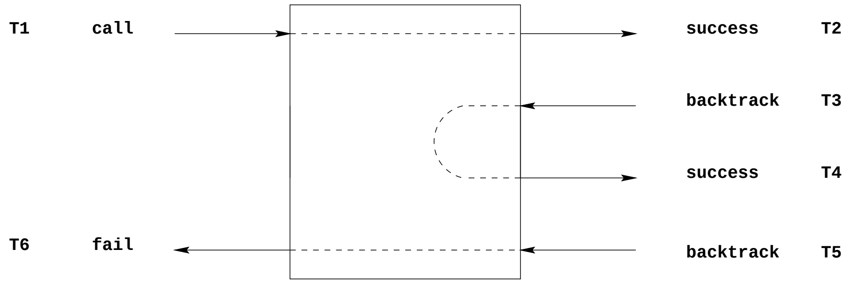

Fig. 1. Prolog control flow in a call

Suppose there are no more alternatives in `p1` and `p2`, and the call fails at time $T6$.

The time spent in `p1` is

$$T = (T2 - T1) + (T4 - T3) + (T6 - T5)$$

However, if we only record the CPU time of the moment of the first entry in `p1` ($T1$), as would be sufficient for unidirectional control-flow languages, the time charged to `p1` will be

$$T = (T2 - T1) + (T4 - T1) + (T6 - T1).$$

Thus, to correctly measure the time spent in a call we need to record not only the time execution first entered the predicate, but also the time execution last entered the predicate, from a call, or re-entered it via backtracking.

We must also have special concerns about a call that fails. Recording just the CPU time immediately after exiting the call (after succeeding) is not sufficient, as we also need to record the time execution exit the predicate via failure, not to underestimate the amount of time spent in the call. Without this consideration the underestimation of the time spent in the call would be equal, in the above example, to $T6 - T5$.

Looking to Figure 1 we can identify four types of arrows entering and exiting the box of predicate `p1`: arrows entering `p1` from the left; arrows exiting `p1` to the right; arrows entering `p1` from the right; and arrows exiting `p1` to the left. These arrows types correspond, in the box-model of Byrd [1], to the four ports *call*, *exit*, *redo* and *fail*, respectively. In [9], Matos describes an interesting Prolog count profiler which displays valuable information, by defining four count values (called *currents*) based on these four ports. Besides showing the values of these currents, the profiler also evaluates *intrinsic* predicate characteristics based on a matrix model, that detail the origin of each current value.

3 WAM-level profiling

Both the Prolog profiler described by Debray [4] and the X-Profiler of Matos [9] are implemented over the Prolog source code, rewriting the program clauses of the predicates to be profiled to call special predicates before and after executing the call. Those special predicates are responsible for the timing or counting of predicates.

Our approach is different, as the profiling is performed and implemented at the WAM [12] level. A first advantage of such approach is that the compiler optimizations (such as clause indexing) that could mislead the true flow of control on the Prolog source profiling implementations, are explicit in the WAM code.

A second reason to implement a profiler at the WAM level is the higher level of detail introduced by WAM instructions in the codification of a predicate, leading to more accurate conversion of count information to time information, as, due to the low-level of the WAM instructions, we can more or less fairly estimate how much time each instruction takes to execute.

A third reason is that the WAM code is much more similar to the code of traditional programming languages, with a simpler control-flow and the traditional problems of Prolog profiling disappear.

A last reason is to perform profiling at a level that can be used with the process of multiple specialization, contributing to obtain an optimal set of versions for a particular predicate.

The implementation of our WAM profiler is extremely simple. The basic idea is to give a certain weight to each WAM instruction. The weight is meant to represent a relative time that the instruction takes to execute. We obtained this weight by inspecting the implementing code of each instruction, counting the number of assignments, comparisons and operations performed. The values of these weights can be seen as an equivalent measure to the CPU instruction cycles each WAM instruction would take to execute.

Each instruction will increment a counter with its weight. The counter can be a single global counter translating, upon program termination, the total execution time, or it can be a number of counters, one per predicate, translating the time spent in each predicate, or it can be a counter per instruction type, for instance, grouping instructions in indexing instructions, control instructions, unification instructions, etc, giving information about particular characteristics of the program. Such characterization is important because in Prolog performance depends of different aspects, such as data structure implementation for some programs, and control for others. We are basically interested in having a counter per predicate, to order predicates by their relevance.

