Tracing Prolog Programs by Source Instrumentation is Efficient Enough

Mireille Ducassé
IRISA/INSA, Campus Universitaire de Beaulieu
F - 35042 Rennes Cedex, France, email: ducasse@irisa.fr

Jacques Noyé
École des Mines de Nantes, 4, rue Alfred Kastler, La Chantrerie
F- 44307 Nantes Cedex 03, France, email: noye@emn.fr

Abstract

Tracing by automatic program source instrumentation has major advantages over compiled code instrumentation: it is more portable, it benefits from many compiler optimizations, it produces traces in terms of the original program, and it can be tailored to specific debugging needs.

The usual argument in favor of compiled code instrumentation is its supposed efficiency. We have compared the performance of two operational low-level Prolog tracers with source instrumentation. We have executed classical Prolog benchmark programs, collecting trace information without displaying it. On average, collecting trace information by program instrumentation is about as fast as using a low-level tracer in one case, and only twice slower in the other. This is a minor penalty to pay, compared to the advantages of the approach.

To our knowledge, this is the first time that a quantitative comparison of both approaches is made for any programming language.

1 Introduction

There are four basic ways to trace executions [4]: manual program source instrumentation, automatic program source instrumentation, instrumentation of meta-interpreters, and compiled code instrumentation. Manual program source instrumentation is tedious and error-prone, and instrumentation of meta-interpreters is notoriously too inefficient. No comparison of automatic program source instrumentation and compiled code instrumentation exists.

Source instrumentation has major advantages over compiled code instrumentation. Firstly, it is more portable than compiled code instrumentation as it is independent from the low-level implementation details of any particular compiler. Secondly, the instrumented programs are compiled and therefore they can benefit from many of the available compiler optimizations.

Thirdly, the instrumented program is traced at the level of the original program. Specht [11] showed that this feature is particularly adapted to tracing deductive databases programs as they are heavily transformed during compilation. This is also the case for plain Prolog. Indeed, many optimizations are made at compile time and the resulting code is quite different from the original one. A low-level tracer cannot always reverse the optimizations to give the information in terms of the original program.

Fourthly, source instrumentation can be tailored to specific debugging needs, and therefore be more efficient. Indeed, in general not the whole trace information is needed for a given application or a given debugging request. A good example is an abstract tracer of a programming language whose compiler is implemented in Prolog. Producing the Prolog trace and then filtering it to get the abstract trace is not the most efficient way of processing. The instrumentation can be adapted to directly trace at the abstract level.

While source instrumentation has advantages over compiled code instrumentation, the usual argument in favor of compiled code instrumentation is its efficiency.

Beside Specht, Calejo [2] uses source instrumentation by program transformation to trace logic programs. No performance measurements are available for any of the systems.

Tolmach and Appel [14] designed and implemented a tracer for Standard ML based on automatic program source instrumentation. The resulting code runs only 3 times slower than optimized code. They conjectured that a low-level tracer would run at about the same speed. However they had no reasonable low-level tracer at their disposal with which they could compare their results.

We have performed such a comparison in the context of Prolog, using two quite different systems: ECLiPSe [8, 5], and SICStus [12]. As far as source instrumentation is concerned, it is based on an extension of O'Keefe's `advice` utility [1]. The instrumentation takes built-in predicates into account and provides the user with a more complete trace model.

We have executed classical Prolog benchmark programs [10] collecting trace information without displaying it. On average, collecting trace information by program instrumentation is only about twice slower than using a low-level tracer when using ECLiPSe and has similar performance when using SICStus. Even a slowdown of two is a minor penalty to pay, compared to the advantages of the approach. Actually, such a slowdown is, in many cases, negligible, either because only part of the execution is traced, or because the execution is dominated by input/output (be it as part of the application or as part of displaying trace information).

To our knowledge, this is the first time that a quantitative comparison of both approaches is made for any programming language.

In the following we first introduce the trace model used by both tracers. We then informally present the instrumentation by program transformation. The transformation can deal with predicates which are *traced* or *skipped*. The difference is that the execution of the subgoals of skipped predicates is, unlike the execution of the subgoals of traced predicates, not traced. Some counters which help to analyze the trace are then introduced. Finally, we present our experiments. The methodology is described followed by some of the results and a discussion.

