Profiling Prolog Programs

Saumya K. Debray

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

Abstract: Profilers play an important role in the development of efficient programs. Profiling techniques developed for traditional languages are inadequate for logic programming languages, for a number of reasons: first, the flow of control in logic programming languages, involving backtracking and failure, is significantly more complex than in traditional languages; second, the time taken by a unification operation, the principal primitive operation of such languages, cannot be predicted statically because it depends on the size of the input; and finally, programs may change at runtime because clauses may be added or deleted using primitives like *assert* and *retract*. In this paper we describe a simple profiler for Prolog. The ideas outlined here may either be used to implement a simple interactive profiler, or be integrated into Prolog compilers.

1. Introduction

Profilers play an important role in the development of efficient programs. It is often not possible to anticipate, beforehand, what the ‘‘hot spots’’ of a program will be, i.e. which portions of it will account for much of its execution time. In such cases, it is usually simpler and more economical to code the program, obtain execution profiles for it on representative input data, and then recode the hot spots to make them more efficient. In contrast, a programmer who blindly tries to optimize the program without knowing which portions of the program to concentrate on may spend a great deal of time and effort optimizing code that is executed relatively infrequently, resulting in only marginal improvements to the performance of the program.

The implementation of profilers for traditional languages appears to be well understood, because of the relatively simple flow of control in such languages. Execution profilers for such languages often rely on sampling techniques [9, 11]; others maintain runtime counts that record the total number of times each program statement is executed ([12], see also [1]). With the increasing popularity of logic programming languages like Prolog, and the increasing volume of code being written in such languages, the need for profiling tools for such languages becomes important. However, in this case the situation is complicated by the complex flow of control, since procedures may be backtracked into and succeed a number of times, or may fail, causing execution to backtrack into other procedures. The techniques used in profiling traditional languages may not always be adequate for languages like Prolog, for several reasons: first, simple counts of how many times a procedure is executed may not provide enough information about the runtime behavior of a program to the programmer: in general, it will also be necessary to indicate how often a procedure is backtracked into and how often it fails. Similarly, while sampling techniques give an indication of the amount of time spent in different procedures, they are inadequate for monitoring the dynamic search behavior of programs, e.g. how often a procedure is called, how often it backtracks, how much time is spent in backtracking, etc.

In this paper, we describe a simple, interactive profiler developed for the SB-Prolog system [6]. Our aim is to develop a profiling tool that integrates well into an interactive programming environment, and which permits profiling to be turned on or off interactively during the program development phase without cumbersome recompilation or relinking of code. It should be pointed out, however, that an equally important objective is to provide a simple description of how profiling issues in the presence of backtracking and failure can be handled. Our solutions can therefore be readily adapted to other environments, e.g. where source-to-source profiling transformations are effected on the program before compilation, or where the compiler directly adds profiling code to the program when the appropriate compiler options are specified.

Related work includes a Prolog profiler described by Gorlick and Kesselman [8], and the profiler for Icon, a programming language that supports pattern-matching and backtracking [10]. The former uses instruction execution counts obtained at runtime, together with statistical estimates of the time spent in each instruction, to compute the time spent in each clause. In Icon, profiling is handled by surrounding each expression in the program with code that accumulates profiling information [Griswold personal communication].

Our work differs from that of Gorlick and Kesselman in several respects. First, our profiler is intended primarily for use in an interactive environment where recompilation and relinking of programs is to be avoided where possible. Second, while our profiler works at the level of predicates, that of Gorlick and Kesselman works at the level of clauses: this implies that their system, unlike ours, can give more accurate information regarding shallow backtracking (i.e., where backtracking occurs between clauses in the process of finding a solution; this is in contrast to deep backtracking, where execution backtracks into a predicate to find alternative solutions). However, our approach can be adapted without much difficulty to work at the clause level, either via source-to-source transformations of the source program or by modifying the compiler to emit the appropriate instructions for profiling. Since we do not rely on statistical analyses of programs to provide estimates of runtime costs of instructions, our approach is more robust in handling predicates whose execution behavior departs significantly from that of the ‘‘typical’’ program. Finally, we are able to handle predicates that are dynamic, i.e. whose clauses may be added to or deleted from at runtime via the Prolog primitives *assert* and *retract*, in a uniform manner without any problem.

