Tracing Prolog programs by source instrumentation is efficient enough

Mireille Ducassé\textsuperscript{a,}\textsuperscript{*}, Jacques Noyé\textsuperscript{b}

\textsuperscript{a} IRISA/INSA, Campus Universitaire de Beaulieu, F-35042 Rennes Cedex, France
\textsuperscript{b} École des Mines de Nantes, 4 rue Alfred Kastler, La Chantrerie, F-44307 Nantes Cedex 03, France

Received 26 October 1998; received in revised form 5 July 1999; accepted 7 July 1999

Abstract

Tracing by automatic program source instrumentation has major advantages over compiled code instrumentation: it is more portable from one Prolog system to another, it produces traces in terms of the original program, and it can be tailored to specific debugging needs. The main argument usually put forward in favor of compiled code instrumentation is its supposed efficiency. We have compared the performances 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. © 2000 Elsevier Science Inc. All rights reserved.

Keywords: Prolog debugging; Program instrumentation; Benchmarking

1. Introduction

There are five basic ways to trace executions:
1. manual program source instrumentation,
2. automatic program source instrumentation,
3. instrumentation of meta-interpreters,
4. compiled code instrumentation, and
5. operating system interrupts.
The first four are discussed in [4], an example of the fifth one is the Unix ptrace primitive.

*Corresponding author. Tel.: +33-2-99-286485; fax: +33-2-99-286458;
E-mail addresses: ducasse@irisa.fr (M. Ducassé), noye@emn.fr (J. Noyé).
Manual program source instrumentation (1) is tedious and error-prone, instrumentation of meta-interpreters (3) is notoriously too inefficient, and operating system interrupts (5) are too low-level. The aim of this article is to compare the efficiency of automatic program source instrumentation (2) and compiled code instrumentation (4). The latter is often referred to as “low-level tracing” in the following.

Source instrumentation has major advantages over compiled code instrumentation. Firstly, it is more portable than compiled code instrumentation. Indeed, it is independent of the low-level implementation details of any particular abstract machine/compiler. It is therefore easy to port from one Prolog system to another. As a matter of fact, for our measurements, we did port our debugger to two different Prolog systems at a negligible cost.

Secondly, the instrumented program is traced at the level of the original program. Specht [12] showed that this feature is particularly adapted to tracing deductive database 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.

Thirdly, source instrumentation can be tailored to specific debugging needs, and therefore be more accurate. Indeed, in general not the whole trace information is needed for a given application or a given debugging request. For example, assume that an interpreter for language $L$ is implemented in Prolog. Programs written in $L$ may have to be traced. One way to proceed is to produce the Prolog trace and then filter out irrelevant information. A more appropriate way can be to customize the program transformation such that only the relevant information is generated in the first place.

While source instrumentation has advantages over compiled code instrumentation, the main argument usually put forward in favor of compiled code instrumentation is its efficiency. Note, however, that the instrumented programs are compiled. They, therefore, can benefit from many of the available compiler optimizations.

Besides 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 [15] designed and implemented a tracer for Standard ML based on automatic program source instrumentation. The resulting code runs only three 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 ECLiPSe [8] and SICStus Prolog [13]. Although, these systems can be seen as relying on the same basic technology – they are both built around a (different) variant of the Warren Abstract Machine written in C – they differ in many respects. In particular, we will see that they implement a different kind of compiled code instrumentation. For both systems, we compared low-level tracing, realized by using the low-level tracer provided with the system, with tracing through source instrumentation on the same system. The source instrumentation is based on an extension of O’Keefe’s
advice utility.\textsuperscript{1} We extended the advice package to take built-in predicates into account, in particular the cut operator and meta-calls, and to provide the user with a richer trace model.

We have executed classical Prolog benchmark programs [11], collecting trace information without displaying it. On average, collecting trace information using program source instrumentation is no slower than using the low-level tracer of SICStus Prolog and only about twice as slow as using the low-level tracer of ECLiPSe.

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 systematic quantitative comparison of both approaches is made for any programming language. Pettersson [10] mentions that the performance of a small particular C program was comparable when instrumented and when run under a standard low-level debugger. However, his instrumentation takes place at compiler intermediate code level, and the standard debugger uses operating system interrupts. He therefore compares levels (4) and (5) whereas we compare levels (2) and (4). Furthermore, his measurements were made on a single program, whereas our study uses 27 programs of a classical benchmark suite.