2 Modeling Prolog executions

A Prolog trace is a sequence of events which gives a picture of a program execution. We use an execution model close to the classical box model of Byrd [1] in which execution events are bound to goals. There are different types of events, called *ports*. A *call* event tells that g is invoked and gives the instantiation of its arguments at the moment of the invocation. A $fail$ event tells that g fails. An *exit* event tells that g succeeds and gives the resulting instantiation of the arguments. A *redo* event tells that the execution is backtracking either to g or to one of its subgoals. We have added the $unify$ event [6] which tells when the execution finds a clause that unifies with g and gives the resulting instantiation of the arguments, it also gives the unified clause.

[1] The `advice` utility is part of the DEC10 Prolog library, available by anonymous ftp from the AIAI of the University of Edinburg (aiai.edinburgh.ac.uk).

mem(X, List, Y, Z) ← *Y and Z are consecutive elements of List such that their sum is equal to X.*

```
mem(X,[Y,Z|Zs],Y,Z) :-
   X is Y + Z.
mem(X,[_|Zs],Y,Z) :-
   mem(X,Zs,Y,Z).
```

The predicate `mem/4` *is transformed to produce trace information.*

```
mem(X,Xs,Y,Z) :-
   ( trace(call,mem(X,Xs,Y,Z))
   ; trace(fail,mem(X,Xs,Y,Z)),
     fail),
   mem$proc(X,Xs,Y,Z),
   ( trace(exit,mem(X,Xs,Y,Z))
   ; trace(redo,mem(X,Xs,Y,Z)),
     fail).

mem$proc(X,[Y,Z|Zs],Y,Z) :-
   trace(unify,mem(X,[Y,Z|Zs],Y,Z),1),
   X is Y + Z.
mem$proc(X,[_|Zs],Y,Z) :-
   trace(unify,mem(X,[_|Zs],Y,Z),2),
   mem(X,Zs,Y,Z).
```

Figure 1: A Prolog program and an instrumented version of it.

In an exhaustive trace, the events related to a given goal are not necessarily consecutive. They are intertwined with events related to subgoals and siblings. A more detailed description of the trace model can be found in [3].

3 Informal presentation of the instrumentation

The first subsection presents a transformation which is basically the one of the `advice` package of the DEC10 Prolog library, extended in order to deal with the *unify* port. The next subsection extends the transformation to deal with *skipped* predicates. Specific transformations for the cut and meta-calls are presented. Useful counters are then introduced.

3.1 Instrumentation of a single predicate

Figure 1 shows a Prolog program and a transformed version of it, where the predicate `mem/4` is transformed to produce trace information. The body of `mem/4` is replaced by trace instructions and a new predicate `mem$proc/4` is created. This predicate calls the body of the original predicate `mem/4`.

The predicates `trace/2/3` can, for example, display their arguments on the standard output or they can send them to a trace analyser such as Opium [3]. The three important properties of the predicates `trace/2/3` are that they (1) succeed exactly once, (2) bind no variable of the original program and, (3) have no side effect on the original program.

With this scheme, the displayed values of the variables depend on the stage of the execution which is traced. The `call` trace gives the value of variables at invocation time; the `unify` trace gives the value of variables resulting from a unification; the `exit` trace gives the value of variables resulting from the execution of the goal; the `redo` trace gives the same values as the previous `exit` trace related to the same goal; the `fail` trace gives the same values as the `call` trace related to the same goal. Indeed, the bindings that could have been made during the execution are undone.

Except for the trace information sent to the standard output, the behavior of the transformed program is the same as the initial one; goals are evaluated in the same order and the same substi-

```
1       :- mem(5, [2,3,2,4],Y,Z).
2    call    mem(5,[2,3,2,4],Y,Z)
3    unify(1) mem(5,[2,3,2,4],2,3)
4    exit    mem(5,[2,3,2,4],2,3)
5       Y = 2
6       Z = 3      More? (;)
7    redo    mem(5,[2,3,2,4],2,3)
8    unify(2) mem(5,[_,3,2,4],Y,Z)
9    call    mem(5,[3,2,4],Y,Z)
10   unify(1) mem(5,[3,2,4],3,2)
11   exit    mem(5,[3,2,4],3,2)
12   exit    mem(5,[2,3,2,4],3,2)
13      Y = 3
14      Z = 2      More? (;)
15   redo    mem(5,[2,3,2,4],3,2)
16   redo    mem(5,[3,2,4],3,2)
17   unify(2) mem(5,[_,2,4],Y,Z)
18   call    mem(5,[2,4],Y,Z)
19   unify(1) mem(5,[2,4],2,4)
20   unify(2) mem(5,[_,4],Y,Z)
21   call    mem(5,[4],Y,Z)
22   unify(2) mem(5,[_],Y,Z)
23   call    mem(5,[],Y,Z)
24   fail    mem(5,[],Y,Z)
25   fail    mem(5,[4],Y,Z)
26   fail    mem(5,[2,4],Y,Z)
27   fail    mem(5,[3,2,4],Y,Z)
28   fail    mem(5,[2,3,2,4],Y,Z)
29      no (more) solution.
```

Figure 2: Trace obtained by the resolution of goal `mem(5,[2,3,2,4],Y,Z)` with the instrumented program of Figure 1

tutions are performed in the same order. The operational semantics of Prolog is preserved.