The remainder of this paper is organized as follows: Section 2 is a brief summary of Prolog, and may be skipped by the reader familiar with the language. Section 3 considers some issues in profiling Prolog programs, and sketches high level solutions for these. Section 4 illustrates the utility of profilers for improving Prolog programs through an example. Section 5 discusses the implementation of a prototype profiler for the SB-Prolog system, and Section 6 concludes with a summary.

2. Preliminaries

This section briefly outlines the basics of Prolog, and may be skipped by the reader familiar with the language.

2.1. Prolog: Syntax

A Prolog symbol is a variable, a function symbol, or a predicate symbol. Data objects in Prolog are called *terms*, which are variables, constants, or a compound term of the form $f(t_1, ..., t_n)$, where f is an n-ary function symbol, i.e. taking n arguments, and the t_i, $1 \le i \le n$, are terms. A compound term $f(t_1, ..., t_n)$ is conceptually analogous to a Pascal record whose type is f and whose fields are $t_1, ..., t_n$.

The basic program entity in Prolog is the clause, which consists of two parts, the *head* and the *body*. The head of a clause is an atomic formula of the form $p(t_1, ..., t_n)$, where p is an n-ary predicate symbol and the t_i, $1 \le i \le n$, are terms. In this case, p is said to be the predicate symbol of the clause. The body of a clause is a *phrase*, which we define as follows: an atomic formula $p(t_1, ..., t_n)$ is a phrase; and if **P**, **Q** and **R** are phrases, then so are *not*(**P**), **P , Q** (read ‘‘**P** *and* **Q**’’), **P ; Q** (read ‘‘**P** *or* **Q**’’), and **P** $\rightarrow$ **Q ; R** (read ‘‘*if* **P** *then* **Q** *else* **R**’’). The operational behavior of these constructs is discussed later in this section. A clause with head *Hd* and body *Bdy* is written *Hd* :– *Bdy*, read ‘‘*Hd if Bdy*’’. If the body of a clause is empty, it is usually omitted, and the clause written simply as *Hd*. Following the convention of

Edinburgh Prolog (see [5]), variable names will be written beginning with upper case letters, while function and predicate symbol names will begin with lower case letters. In addition, we will adopt the following notation for lists: the empty list will be written as '[]', while a list with head H and tail L will be written '[H|L]'.

A Prolog program consists of a set of predicate definitions. A predicate definition consists of a sequence of clauses having the same predicate symbol. Conceptually, a clause corresponds to a procedure definition, where the head gives the formal parameters and the literals in the body correspond to the procedure calls defining the procedure body. A predicate definition corresponds to a procedure definition, each clause for the predicate corresponding to an alternative body for the procedure. Thus, a predicate definition in Prolog can be thought of as a simple generalization of procedure definitions in traditional languages, in that multiple alternative procedure bodies, not necessarily mutually exclusive, are permitted. An example of a predicate definition is given by the following clauses for quicksort:

```
qsort([],[]).
qsort([M|L],R) :-
   split(M,L,U1,U2), qsort(U1,V1), qsort(U2,V2), append(V1,[M|V2],R).
```

2.2. Prolog: Operational Semantics

While the meaning of Prolog programs is usually given declaratively in terms of the model theory of first order logic, such programs can also be understood procedurally. In this view, each predicate is a procedure defined by its clauses. Each clause provides an alternate definition of the procedure body. The terms in the head of the clause correspond to the formal parameters, and each literal in the body of the clause corresponds to a procedure call. Thus, the definition of the `qsort` predicate above could be understood as two alternative procedure bodies: one for the case where the list to be sorted is empty, and the other, for the case involving a nonempty list, consisting of a call to the procedure `split`, followed by two recursive calls to the procedure `qsort`, and finally a call to the procedure `append`.

Parameter passing in such procedure calls is via a generalized pattern matching procedure called *unification*. Briefly, two terms t_1 and t_2 are unifiable if there is some substitution of terms (called the *unifier*) for the variables occurring in t_1 and t_2 that make t_1 and t_2 identical. For example, the terms $f(X, g(X, Y))$ and $f(a, Z)$ are unifiable with the unifier $\{X \rightarrow a, Z \rightarrow g(a, Y)\}$. Usually the most general unifier, i.e. one that does not make any unnecessary substitutions, is used. The result of unifying two terms is the term produced by applying their most general unifier to them. If the terms under consideration are not unifiable, then unification is said to *fail*.