A problem with giving a fix weight to a WAM instruction is that the complexity of the instruction varies, depending on the input. For instance, a large number of WAM instructions have an intrinsic dereferencing operation. This dereferencing operation is of variable complexity, depending on the chain length to be dereferenced. Some other instructions operate either on write or read mode, and their complexity varies depending on this execution mode. This variations are, however, of small relevance. The average length of reference chains has been shown to be very small. Also, the differences between read and write mode are not very significant. There is, however, an important problem when giving a fixed weight to a WAM instruction, and this problem has to do with unification instructions, namely unification between non-variable terms. The problem is that the terms to be unified can have very different depths and arities, and such depth and arity linearly increases the complexity of the unification instructions. Moreover, unification is everywhere in Prolog program, making crucial the correct measurement of such operation. WAM instructions like `get_x_value` and `unify_x_value` call a general unification routine, `Unify`, whose code we present in figure 2.

```
Bool Unify(WamWord start_u_word,WamWord start_v_word)
{
 WamWord u_word,v_word,u_tag,v_tag,*u_adr,*v_adr; int i;
 Deref(start_u_word,u_word,u_tag,u_adr)
 Deref(start_v_word,v_word,v_tag,v_adr)
 if (u_tag==REF)  {
     if (v_tag==REF) { ... }
     else { ... }
     return TRUE; }
 switch(v_tag) {
     case REF:
         ...
         return TRUE;
     case INT:
     case CST:
         return (u_word==v_word);
     case LST:
         if (u_tag!=LST) return FALSE;
         u_adr=UnTag_LST(u_word);
         v_adr=UnTag_LST(v_word);
         if (u_adr==v_adr) return TRUE;
         return Unify(Car(u_adr),Car(v_adr)) &&
                Unify(Cdr(u_adr),Cdr(v_adr));
     default:
         if (u_tag!=STC) return FALSE;
         u_adr=UnTag_STC(u_word);
         v_adr=UnTag_STC(v_word);
         if (u_adr==v_adr) return TRUE;
         if (Functor_And_Arity(u_adr) != Functor_And_Arity(v_adr))
             return FALSE;
         i=Arity(u_adr);
         do{
             --i;
             if (!Unify(Arg(u_adr,i),Arg(v_adr,i))) return FALSE;
         }while(i);
         return TRUE;}
}
```

Fig. 2. General unification routine

In the cases where one of the terms is unbound the unification is reduced to making the unbound term point to the other. This has a fixed cost that does not introduce errors in our count profiling, and we have ommited these cases in the `Unify` routine of figure 2. From this figure it can be seen that unifying lists and structures is made recursively. In the case of structures the recursion is iterated in the number of arguments. The size of such recursion and iteration must be considered when profiling the program to obtain accurate results. Therefore, the instructions `get_x_value`, `get_y_value`, `unify_x_value`, `unify_y_value`, `unify_x_local_value`, and `unify_y_local_value`, can not be accurately count profiled in the same manner as the other instructions because they depend on the size of the input.

To accurately count profile these instructions we modified the `Unify` routine to return a number representing the size of the recursion and iteration of its execution. The weight of the WAM instruction is then calculated based on the returned value, translating the size of the input. If we wanted more accuracy on our profiling information, the same scheme could be implemented on the instructions which operate in write or read mode, returning different values according to the execution mode.

4 Profiling results for a benchmark set

To the best of our knowledge, there is no published work where profiling results for a typical set of Prolog benchmarks are presented. Even about Prolog profilers implementations there are very few publications, showing the current handicap of Prolog as a real implementation programming language, when compared with other languages where profilers exist and are used as an essential tool for program optimization.

In this section we present results from our WAM profiler on a representative set of Prolog benchmarks. More than just presenting the profiling results for these benchmarks, we want to show that the distribution of time per predicate in Prolog programs follows a pattern that is very adequate for the predicate selection that motivated the development of the profiler. If it is well known that, in traditional programming languages, programs spend the great majority of time in just a few cyclic functions, it can be expected that this also happens with Prolog predicates, but this has to be experimentally checked. The goal of our profiling is to chose a *reasonably small* number of predicates to multiply specialize that represents a *reasonably large* percentage of the execution time of the program. The ratio between the first and the second *reasonably* is determinant in the success of our multiple specialization strategy, and depends on the time per predicate distribution of Prolog programs that is presented next.

The result of executing a program compiled with the `-profile` flag has the output shown in figure 3 for the `queens8.pl` benchmark.

The program is executed and the count profiling is performed. Upon finishing, the profile report that is printed represents the percentage spent in each predicate. In `queens8.pl` the predicate `pair/3` is responsible for almost 50% of the execution time.

Table 2 shows the profile information for 16 representative benchmarks from the set of previously proposed by Van Roy [11]. The columns of the table (`P1,P2,...`) represent the predicates of each benchmark, ordered descendently by time of execution. The values presented are the accumulative percentage of the execution time.