The resolution by Prolog of the goal `mem(5,[2,3,2,4],Y,Z)` of the instrumented program generates the trace of Figure 2. Line numbers have here been added by hand.

3.2 *Skipped* predicates

Figure 2 shows a trace related to one predicate, `mem/4`. In order to see information related to other predicates one can transform them, if their definition is available. This is unfortunately not the case for built-in predicates. Indeed, their source code is usually not available and, moreover, they are often not even implemented in Prolog.

Yet, a detailed trace of built-in predicates is not desirable. They are normally not under scrutiny while debugging user programs. What is interesting is to trace the fact that they are called and which result they return. The same applies to user-defined predicates which have been tuned and are assumed correct. We call these predicates *skipped*. Built-in predicates are handled in the same way as skipped predicates.

In the following, the predicates whose definitions are transformed are called *traced* to distinguish them from skipped (and built-in) predicates.

We therefore introduce a special transformation for skipped predicates. The principle is to use the same wrapper as for traced predicates, but instead of wrapping the definition, the transformation wraps the invocation.

Figure 3 shows the modifications of the instrumentation for the skipped predicate `is/2`, and the resulting differences on the trace. The invocation of `X is Y+Z` is replaced by the invocation of the wrapped version namely `is$trace(X, Y+Z)`[2]. The wrapped invocation is constructed only once.

Figure 3 also shows some resulting modifications of the trace. For example, one can see that the first clause of `mem/4` does not produce any solution because `5 is 4+2` fails.

[2]In this case the redo is useless, as `is/2` is known not to be resatisfiable.

A clause calling a skipped predicate

```
mem$proc(X,[Y,Z|Zs],Y,Z) :-
   trace(unify,mem(X,[Y,Z|Zs],
                  Y,Z),1),
   X is Y + Z.
```

is transformed into

```
mem$proc(X, [Y, Z|Zs],Y,Z) :-
   trace(unify, mem(X,[Y,Z|Zs],
                   Y,Z),1),
   is$trace(X,Y + Z).
```

```
is$trace(X, Expr) :-
   ( trace(call,is(X, Expr))
   ; trace(fail,is(X, Expr)),
     fail),
   is(X, Expr),
   trace(exit,is(X, Expr)).
```

The portion of trace

```
19  unify(1) mem(5,[2,4],2,4)
20  unify(2) mem(5,[_,4],Y,Z)
```

becomes

```
19  unify(1) mem(5,[2,4],2,4)
    call    5 is 4+2
    fail    5 is 4+2
20  unify(2) mem(5,[_,4],Y,Z)
```

Figure 3: Modifications of the instrumentation for the skipped predicate `is/2`, and the resulting differences on the trace.

3.3 Cut, conditionals and meta-calls

The wrapping of an invocation of the built-in predicate `!/0` cannot be folded into a predicate `!$trace`, otherwise the scope of the cut operation would be incorrect. The wrapping is therefore done in place as illustrated by Figure 4. Note that, in the transformed program, the cut operation correctly cuts the choice points attached to the goals `q$proc` and `s`. It does not cut the choice points attached to the wrapping of `q`, nor the ones attached to the goal `t`. When the execution backtracks to `t`, the redo information of line 12 correctly informs that there is some backtracking inside the box of `q`. The same in-place wrapping is done for conditional constructs.

The goals which are arguments of meta-call predicates have also to be wrapped, especially if they are skipped. Moreover, most of the time their value is not known at transformation time, in that case the wrapping is postponed until execution time. This is illustrated by Figure 5. The predicate `wrap_goal/2` is called at execution time. Note that in the general case, with the counters introduced in the following, the predicate `wrap_goal/2` requires additional arguments. It is therefore slightly more complicated and costly.