In the following we first introduce the trace model used by the tracers we use for our measurements. We then informally present the instrumentation by program transformation. The transformation can deal with predicates which are fully traced or whose execution details are skipped. Specific transformations for the cut and meta-calls predicates are discussed. We show how the transformation deals with useful traditional counters. 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 a goal \( g \) 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.

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].

\textsuperscript{1} The advice utility is part of the DEC10 Prolog library, available by anonymous ftp from the AIAI of the University of Edinburgh (aiai.edinburgh.ac.uk).
3. Informal presentation of the instrumentation

Section 3.1 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. Section 3.2 extends the transformation to deal with predicates whose execution details are skipped (called skipped predicates). Specific transformations for the cut and meta-call predicates are presented. Useful counters, traditionally used in Prolog debuggers, are then introduced.

3.1. Instrumentation of a single predicate

Fig. 1 shows a Prolog program and a transformed version of it. The initial predicate mem/4 has been split into two predicates mem/4 and mem$proc/4. The new version of mem/4 simply calls mem$proc/4 and contains trace hooks for invocation (call), success (exit), failure (fail) and backtracking (redo). The predicate mem$proc/4 is the same as the initial mem/4 where a trace hook about unification (unify) has been inserted in each clause.

The predicates trace/2 and trace/3 can, for example, display their arguments on the standard output or they can send them to a trace analyzer such as Opium [3]. The three important properties of the trace predicates 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. Except for the trace information sent to the standard output, the behavior of the transformed program is thus the same as the initial one; goals are evaluated in the same order and the same substitutions are performed in the same order. The operational semantics of Prolog is preserved.

With this scheme, the displayed values of the goal arguments depend on the stage of the execution which is traced:

\[
\begin{align*}
\text{mem}(X, \text{List}, Y, Z) & \leftarrow \\
& \text{Y and Z are consecutive elements of List such that their sum is equal to } X. \\
\text{mem}(X,[Y,Z|Zs],Y,Z) & \leftarrow \\
& \text{mem}(X,Zs,Y,Z). \\
\text{The predicate mem/4 is transformed to produce trace information.}
\end{align*}
\]

\[
\begin{align*}
\text{mem}(X,\text{Xs},Y,Z) & \leftarrow \\
& (\text{trace}(\text{call},\text{mem}(X,\text{Xs},Y,Z))) \\
& (\text{trace}(\text{fail},\text{mem}(X,\text{Xs},Y,Z)), \\
& \text{fail}), \\
& \text{mem$proc}(X,\text{Xs},Y,Z), \\
& (\text{trace}(\text{exit},\text{mem}(X,\text{Xs},Y,Z))) \\
& (\text{trace}(\text{redo},\text{mem}(X,\text{Xs},Y,Z)), \\
& \text{fail}).
\end{align*}
\]

Fig. 1. A Prolog program and an instrumented version of it.
The resolution by Prolog of the goal \texttt{mem(5, [2,3,2,4], Y, Z)} of the instrumented program generates the trace of Fig. 2. Line numbers have here been added by hand.

### 3.2. Skipped predicates

Fig. 2 shows a trace related to one predicate, \texttt{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.