The execution of a Prolog program follows the textual order of clauses and literals. Execution begins with a query from the user, which is a sequence of literals processed from left to right. The processing of a literal proceeds as follows: the clauses for its predicate are tried, in order, until one is found whose head unifies with the literal. If there are any remaining clauses for that predicate whose heads might unify with that literal, a backtrack point is created to remember this. After this, the body of the clause is executed. If unification fails at any point, execution backtracks to the most recent backtrack point: any variables that were instantiated after the backtrack point had been created have their instantiations undone, and then the next clause is tried. This process continues recursively until either all the literals have been processed completely (in which case execution is said to have *succeeded*), or when no alternatives are left to try (in which case it is said to have *failed*).

The execution of the body of a clause proceeds as follows: a conjunction of the form **P , Q** is executed sequentially from left to right: **P** is executed completely first, then **Q**. In a disjunction **P ; Q**, **P** is executed first, and if its execution fails, **Q** is executed. The processing of the conditional **P** $\rightarrow$ **Q ; R** proceeds as follows: **P** is executed first; if this succeeds, then any remaining alternatives for **P**, together with **R**, are discarded, and then **Q** is executed; if the attempt to solve **P** fails, **R** is tried. The processing of a negation *not*(**P**) is as follows: if the execution of **P** succeeds then that of *not*(**P**) fails, while if the execution of **P** fails then that of *not*(**P**) succeeds.

3. Issues in Profiling Prolog Programs

This section addresses some of the problems encountered in dealing with backtracking and failure when profiling Prolog programs. For ease of exposition, we assume an environment where each call to be profiled is intercepted by the profiler. The profiler we describe works at the predicate level, so that while it can keep track of the deep backtracking behavior of predicates, it does not provide information about shallow backtracking. However, the approach described here can readily be adapted to systems where the profiling code is generated at compile time rather than dynamically at runtime, and the techniques outlined here can easily be extended to the clause level for such systems.

3.1. Counting Calls

Maintaining a count of the number of calls to a predicate is conceptually a simple matter: at each call, a counter for that predicate is incremented. In order to access the appropriate table entry, it is first necessary to extract the predicate name and arity from the call. This is done by a predicate `get_pred_name`. The predicate `$profile` that handles the profiling of a call therefore looks something like:

```
$profile(Call) :-
     get_pred_name(Call, Pred, Arity),
     update_count(Pred, Arity), ...
```

The predicate `update_count` updates the count of the predicate if it is being counted, and does nothing otherwise:

```
update_count(Pred, Arity) :-
     counting(Pred, Arity) →
          (get_old_count(Pred, Arity, OldCount),
           Newcount is Oldcount + 1,
           update_count_table(Pred, Arity, NewCount)
          ) ;
```

```
        true.           /* not being counted */
```

3.2. Timing Calls

3.2.1. A Naive Approach

The discussion above on call counting has not revealed anything new. When we concern ourselves with the timing of calls, however, some interesting problems arise, having to do with the management of timing in the presence of backtracking and failure. Much of the remainder of this section will be devoted to addressing these issues.

A naive approach to timing predicates would be similar to the counting of calls, giving something like:

```
$profile(Call) :-
     get_pred_name(Call, Pred, Arity),
     update_count(Pred, Arity),
     update_time(Pred, Arity, Call).

update_time(Pred, Arity, Call) :-
     timing(Pred, Arity) →
        (cputime(T0), call(Call), cputime(T1),
         get_pred_time(Pred, Arity, OldTime),
         NewTime is OldTime + T1 - T0,
         set_pred_time(Pred, Arity, NewTime)
        ) ;
        call(Call).        /* not being timed */
```

where `cputime` is a system predicate that, when called, unifies its argument with the time elapsed since the Prolog system began was started up.

The reader will immediately notice several problems with this, however. The first is that this does not work quite right for recursive predicates, since the time taken in recursive calls is considered more than once, giving inflated time usage figures for recursive predicates. A straightforward solution to this is to maintain, for each predicate, a bit that indicates whether or not a call for it is already being timed. At entry to a call, this bit is checked to see if the call should be timed: a value of **0** indicates that the call is not a recursive call for the predicate, and hence should be timed. The profiler then sets the bit to **1** and proceeds with timing, and at the exit from the call it updates the time usage figures and resets this bit to **0**. If any recursive calls were encountered during the execution of this call, this bit would have the value **1**, and the recursive calls would therefore not be timed. The modified `update_time` routine is

```
update_time(Pred, Arity, Call) :-
     (timing(Pred, Arity), get_reccall_bit(Pred, Arity, 0)) →
        (set_reccall_bit(Pred, Arity, 1)
         cputime(T0), call(Call), cputime(T1),
```

```
       exit_update(Pred, Arity, T0, T1)
         ) ;
         call(Call).        /* not being timed */
```