3.4 The *call number* and *depth* counters

Considering lines 18-26 of Figure 2, one can see that they only deal with predicate `mem/4`. However, all the lines do not relate to the same goal. Indeed, line 18 relates to goal `mem(5,[2,4],Y,Z)` while line 23 relates to `mem(5,[],Y,Z)`. As it is necessary to be able to easily distinguish between different goals, some information about the invocation number must be introduced in a trace line/event, namely the *call number*. The *depth* of execution is also very useful when reconstructing search and proof trees. We therefore add another counter which represents the size of the invocation stack.

Figure 6 shows the transformation of `mem/4` in order to get the *call number* and *depth* counters. The predicates `mem$proc` and `is$trace` are also modified. A new predicate, `mem$trace/5`, is created. It initializes the `depth` to `1` while keeping the same external interface for predicate

Initial program

```
q :- s, !, t.
q .

s.

t.
t.
```

q is basically transformed into

```
q :-
    trace_call_fail(q),
    q$proc,
    trace_exit_redo(q).

q$proc :-
    trace_unify(q, 1),
    s,
    trace_call(!),
    !,
    trace_exit(!),
    t.
q$proc :-
    trace_unify(q, 2).
```

The corresponding trace

```
      :- q, fail.
1   call   q
2   unify  q
3     call   s
4     unify  s
5     exit   s
6     call   !
7     exit   !
8     call   t
9     unify  t
10    exit   t
11  exit   q
12  redo   q
13    redo   t
14    unify  t
15    exit   t
16  exit   q
17  redo   q
18    redo   t
19    fail   t
20  fail   q

no (more) solution.
```

Figure 4: Transformation of a predicate with a cut, and its corresponding trace.

Initial program

```
q(X) :- call(X).
```

q is basically transformed into

```
q(X) :-
    trace_call_fail(q(X)),
    q$proc(X),
    trace_exit_redo(q(X)).

q$proc(X) :-
    trace_unify(q(X), 1),
    wrap_goal(X, WrappedX),
    trace_call_fail(call(X)),
    call(WrappedX),
    trace_exit_redo(call(X)).
```

```
wrap_goal(Goal, WrappedGoal) :-
    (  traceable(Goal)
    -> WrappedGoal = Goal
    ;  WrappedGoal =
         (trace_call_fail(Goal),
          Goal,
          trace_exit_redo(Goal))
    ).
```

The corresponding trace

```
:-  q(true).

1  call  q(true)
2  unify  q(true)
3    call  call(true)
4      call  true
5      exit  true
6    exit  call(true)
7  exit  q(true)
```

Figure 5: Transformation of a predicate with a meta-call, and its corresponding trace.

`mem/4`. The predicate `incr_call/1` dynamically increments the global counter which handles the *call number*. *Depth* is incremented into `NewDepth`, which is passed to the subgoals by `mem$proc/7`. We introduce the `trace_call_fail/3` and `trace_exit_redo/3` predicates to ease the presentation.

Figure 6 also shows a portion of trace taking skipped predicates and counters into account. The first number in a line is the invocation number (*call number*), the second one, in brackets, is the *depth*. Note that the indentation has been calculated according to the *depth* information, starting with an indentation of 0 for depth 3.

4 Experimental Results

4.1 Methodology

In order to get a feel for the relative performance of program source instrumentation versus compiled code instrumentation, we have implemented, using ECLiPSe, the program transformations described above. We have also implemented, both in ECLiPSe and SICStus, the kernel of an interactive tracer, hereafter referred to as Poppy.

A trace can therefore be generated in two ways, through compiled code instrumentation or through source code instrumentation.

In order to use compiled code instrumentation, the program is compiled and then executed in debug mode (the interpretation of the instructions by the virtual machine produces trace information). Note that, when using ECLiPSe, the program must be compiled in a special debug compilation mode (specific ECLiPSe virtual machine instructions are produced). This makes it possible to mix, in debug mode, fast optimized code, and slower debuggable code. As far as SICS-