Fortunately, a detailed trace of built-in predicates is usually not desired. They are normally not under scrutiny while debugging user programs. What is interesting is to trace the fact that they are called and the result they return. The same applies to user-defined predicates which have been tuned and are assumed correct. We call these predicates \textit{skipped}. Built-in predicates are handled in the same way as skipped predicates. In the following, the predicates whose definitions are transformed are called \textit{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

```prolog
1 :- mem(5, [2,3,2,4], Y, Z).  15 redo mem(5, [2,3,2,4], Y, Z).  3, 2)
2 call mem(5, [2,3,2,4], Y, Z)  16 redo mem(5, [3,2,4], Y, Z).  2, 3)
3 unify(1)mem(5, [2,3,2,4], Y, Z)  17 unify(2)mem(5, [.,2,4], Y, Z).
4 exit mem(5, [2,3,2,4], Y, Z)  18 call mem([2,4], Y, Z).
5 Y = 2  19 unify(1)mem(5, [2,4], Y, Z).
6 Z = 3 More? (.)  20 unify(2)mem(5, [.,4], Y, Z).
7 redo mem(5, [2,3,2,4], Y, Z)  21 call mem([5,4], Y, Z).
8 unify(2)mem(5, [.,3,2,4], Y, Z)  22 unify(2)mem(5, [.,Y, Z).
9 call mem([5,3,2,4], Y, Z)  23 call mem([5,4], Y, Z).
10 unify(1)mem([5,3,2,4], Y, Z)  24 fail mem(5, [.,Y, Z).
11 exit mem([5,3,2,4], Y, Z)  25 fail mem([5,4], Y, Z).
12 exit mem([5,2,3,2,4], Y, Z)  26 fail mem([5,2,4], Y, Z).
13 Y = 3  27 fail mem([5,3,2,4], Y, Z).
14 Z = 2 More? (.)  28 fail mem([5,2,3,2,4], Y, Z).
```

Fig. 2. Trace obtained by the resolution of goal \texttt{mem(5, [2,3,2,4], Y, Z)} with the instrumented program of Fig. 1.
A clause calling a skipped predicate

\[
\text{mem$\$$proc}(X, [Y, Z|Zs], Y, Z) :-
\text{trace}(\text{unify}, \text{mem}(X, [Y, Z|Zs], Y, Z), 1),
\]
\[
X \text{ is } Y + Z.
\]
definition, the transformation wraps the invocation. Fig. 3 shows the modifications of the instrumentation for the skipped predicate \textit{is/2}, and the resulting differences on the trace. The invocation of \textit{X is Y+Z} is replaced by the invocation of the wrapped version, namely \textit{is$\$$trace(X, Y + Z)}.

is transformed into

\[
\text{mem$\$$proc}(X, [Y, Z|Zs], Y, Z) :-
\text{trace}(\text{unify}, \text{mem}(X, [Y, Z|Zs], Y, Z), 1),
\]
\[
is$\$$trace(X, Y + Z).
\]
The portion of \textit{trace}

\[
19 \quad \text{unify}(1) \quad \text{mem}(5, [2,4], 2,4)
\]
\[
20 \quad \text{unify}(2) \quad \text{mem}(5, [-,4], Y, Z)
\]

becomes

\[
19 \quad \text{unify}(1) \quad \text{mem}(5, [2,4], 2,4)
\]
\[
call \quad 5 \text{ is } 4+2
\]
\[
fail \quad 5 \text{ is } 4+2
\]
\[
20 \quad \text{unify}(2) \quad \text{mem}(5, [-,4], Y, Z)
\]

Fig. 3. Modifications of the instrumentation for the skipped predicate \textit{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 \textit{!/0} cannot be folded into a predicate \textit{!$\$$trace}, otherwise the scope of the cut operation would be incorrect. Indeed, the cut would not cut beyond the procedure \textit{!$\$$trace} and would, thus, have no effect on its calling goal. The wrapping is therefore done in place as illustrated by Fig. 4. Note that, in the transformed program, the cut operation correctly cuts the choice points attached to the goals \textit{Q$\$$proc} and \textit{s}. It does not cut the choice points attached to the wrapping of \textit{q} nor the ones attached to the goal \textit{t}. When the execution backtracks to \textit{t}, the redo information of line 12 correctly tells that there is some backtracking inside the box of \textit{q}.

The same in-place wrapping is done for conditional constructs. Indeed cuts occurring in the “then” or “else” part of a conditional construct cut through to the head of the clause. A wrapping would have the same incorrect effect as for the cut.

---

\footnote{In this case the redo is useless, as \textit{is/2} is known not to be resatisfiable.}
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 Fig. 5. The predicate `wrap_goal/2` is called at execution time.

We define the `trace_call_fail/3` and `trace_exit_redo/3` predicates to ease the presentation and shorten the transformed code. Note that in the general case, with the counters introduced in the following, the predicate `wrap_goal/2` requires additional arguments and is, therefore, slightly more complicated and costly.