where `exit_update` is defined as

```
exit_update(Pred, Arity, T0, T1) :-
     set_reccall_bit(Pred, Arity, 0),
     get_pred_time(Pred, Arity, OldTime),
     NewTime is OldTime + T1 - T0,
     set_pred_time(Pred, Arity, NewTime).
```

Notice that in `update_time`, the call to `get_reccall_bit` unifies the value of the ‘‘recursive call’’ bit with **0**: for a recursive call, this bit will have the value **1**, unification will fail and the recursive call will therefore not be timed separately.

3.2.2. Handling Backtracking

While the solution above suffices to handle recursive calls, it still suffers from a major defect: it does not handle backtracking and failure correctly. We first consider the problem of backtracking. Consider the schematic of control flow illustrated in Figure 1. The call to be timed is entered at time t_1, and succeeds for the first time at time t_2. Execution fails back into it at time t_3, and succeeds upon backtracking at time t_4, and so on; eventually, execution backtracks into the call at time t_{k-1}, and succeeds for the last time at time t_k. The next time execution fails back into the call, at time t_{n-1}, it fails out of the call at time t_n.

The time T that should be charged to this call is the time spent ‘‘inside the box’’, i.e.

$$T = (t_2 - t_1) + (t_4 - t_3) + \ldots + (t_k - t_{k-1}) + (t_n - t_{n-1}).$$

However, observe the behavior of the `update_time` predicate defined above: the clock time t_1 is recorded just before entry into the call, and each time execution succeeds through the call, the clock time t_2 is noted at the successful exit, and the difference $t_2 - t_1$ is charged to the call. Over several successful exits from the call due to backtracking, the total amount of time charged to the call is

$$T' = (t_2 - t_1) + (t_4 - t_1) + \ldots + (t_k - t_1).$$

Thus, the amount of time charged to the call is in general more than just the amount of time spent ‘‘inside the box’’, thereby giving inaccurate results.

Clearly, what is necessary here is to keep track of the time at which execution last entered (from a call) or reentered (via backtracking) the box. This is done by maintaining a table that associates, with each predicate being timed, the time at which execution last entered or reentered the most recent call to that predicate that was timed (since recursive calls are not timed, we do not have to worry about them). Additional code is packaged around the call so that whenever execution backtracks into it, the time-last-entered table is updated appropriately. The modified code for `update_time` is now as follows:

t_1	call	–	–	success	t_2
			–	backtrack	t_3
			–	success	t_4
			–	backtrack	t_{k-1}
			–	success	t_k
t_n	fail	–	–	backtrack	t_{n-1}

Figure 1: Schematic of Control Flow in a call

```
update_time(Pred, Arity, Call) :-
     (timing(Pred, Arity), get_reccall_bit(Pred, Arity, 0)) →
        (entry_update(Pred, Arity),
         call(Call),
         ( cputime(T1) ; backtrack_update(Pred, Arity) ),
         exit_update(Pred, Arity, T1)
         ) ;
         call(Call).       /* not being timed */
```

where the utilities `entry_update`, `backtrack_update` and `exit_update` are defined as follows:

```
entry_update(Pred, Arity) :-
     set_reccall_bit(Pred, Arity, 1),
     cputime(T0),
     set_TimeLastEntered(Pred, Arity, T0).
```

```
backtrack_update(Pred, Arity) :-
     cputime(T1),
     set_TimeLastEntered(Pred, Arity, T1),
     fail.

exit_update(Pred, Arity, T1) :-
     set_reccall_bit(Pred, Arity, 0),
     get_pred_time(Pred, Arity, OldTime),
     get_TimeLastEntered(Pred, Arity, LastEntry),
     NewTime is OldTime + T1 - LastEntry,
     set_pred_time(Pred, Arity, NewTime).
```

Now, each time execution succeeds through `Call`, the variable `T1` is instantiated to the time at success. If execution fails back, however, it first backtracks into `backtrack_update`: this results in the time-last-entered table being updated with the clock value at the time of backtracking, after which execution fails back into `Call`. Since the time-last-entered table is also set at the time `Call` is first entered, the net result is that it always contains the proper value of the clock when execution last entered the call. This table is looked up by `exit_update`, and the value recorded in it used, when updating the table entry for the amount of time spent in the call.