Transformation of `mem/4`, *a new predicate* `mem$trace/5` *is created*

```
mem(X,Xs,Y,Z) :-
   mem$trace(1,X,Xs,Y,Z).

mem$trace(Depth,X,Xs,Y,Z) :-
   incr_call(CallNb),
   trace_call_fail(mem(X, Xs, Y, Z),
                   Depth, CallNb),
   NewDepth is Depth + 1,
   mem$proc(Depth, CallNb, NewDepth,
            X,Xs,Y,Z),
   trace_exit_redo(mem(X, Xs, Y, Z),
                       Depth,CallNb).

mem$proc(Depth, CallNb, NewDepth,
         X, [Y,Z|Zs], Y, Z) :-
   trace(unify,
         mem(X, [Y,Z|Zs], Y, Z),
         Depth, CallNb, 1),
   is$trace(NewDepth,X,Y+Z).
mem$proc(Depth,CallNb, NewDepth,
          X, [_ |Zs], Y, Z) :-
   trace(unify,
         mem(X, [_ |Zs], Y, Z),
         Depth, CallNb, 2),
   mem$trace(NewDepth,X,Zs,Y,Z).
```

```
trace_call_fail(G,Depth,CallNb) :-
   trace(call,G,Depth,CallNb).
trace_call_fail(G,Depth,CallNb) :-
   trace(fail,G,Depth,CallNb),
   fail.

trace_exit_redo(G,Depth,CallNb) :-
   trace(exit,G,Depth,CallNb).
trace_exit_redo(G,Depth,CallNb) :-
   trace(redo,G,Depth,CallNb),
   fail.
```

The portion of trace equivalent to the lines 18-26 of Figure 2 with skipped predicates and counters is now

```
5[3] call     mem(5,[2,4],Y,Z)
5[3] unify(1) mem(5,[2,4],2,4)
6[4]   call 5 is 2 + 4
6[4]   fail 5 is 2 + 4
5[3] unify(2) mem(5,[_,4],Y,Z)
7[4]   call    mem(5,[4],Y,Z)
7[4]   unify(2) mem(5,[_],Y,Z)
8[5]      call    mem(5,[],Y,Z)
8[5]      fail    mem(5,[],Y,Z)
7[4]   fail    mem(5,[4],Y,Z)
5[3] fail     mem(5,[2,4],Y,Z)
```

Figure 6: The transformation of `mem/4` in order to get the *call number* and *Depth* counters. A portion of the trace with counters and skipped predicates.

tus is concerned, we have used SICStus version 3.7 beta [12]. This version makes it possible to trace compiled code (previous versions were relying on interpreted code for debugging) but debugging support is hidden to the compiler.

In order to use source code instrumentation, the program is first instrumented through program transformation. The instrumented program is then compiled and executed in optimized (non debug) mode. The trace is produced through calls to Poppy, *i.e.*, basically, calls to the predicates `trace_call/3`, `trace_call_fail/3`, `trace_exit/3`, `trace_exit_redo/3`, and `incr_call/1`. The predicate `trace_call/3` is a specialization of `trace_call_fail/3`, used for built-in predicates which are known not to fail (*e.g.* `write/1`) and `trace_exit/3` a specialization of `trace_exit_redo/3` used for built-in predicates which are known not to be resatisfiable (*e.g.* `is/2`).

In order to capture the cost associated to managing several tracing modes, as well as spied predicates, a subset of the usual tracing facilities has been implemented: step by step mode, *leap* mode (no trace is displayed until a spied predicate is encountered), *skip* mode (no trace is displayed until a trace line referring to the same call is encountered), and trace disabling.

The trace generated by ECLiPSe, SICStus, and Poppy have been made as close as possible. The leashing modes[3] of ECLiPSe have been set so that *next* events, specific to ECLiPSe, are not traced but so that *unify* events, not traced by default, are traced. As both ECLiPSe and SICStus do not explicitly trace disjunctions and conditional constructs, producing specific tracing predicates for these constructs has been disabled from our transformations. Tracing usual sequences consisting of a type checking predicate followed by a cut, used for indexing purposes, has also been disabled. Indeed, such sequences are not shown by ECLiPSe nor by SICStus (tracing them would mean deteriorating indexing).

However, discrepancies between both traces remain. Sometimes ECLiPSe and SICStus do less work. SICStus does not deal with *unify* ports and ECLiPSe does not show the results of successful unification of a fact. Neither ECLiPSe nor SICStus show pseudo backtracking (as *redo* events) into deterministic subtrees of the execution tree. Sometimes, they do more. Having direct access to the run-time data structures, both ECLiPSe and SICStus have the possibility of displaying information such as the determinacy of an exit. Moreover, when tracing a cut, all the goals made deterministic are shown by ECLiPSe.

All our measures have been made in leap mode, without any spy point set. This means that the trace information is produced but not displayed. Indeed the time spent displaying the trace would dominate the time spent in the tracing machinery.