### 3.4. The call number and depth counters

Considering lines 15–26 of Fig. 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, especially those involving the same predicate, some information about the invocation number must be introduced into 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 number of ancestors of a goal.
Fig. 6 shows the transformation of \texttt{mem/4} in order to get the \textit{call number} (\texttt{CallNb}) and \textit{depth} (\texttt{Depth}) counters. The predicates \texttt{mem$\_proc} and \texttt{is$\_trace} are also modified. A new predicate, \texttt{mem$\_trace/5}, is created. It initializes the \texttt{depth} to 1 while keeping the same external interface for predicate \texttt{mem/4}. The predicate \texttt{incr\_call/1} dynamically increments the global counter which handles the \textit{call number}. The \texttt{depth} is incremented into \texttt{NewDepth}, which is passed to the subgoals by \texttt{mem$\_proc/7}.

Fig. 6 also shows a portion of trace taking skipped predicates and counters into account. The first number in a line is the invocation number (\textit{call number}), the second one, in brackets, is the \textit{depth}. Note that the indentation has been calculated according to the \textit{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,
Transformation of \textit{mem/4}. A new predicate \texttt{mem$trace/5} is created:

\begin{verbatim}
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.
\end{verbatim}

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

\begin{verbatim}
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,[_,4],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)
\end{verbatim}

Fig. 6. Transformation of \textit{mem/4} (defined in Fig. 1) in order to get the call number and Depth counters; and a portion of the trace with counters and skipped predicates.

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.
Poppy simply consists of the definitions of all the predicates introduced so far: \texttt{trace/4}, \texttt{trace/5}, \texttt{trace\_call\_fail/3}, \texttt{trace\_exit\_redo/3}, \texttt{incr\_call/1}, as well as two new predicates, \texttt{trace\_call/3} and \texttt{trace\_exit/3}. The predicate \texttt{trace\_call/3} is a specialization of \texttt{trace\_call\_fail/3}, used for built-in predicates which are known not to fail (e.g. \texttt{write/1}) and \texttt{trace\_exit/3} a specialization of \texttt{trace\_exit\_redo/3} used for built-in predicates which are known not to be resatisfiable (e.g. \texttt{is/2}).

In order to capture the cost associated with managing several tracing modes, as well as spied predicates, a subset of the usual tracing modes, namely \texttt{step by step}, \texttt{leap}, \texttt{skip}, and \texttt{no trace} was implemented. Indeed, for a given event, the behaviour of the predicates \texttt{trace/4} and \texttt{trace/5} depends on the tracing mode as well as on whether the predicate is spied or not. In the \texttt{step by step} mode the trace line is displayed (and the user is prompted for the mode to be used next). In the \texttt{leap} mode, no trace line is displayed until a so-called spied predicate is encountered. Our prototype uses a very naive implementation of spied predicates: a dynamic predicate \texttt{is\_spied/1} records the functors of spied predicates. In the \texttt{skip} mode, no trace is displayed until a trace line referring to the same call is encountered. Finally, the \texttt{no trace} mode disables tracing until the execution returns to the top level. We rely on dynamic compilation to get an efficient implementation of the predicate \texttt{trace/4} depending on the tracing mode. The predicate does not test the tracing mode on each event but its definition is changed through dynamic compilation (or dynamic loading if available) each time the tracing mode is changed.

In the context of a given Prolog system, a trace can therefore be generated in two ways, through compiled code instrumentation, using the native tracer of the system, or through source code instrumentation, using Poppy.

In order to use compiled code instrumentation, the program is compiled and then executed in debug mode. Typically, a different interpretation of the abstract instructions produces trace information or, when compiling to native code, different parts of the runtime routines dealing, for instance, with control transfer are executed. Another interesting option, implemented in ECLiPSe, is to produce abstract instructions specific to debugging. It is then possible to mix, in debug mode, fast optimized code, and slower debuggable code. But this also means that, in order to be debuggable, a program has to be compiled in a special debug compilation mode. As for SICStus, we have used SICStus version 3.7 beta [13]. This version makes it possible to trace compiled code (previous versions were relying on source interpretation for debugging) but there is no special compilation mode. Switching between debug and no debug mode is done within the abstract machine. The compiler modes \texttt{compactcode} and \texttt{fastcode} make it possible to choose between the production of compact byte-coded abstract instructions or fast native machine instructions. Both kinds of programs can be traced. ECLiPSe and SICStus give therefore access to a good range of low-level tracing schemes.