3.2.3. Handling Failure

While the modification described above improves the situation to a great extent, it may still not give quite the right results for some predicates. There are two reasons for this. The first is that the time spent in the call between the time execution last backtracked into it (t_{n-1} in Figure 1) and the time execution failed out of it (t_n in Figure 1) is not being charged to the call. Thus, where we had earlier been overestimating the amount of time spent in the call, we are now underestimating it by the amount $t_{n-1} - t_n$. It is possible for this amount to be quite large, resulting in inaccurate results from the profiler. A more serious problem is that if the call that fails is a recursive one, then the bit indicating whether or not the call is a recursive one (and hence, whether or not it should be timed) is not reset, and hence future calls to the called predicate are never timed.

At this point, however, we have enough machinery in place to be able to handle this without much trouble. Notice that the time t_{n-1} when execution reentered the predicate for the last time will have been noted in the time-last-entered table. To handle failure properly, therefore, it is only necessary to get the time t_n on the failure exit and use this to appropriately update the amount of time used for the call. This can be done simply by providing an alternative to `Call` so that when `Call` finally fails, this execution path can be taken, and the appropriate tables and flags updated as necessary, before failing:

```
update_time(Pred, Arity, Call) :-
     (timing(Pred, Arity), get_reccall_bit(Pred, Arity, 0)) →
        (entry_update(Pred, Arity),
        ( call(Call),
          (cputime(T1) ;                    % success
           backtrack_update(Pred, Arity)   % backtrack
          ),
          exit_update(Pred, Arity, T1)
         ) ;
         failure_update(Pred, Arity)       % failure
        ) ;
        call(Call).                         % not being timed
```

where `failure_update` is defined as

```
failure_update(Pred, Arity) :-
     cputime(T),                        % time at failure
     exit_update(Pred, Arity, T),
     fail.
```

The predicates `entry_update`, `backtrack_update` and `exit_update` are as defined at the end of Section 3.2.2. Notice that since execution does not reenter the call in the failure case, it is not necessary to update the time-last-entered table in `failure_update`.

3.3. Counting Backtracks and Failures

The mechanism described so far allows the user to determine (*i*) the number of calls to any predicate during execution; and (*ii*) the total time spent in the execution of any predicate. This information alone, however, may not always be sufficient for identifying the real hot spots in a program. This is because the information so gathered says little or nothing about the backtracking and failure behavior of predicates. Two measures of interest when profiling a Prolog program are the number of times a predicate is backtracked into, and the number of times it fails. A large number of failures suggests that calls to the predicate may be exploring an overly large search space, which might be reduced, for example, by rearranging literals in its clause bodies. Alternatively, where feasible, intelligent or semi-intelligent backtracking algorithms could be invoked [2, 4]. If a predicate is backtracked into an inordinately large number of times, it suggests that other predicates ‘‘downstream’’ from it are failing a lot, and again the user has the alternatives of experimenting with literal orderings or invoking more intelligent backtracking strategies. Notice that in general, both kinds of information – how often a predicate is backtracked into, and how often it fails – can be useful. For example, consider a predicate `p` defined by the clause

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

and assume that `p` is called with its first argument bound and the second argument free. An execution profiling of the backtracking behavior showing that `q` failed a large number of times would suggest that a different literal ordering, e.g.

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

might be more efficient. On the other hand, given a predicate defined as

```
p(X, Y) :- q(X, Z), r(Z, Y).
p(X, Y) :- q(X, Z), s(Z, Y).
```

it might be the case that the total number of times `q` is backtracked into is greater than the number of failures for either `r` or `s` alone. This suggests that `q` may be generating a lot of ‘‘wrong’’ bindings, and that reexamining its code might be useful.