Our comparisons are based on the Aquarius benchmark suite. This is a set of benchmark programs put together, from a number of different sources, as part of the Aquarius project (University of California, at Berkeley). This suite is well known in the Prolog community; it has, among others, been used to assess the performance of the BAM processor [7], the KCM machine [9], the Aquarius Prolog Compiler [10], and Parma [13], a Prolog compiler for the MIPS RISC architecture. It is also interesting in that it includes programs of different sizes and programming styles and does not discard the use of important built-in predicates such as `arg/3`, `functor/3`, or `write/1`, which often represent an important share of the execution time. Note that new versions of three programs, **flatten**, **reducer** and **sdda**, without calls to `write/1`, have been added to the initial benchmarks to get a better feel for the weight of this predicate. They are distinguished by a `_nw` suffix.

The programs were executed on a lightly loaded SUN Ultra-1/SunOS 5.5.1 workstation, using ECLiPSe version 3.5.2 and SICStus 3.7 beta, with garbage collection on. When using SICStus, both the `compactcode` and the `fastcode` compilation mode have been measured. The first mode produces byte-coded abstract instructions whereas the second one produces native machine instructions.

The execution times (user cpu time including garbage collection time) have been obtained by incrementally compiling and running each program one after another in a repeat/fail loop such that each program runs about 20 seconds. This guarantees that the timing intervals are well above the clock accuracy and reduces cache effects.

4.2 Results

The main results of our experiments are shown in figure 7. This figure gives for each compilation environment, ECliPSe, SICStus in compact compilation mode, and SICStus in fast compilation mode, the ratios between the execution times for producing a program trace with the native tracer and with Poppy, using the methodology previously described. A ratio greater than one means that Poppy is slower than the corresponding native tracer.

The corresponding average ratios are 1.18 (SICStus fast), 1.26 (SICStus compact), and 2.18

[3]The *leashing mode* of a port tells whether this port should be included in the trace and lead to ask the user for a new command.