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 (no debug) mode. The trace is produced through calls to Poppy.

The trace generated by the native debugger of ECLiPSe, by the native debugger of SICStus, and by Poppy have been made as close as possible. The.leashing
modes \(^3\) of ECLiPSe were 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 was disabled from our transformations. Tracing usual goal sequences consisting of a type checking goal followed by a cut, used for indexing purposes, was also 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 successfully unifying 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 were 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 [11], and Parma [14], 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 of “real” Prolog programs. 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.\(^4\) The execution times (user CPU time including garbage collection time) were obtained by incrementally compiling and running each program one after another in a repeat/fail loop such that each program runs at least 20 s. This guarantees that the timing intervals are well above the clock accuracy and reduces cache effects. Of course, the execution times correspond to the actual execution of the programs. They do not include any compilation or source transformation time. The time taken by the repeat/fail loop is also deduced by running and timing a dummy repeat/fail loop of the same length.

\(^3\) The leashing mode of a port tells whether this port should be included in the trace and should lead to ask the user for a new mode.

\(^4\) We also made measurements with garbage collection off, the figures turned out to be only marginally different.
4.2. Results

The main results of our experiments are shown in Fig. 7. This figure gives for each possible configuration, 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 on the same Prolog system, using the methodology previously described. A ratio greater than one means that Poppy is slower than the native tracer.

The corresponding average ratios are 1.18 (SICStus fast), 1.26 (SICStus compact), and 2.18 (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 the only alternative in some extreme cases. 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 extreme figure is obtained for nreverse, which is more than 4 times slower when traced through source instrumentation. The result was somehow predictable; naive reverse is a totally deterministic program with no failure and no built-in predicates. The transformation introduces two choice points per invocation (one via \texttt{trace\_call\_fail/3} and one via \texttt{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 \texttt{trace\_call\_fail/3} by calls to \texttt{trace\_call/3}, and calls to \texttt{trace\_exit\_redo/3} by calls to \texttt{trace\_exit/3}. Running the corresponding program almost halves the ratio. Looking at the results obtained with SICStus, however, \texttt{nreverse} behaves very well. Analyzing these discrepancies requires further work.

Fig. 7 compares different tracing schemes. But what are the costs of tracing compared to running optimized code? Fig. 8 gives, on the vertical axis, relative average speeds depending on the Prolog system used (SICStus or ECLiPSe), the compilation mode for SICStus (fast, compact, or consult – the latter mode corresponds to source interpretation) and the debug mode (no debugging, native debugging, source instrumentation). The fastest execution configuration, SICStus running native code with debugging off, is taken as the reference and, therefore, the value of SICStus fast in no debug mode is 1.

The figure shows the very significant slowdown resulting from setting debugging on, independently of the tracing scheme. As for source instrumentation, the slowdown is not surprising when looking at the static and dynamic effects of instrumentation. Indeed, the figures generated by the ECLiPSe and SICStus compilers during compilation showed that the compiled source-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

5 This also advocates for making it possible to mix code compiled in optimized mode with code compiled in debug mode.
Fig. 7. Source instrumentation and Poppy vs code instrumentation ratios.
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. But native tracing has also got its cost. It is indeed significantly more efficient than mere interpretation (about twice as efficient) but remains, on SICStus, around 80 times slower than optimized code. The performance of source instrumented code is similar. On ECLiPSe, source instrumented code is faster. This is due, at least partly, from the slowness of the SICStus blackboard (see [13]), 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. However, the efficiency of the native tracer of ECLiPSe explains that the ratio Poppy/ECLiPSe tracer is not as good as on SICStus.

The previous figures ignore the time taken by displaying the trace. When tracing in a standard way, 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 half as efficient as tracing at the abstract machine level, should not be an issue. When looking at elapsed times rather than CPU times, it is significant that, compared to their “no write” versions, 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 to resort to user annotation or better compiler technology based on program analysis (mode analysis, determinacy analysis, ...) to improve the transformation. The previous discussion on nreverse showed that this approach is indeed promising. 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 configuration, 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 analyzed and therefore traced. 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 (Poppy, program transformation), 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. We also thank the anonymous referees for their comments and suggestions.

References