At this point, given the mechanism discussed in Section 3.2.3 for handling the timing of calls in the context of backtracking and failure, it is relatively straightforward to handle the counting of backtracking and failure as well. Two additional tables are maintained, one to count the number of times predicates are backtracked into, and one to count the number of times predicates fail. When execution backtracks into a call, the backtrack count of the appropriate predicate is incremented; when a call fails, the failure count of its predicate is incremented. Minor modifications are necessary to the code developed so far, because backtracking and failure have been handled within the predicate `update_time`, which does its table manipulation only if the called predicate is being timed. The predicates `entry_update`, `backtrack_update`, `exit_update` and `failure_update`, which do the actual table manipulation, also undergo minor modifications because the code for counting and timing has merged. At entry to the call, `entry_update` checks to see whether the call should be timed: if the predicate is not being timed, or if this is a recursive call that should not be timed, then it sets the variable *NoTime* to **1**; later routines such as `backtrack_update` look at the value of this variable to determine their timing action. The code for these predicates is given in the Appendix.

3.4. Interaction with Cuts

An unstated assumption throughout this paper has been that backtracking will be done in a strict LIFO manner, i.e., execution will always backtrack to the most recent backtrack point upon failure. This assumption is necessary if the maintenance of backtrack and failure counts, as well as the values of the recursive call bits, is to be handled properly. Prolog’s backtracking strategy usually follows this discipline, and the backtrack points are maintained in a stack (the *choice point stack*), so this assumption is usually not unjustified. Unfortunately, there is one situation where backtrack points may be removed from this stack even when they are not at the top of the stack, thereby upsetting the LIFO stack discipline. This has to do with Prolog’s *cut* construct.

While a full description of the behavior of the *cut* construct is beyond the scope of this paper (but see standard texts on Prolog, e.g. [5]), it is enough for our purposes to note that when a *cut* is executed, some backtrack points are removed from the top of the choice point stack. The number of backtrack points so deleted depends on the dynamic behavior of the program, and cannot, in general, be predicted statically. The problem we are faced with is that if any of the backtrack points so removed happens to be one set up to handle profiling, then the maintenance of backtracking and failure counts will not be handled properly, and the profiling information gathered will be incorrect.[1]

A straightforward way to fix this problem is to modify the implementation of *cut*. A common implementation technique for the *cut* is to ‘‘remember’’, at appropriate points, how far back on the choice point stack to cut back to (the *cut-to* point) and to trim the stack back to that point upon encountering a *cut* [3]. While this scheme is quite fast, it makes it difficult to determine whether or not "sensitive" backtrack points, such as those necessary for maintaining profiling information, are being discarded.

Our proposal to rectify this problem is as follows: first, a bit is added to each backtrack point. The setting of this bit indicates whether or not a backtrack point is a "sensitive" one. The system also maintains a global bit indicating whether or not the execution is being profiled. The implementation of *cut* is modified as follows: if the global profiling bit is set to **0**, i.e. no profiling is being done, then cut behaves exactly as before: the choice point stack is simply trimmed back to the appropriate *cut-to* point. However, if the profiling bit is set to **1**, the choice point stack is traversed, from its top down to the appropriate *cut-to* point, with only non-"sensitive" backtrack points being deleted; the top of the choice point stack is then set to point to either the topmost "sensitive" backtrack point, or the *cut-to* point for that *cut*, whichever is higher on the choice point stack. With this modification, profiling can be carried out in the presence of cuts without any problems.

While this modification does increase the overhead associated with *cut*, we feel that the increase is not unreasonably large. First, note that the additional bit required for each backtrack point need not result in a real increase in the space required for a backtrack point, since it can, in almost all cases, be absorbed into space used for maintaining other information necessary for backtracking. If no profiling is being done, then executing a *cut* now costs one extra test, that of the global profiling bit: this, we feel, is not intolerable. In any case, it is possible to have two forms of *cut*: one that tests the global profiling bit, and one that does not. Programs to be profiled have to be compiled with a switch that turns off certain optimizations, and this same switch could be engineered to translate *cuts* to the appropriate form depending on whether or not profiling was desired. With this, programs that are not being profiled do not have to pay the overhead of testing the global profiling bit in *cuts*.