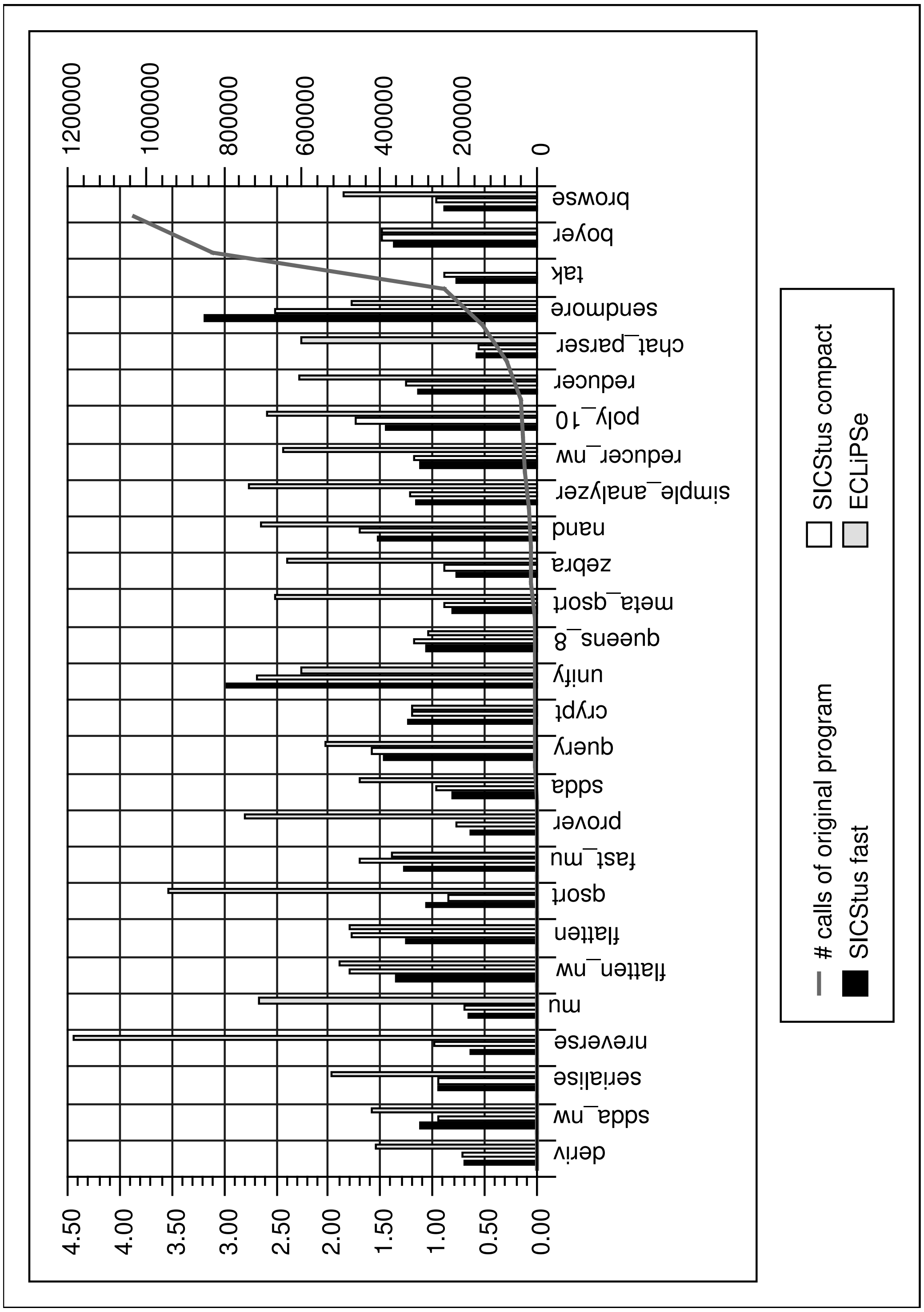

Figure 7: Source instrumentation vs code instrumentation ratios.

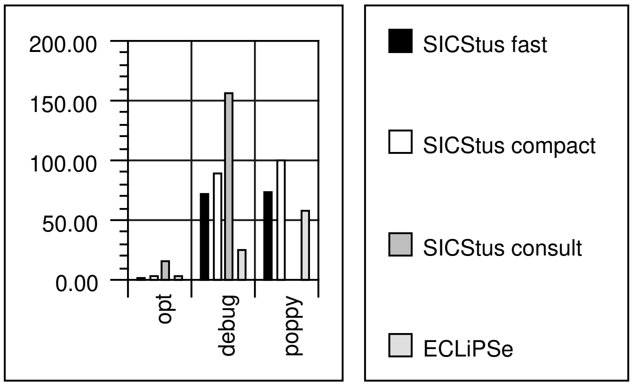

Figure 8: Relative speed depending on compilation mode and environment.

(ECLiPSe) with standard deviations of 0.62, 0.53, and 0.73, respectively. These results show that source instrumentation is a viable alternative to compiled code instrumentation.

Actually, it may be in some extreme cases, the only alternative. This is illustrated by the program `tak`. Using the standard configuration of ECLiPSe, running this program in debug mode, using the tracer of ECLiPSe and its ad-hoc data structures, results on a heap overflow, while Poppy handles the tracing without any problem. This is the reason why there is a bar missing for `tak`.

Looking at the results obtained with ECLiPSe, another bad result is obtained for `nreverse`. The result was somehow predictable; naive reverse is a totally deterministic program with no failure and no built-in predicate. The transformation introduces two choice points per invocation (one via `trace_call_fail/3` and one via `trace_exit_redo/3`). Assuming the transformation could tell that the program was deterministic, through program analysis, or user annotation, these choice points could be eliminated by replacing calls to `trace_call_fail/3` by calls to `trace_call/3`, and calls to `trace_exit_redo/3` by calls to `trace_exit/3`. Running the corresponding program almost halves the ratio. Looking at the results obtained with SICStus, however, `nreverse` behaves very well. Analyzing theses discrepancies requires further work.

Figure 8 gives relative average speedups, using SICStus in fast compilation mode as a reference. This figure shows the very significant slowdown resulting from setting debugging on[4]. As for source instrumentation, the slowdown is not surprising when looking at the static and dynamic effects of instrumentation. Indeed, the compiled instrumented programs are on average 3.06 (SICStus fast), 3.75 (SICStus compact), and 3.56 (ECLiPSe) times bigger than the compiled initial programs. As far as the execution is concerned, the ECLiPSe profiler shows that, on average, the number of goal invocations is multiplied by 17.6 and the number of choice points by 11.6. Nevertheless, debugging compiled code is faster than debugging interpreted code.

[4]This advocates for making it possible to mix code compiled in optimized mode and code compiled and debug mode.

The figure additionally shows that the debug mode of ECLiPSe is quite effective. The relative slowness of Poppy on SICStus comes, at least partly, from the slowness of the blackboard (see [12]), compared to the global variables of ECLiPSe. Indeed, the blackboard in SICStus and global variables in ECLiPSe are used to implement the call number, whose handling has, according to preliminary profiling, a significant weight.