In the first column, the number after the benchmark name indicates the number of predicates of the program. This is the number of predicates after syntactic transformations, like if-then-else constructs elimination, have been performed. The smallest benchmark, `qsort` as only 4 predicates and the largest, `chat_parser` as 158. From table 2 it can be seen that for the benchmarks with

```
[p(1,1),p(2,5),p(3,8),p(4,6),p(5,3),p(6,7),p(7,2),p(8,4)]
time : 610

Profile information:

00.00 --> exe_1/0
00.00 --> q/0
00.00 --> q8/0
00.81 --> q/2
09.32 --> perm/2
09.66 --> safe/2
10.25 --> test/2
12.34 --> nd/2
15.00 --> sel/3
42.61 --> pair/3
```

Fig. 3. Profiling of the `queens8.pl` benchmark

less than 15 predicates in the 4 most important predicates we have more than 80% of the execution time. The most important predicate represents always more than 17% of the execution time, even in programs with more than 150 predicates. Ordering the predicates by their importance, the distribution of time per predicate follows a clear logarithm pattern, as can be seen by figure 4, where this distribution is illustrated for a medium size Prolog program, the `chat_parser` benchmark.

Considering the size differences of the benchmarks, we analyzed the profiling information with respect to the *percentage of the number of predicates*, using 10 measuring points from 10% to 100%, in order to reach a meaningful average information that is summarized in table 3.

On average 58% of the total execution time is concentrated on 10% of the predicates. Multiply specializing this small part of the program can significantly improve the efficiency of the program. Multiple specialization could be taken further, and aplyed to 20% of the predicates, which represent on average more than 75% of the execution time. If performance is critical and the program being considered is not large we could select the 30% most important predicates, where almost 90% of the total execution time is present. Further than that only marginal improvements would be achieved, or the code growth could even introduce some slow-downs due to caching problems.

Note that the results from table **??** accentuate as the programs become larger. Considering the last 3 programs which have more than 40 predicates, the average execution percentage on 20% of the predicates is 83.4%. Considering the first 7 programs which have less than 10 predicates, this percentage is of 64.6%.

Program	P1	P2	P3	P4	P5	P6	P7	P8	P9
qsort(4)	73.32	95.81	99.76	100	-	-	-	-	-
query(6)	35.36	69.00	84.82	99.51	100.0	100.0	-	-	-
derive(7)	86.24	89.68	93.12	96.10	98.79	99.02	100.0	-	-
zebra(7)	52.60	78.99	96.40	99.95	99.97	99.99	100.0	-	-
serialise(8)	49.60	73.50	85.81	91.31	96.35	99.46	99.84	100.0	-
mu(9)	25.95	50.70	71.00	84.16	91.35	96.01	99.61	100.0	100.0
fast_mu(9)	56.12	80.82	91.41	97.19	98.81	99.22	99.56	99.79	100.0
crypt(10)	57.90	89.77	94.63	97.71	98.47	99.18	99.97	100.0	100.0
meta_qsort(11)	41.03	75.38	83.77	89.51	94.35	97.55	99.97	100.0	100.0
queens8(11)	42.61	57.61	69.95	80.2	89.86	99.18	100.0	100.0	100.0
nreverse(12)	91.20	99.75	99.91	99.94	99.97	100.0	100.0	100.0	100.0
prover(15)	40.76	60.04	74.22	86.76	93.27	96.41	98.36	99.47	99.65
browse(19)	51.83	95.00	98.35	99.01	99.25	99.46	99.59	99.71	99.80
boyer(29)	50.24	75.07	86.96	95.33	96.93	97.84	98.44	99.01	99.39
sdda(42)	30.97	60.92	69.10	75.99	80.08	83.88	86.70	88.97	90.60
nand(78)	17.41	29.30	38.67	47.55	53.83	57.86	61.56	65.17	68.60
chat_parser(158)	25.02	29.98	34.58	38.82	42.70	46.27	48.79	50.93	52.80

Table 1. Time percentage per predicate

5 No execution based selection

Without doubt profiling information is the most useful information to identify the hot spots of a program. However, not every program available for compilation is suitable for execution. Execution is goal directed, implying the existence of an entry point. This entry point defines a determinate execution path, giving accurate information about the number of times each procedure is executed. In a goal-independent perspective, however, no entry points are to be used, as we want to make the analysis independent of a particular entry goal pattern. Without entry points a program cannot be executed and, consequently, cannot be profiled. Therefore, profiling is applicable in a goal-dependent perspective and not in a goal-independent perspective.

In order to satisfy a goal-independent perspective we need to be able to perform a non execution-based selection. Such selection is, clearly, not objective and accurate, as only execution can determine the number of times each part of the program is executed. Although a particular entry goal pattern can introduce some errors in the profiling information, that might be different if another entry goal pattern has been used, the error is clearly much smaller than if no execution path is used. However, even if greater errors are expected, a no execution based process of selecting predicates to multiply specialized must be available to be used with a goal-independent multiple specialization perspective.

Looking to the WAM code of a predicate important information regarding its complexity can be easily extracted. The number of clauses and the indexing scheme, the head unification terms structure, the number of goals and its arguments complexity, etc. Such information is much more easily quantified at the