4. An Example

This section shows how call, backtrack and failure counts obtained from the profiler discussed in the previous section can be used to successively transform a program to be more efficient. We use as our example a Prolog program for querying databases for an academic department, adapted from a similar example in [2]. The program finds students who are taking two courses from the same teacher, with both courses meeting in the same room:

---

[1] This problem was pointed out to us by David S. Warren. It is not unique to profiling, but is also encountered in the context of goal caching in Prolog [7].

```
student(john, cs453).            teacher(binkley, cs453).
student(john, cs520).            teacher(binkley, cs342).
student(john, cs455).            teacher(opus, cs455).
student(john, ma561).            teacher(dallas, cs520).
student(tom, cs342).             teacher(dallas, ma561).
student(tom, cs453).
student(mary, cs455).            course(cs453, eco103, tue).
student(mary, cs520).            course(cs455, gs701, mon).
student(paul, cs520).            course(cs455, gs701, wed).
student(jane, cs453).            course(cs342, eco103, fri).
student(jane, ma561).            course(cs520, gs703, tue).
student(robert, cs342).          course(ma561, ma123, mon).
student(larry, ma561).
student(larry, cs342).
student(larry, cs455).
```

We begin with a rather simple specification of the problem:

```
prog(L) :– findall( (S, T, R), p(S, T, R), L).
p(S, T, R) :-
     student(S, C1), student(S, C2),
     teacher(T, C1), teacher(T, C2),
     course(C1, R, D1), course(C2, R, D2),
     not(C1 = C2).
```

The predicate `findall(T, P, L)` finds all instances of the term `T` for which `P` is satisfiable, and collects them in the list `L`. Thus, a call `prog(L)` succeeds in binding `L` to a list of triples (*s*, *t*, *r*) where student *s* takes two different courses, both taught by *t* and both meeting in room *r*. When executed with count points set on the predicates `teacher`, `student` and `course`, a call

```
| ?- prog(X).
```

takes 1170 mS to execute,[2] and we find the following execution profile:

[2] All execution times were obtained using the SB-Prolog system [6] on a Sun-3/75.

pred	*calls*	*backtracks*	*failures*
`teacher`	78	58	78
`student`	16	54	16
`course`	41	48	41

As we can see, the predicate `teacher` is being called a disproportionately large number of times. This suggests that reordering the literals in the clause defining `p` might be useful. Since we are concerned with reducing the number of calls to, and failures from, the predicate `teacher`, the literals for this predicate are pulled forward, yielding the definition

```
p(S, T, R) :-
     teacher(T, C1), teacher(T, C2),
     student(S, C1), student(S, C2),
     course(C1, R, D1), course(C2, R, D2),
     not(C1 = C2).
```

The query now takes 738 mS to execute, an improvement of about 37%. The execution profile for the transformed program is

pred	*calls*	*backtracks*	*failures*
`teacher`	6	14	6
`student`	36	46	36
`course`	41	48	41

The transformation has thus succeeded in reducing the number of calls to `teacher`. However, one undesirable effect of the transformation is that the number of calls to the predicate `student` has more than doubled. What this suggests is that the literals

. . ., `teacher(T, C1), teacher(T, C2)`, . . .

may be defining too large a search space, thereby generating ‘‘useless’’ bindings for the variables `C1` and `C2`. A problem that reveals itself on closer inspection is that while the program requires the variables `C1` and `C2` to have different bindings, this is not checked until the very end of the clause for `p`. This results in the `student` and `course` relations being explored with incompatible values of `C1` and `C2`, leading to wasted computation and loss of efficiency. We next transform the program to check the distinctness of the bindings of `C1` and `C2` as soon as they have been generated:

```
p(S, T, R) :-
     teacher(T, C1), teacher(T, C2),
     not(C1 = C2),
     student(S, C1), student(S, C2),
     course(C1, R, D1), course(C2, R, D2).
```

With this program the query takes 264 mS to execute – a speedup by a factor of almost 3 – and the execution profile now is

pred	*calls*	*backtracks*	*failures*
`teacher`	6	14	6
`student`	16	16	16
`course`	8	6	8

Notice that the number of calls, backtracks into and failures has decreased significantly for both `student` and `course`. However, notice that `student` still appears to be called a disproportionately large number of times, suggesting that there may still be room for improvement in the order of literals in the clause. We next observe that the number of students will typically be larger than the number of courses being offered, so that the size of the *student* relation can be expected to be significantly larger than the size of *course*. This suggests that the *course* relation should be explored before the *student* relation, yielding the clause

```
p(S, T, R) :-
     teacher(T, C1), teacher(T, C2),
     not(C1 = C2),
     course(C1, R, D1), course(C2, R, D2),
     student(S, C1), student(S, C2).
```

This transformation achieves a further improvement in speed of 32%, and the query now executes in 200 mS. The execution profile now is

pred	*calls*	*backtracks*	*failures*
`teacher`	6	14	6
`student`	8	8	8
`course`	8	6	8

The overall improvement in execution speed is therefore seen to be from 1170 mS to 200 mS, a speedup of a factor of almost 6. However, these execution times were obtained with profiling on, and are somewhat inflated because of the overhead associated with maintaining profiling tables, and also because certain compiler optimizations had to be turned off so that the profiler could intercept all calls. A more accurate comparison, done without any profiling and without turning off compiler optimizations, indicates that the final program is about 7.5 times faster than the original one.