When tracing in a standard way, *i.e.* mainly on a step by step basis, so much time is taken by the display of the data that the time taken by tracing through instrumentation, even if twice longer than tracing at the abstract machine level, should not be an issue. When looking at elapsed times vs cpu times, it is significant that, compared to their "no write" version, the elapsed time ratios of the three programs **flatten**, **reducer**, and **sdda** are quite good (on ECLiPSe, the ratios are around 1 or below).

When used together with a trace analyzer such as Opium [3], the context is different. The whole trace may have to be examined by the trace analyzer before the user sees any result. Thus the time taken by the tracing mechanism may be an issue. A solution may be, as previously mentioned, to resort to user annotation or better compiler technology to improve the transformation. Another interesting path could be to customize the instrumentation depending on the user request.

5 Conclusion

We have presented a program transformation making it possible to trace Prolog programs without resorting to a low-level tracer.

As far as efficiency is concerned, we have shown that a typical slowdown of 2 is observed, in the less favorable environment, when generating the trace of standard benchmark programs. Considering the advantages in terms of portability and versatility, this slowdown is quite acceptable. We do not expect it to be significant when dealing with "standard" tracing, dominated by input/output. This may be more of an issue when connecting tracing to a trace analyser such as Opium [3]. Indeed, large fractions of the execution may need to be analysed and therefore traced. We are currently extending our first prototype with a connection to Opium to check the issue.

If performance turns out to be an issue, it could be interesting to resort to program analysis to improve the transformation. Another possibility would be to customize the instrumentation according to the user requests.

We are currently working on the prototype, adding some more basic tracing facilities and making the implementation more portable. This would provide Prolog systems without proper tracing facilities with a standard tracer. This would also make it possible to assess the approach in different Prolog environments, comparing it with other trace extraction approaches, and looking at the influence of the compilation technology.

Acknowledgements Erwan Jahier and Sarah Mallet gave fruitful comments on earlier versions of this article.

References

[1] L. Byrd. Understanding the control flow of Prolog programs. In S.-A. Tärnlund, editor, *Logic Programming Workshop*, Debrecen, Hungary, 1980.

[2] M. Calejo. *A framework for declarative Prolog debugging.* PhD thesis, New University of Lisbon, Portugal, March 1992.

[3] M. Ducassé. Opium: An extendable trace analyser for prolog. *The Journal of Logic programming*, 1999. To appear in the special issue on Synthesis, Transformation and Analysis of Logic Programs, A. Bossi and Y. Deville (eds), Also Rapport Technique INRIA 3257.

[4] M. Ducassé and J. Noyé. Logic programming environments: Dynamic program analysis and debugging. *The Journal of Logic Programming*, 19/20:351–384, May/July 1994.

[5] ECRC, Munich, Germany. *ECLiPSe 3.5 - User Manual*, Dec. 1995.

[6] M. Eisenstadt. A powerful Prolog trace package. In *Proceedings of the 6th European Conference on Artificial Intelligence*. North-Holland, 1985, September 1984.

[7] B. Holmer, B. Sano, M. Carlton, P. Van Roy, R. Haygood, W. Bush, and A. Despain. Fast Prolog with an extended general purpose architecture. In *Proceedings of the* 17^{th} *Annual International Conference on Computer Architecture*. IEEE/ACM, IEEE Computer Society Press, May 1990.

[8] M. Meier, A. Aggoun, D. Chan, P. Dufresne, R. Enders, D. Henry de Villeneuve, A. Herold, P. Kay, B. Perez, E. van Rossum, and J. Schimpf. SEPIA - an extendible Prolog system. In *Proceedings of the IFIP '89*, 1989.

[9] J. Noyé. An overview of the Knowledge Crunching Machine. In M. Abdelguerfi and S. Lavington, editors, *Emerging Trends in Database and Knowledge-base Machines*. IEEE Computer Society Press, 1994.

[10] P. V. Roy and A. M. Despain. High-performance logic programming with the Aquarius Prolog compiler. *Computer*, 25(1):54–68, Jan. 1992.

[11] G. Specht. *Source-to-source Transformationen zur Erklärung des Programmverhaltens bei deduktiven Datenbanken*. PhD thesis, Technischen Universität München, Juni 1992. In German.

[12] Swedish Institute of Computer Science. *SICStus Prolog User's Manual*, march 1998. Release 3.7 beta.

[13] A. Taylor. High performance Prolog on a RISC. *New Generation Computing*, 9:221–232, 1991.

[14] A. Tolmach and A. Appel. A debugger for Standard ML. *Journal of Functional Programming*, 5(2):155–200, April 1995.