5. A Prototype Implementation

This section discusses some aspects of a prototype interactive Prolog profiler implemented for the SB-Prolog system [6] along the lines discussed in Section 3.

The profiler we describe is itself written in Prolog, the code being almost identical to that described in Section 3. A secondary problem that has to be addressed in this context is that of managing table entries efficiently and preserving their values over backtracking, since Prolog does not provide facilities for destructive assignment. Prolog systems typically use the primitives `assert` and `retract` to manipulate and preserve values over backtracking. For example, a programmer wishing to increment a counter and preserve the new value over backtracking might write something like

```
..., cnt(C0), C1 is C0+1, retract(cnt(C0)), assert(cnt(C1)), ...
```

However, `assert` and `retract` are general purpose routines that can incur significant runtime overhead. SB-Prolog provides a specialized primitive, `globalset`, that combines the effects of these primitives for a restricted class of situations. If `N` is an atomic value, and *cnt* a unary predicate defined by a single unit clause, then the effect of a call

```
. . ., globalset( cnt(N) ), . . .
```

is equivalent to that of

```
. . ., retract( cnt(_) ), assert( cnt(N) ), . . .
```

the difference being that `globalset` is significantly more efficient than the corresponding `retract`/`assert` combination. This primitive is often used in SB-Prolog for manipulating tables maintained as sets of unary predicates.

5.1. Interacting with the Profiler

In SB-Prolog, programs are typically executed within the command loop interpreter. Once the module containing the predicates to be profiled have been loaded into the system, the user may set ‘‘count points’’ on a set of predicates `L` by issuing the command

```
| ?- count(L).
```

Alternatively, or in addition, ‘‘time points’’ may be set using the command

```
| ?- time(L).
```

These commands may also be embedded in programs, thereby allowing the programmer to begin profiling at arbitrary points during execution. The profiler maintains counts of the number of calls to, backtracks into, and failures from predicates that have count points set on them; it also keeps track of the time spent in the subtrees below calls to those predicates that have time points set.

The statistics accumulated by the profiler may be displayed at any time by calling a predicate `prof_stats`. This predicate takes an optional argument that allows the user to specify whether or not the accumulated statistics should be reset after they have been displayed. In addition, the user can turn profiling on or off at any point using the predicates `profile` and `noprofile`: if profiling is turned off, none of the profiling actions are carried out, and the overhead associated with calling a predicate that has a count point or time point set on it reduces to checking a bit to determine whether profiling is on or not. The user can also selectively turn off profiling for any predicate that is being profiled: in this case, the entry point for the predicate in the runtime symbol table is reset to its original value, i.e. the intermediate call to the predicate that handles profiling is short circuited, so that no further profiling overhead is incurred for calls to this predicate.

Because of the way in which our system handles profiling, it is robust with respect to predicates whose runtime behavior does not conform to that of the ‘‘typical’’ Prolog program, and predicates that are dynamic, i.e. which can have clauses added to or deleted from its definition at runtime via `assert` and `retract`. One weakness that it has is that its timing statistics may be affected if there is a lot of garbage collection during execution; however, this may be dealt with by running the program more than once and averaging the results from the different runs.

5.2. Intercepting Calls

In order to profile a program, it is necessary to be able to intercept calls to the predicates being profiled. Under ordinary circumstances, this may not always be possible for compiled code, because the SB-Prolog compiler tries to optimize subroutine linkages wherever possible, replacing `call` and `execute` instructions (which access the symbol table to retrieve a procedure's entry point) by direct branches when the address of the called procedure is known at compile time. It is therefore necessary to compile the program with an option that turns off this optimization.[3] However, if the predicates to be profiled are being interpreted via *consult*, this optimization is not carried out and call interception is not a problem.

The actual work of profiling different predicates is handled by a Prolog predicate `$profile`. The call interception mechanism is best illustrated through an example. Assuming that calls are routed through the runtime symbol table, setting up the interception of calls proceeds as follows: assume that an *n*-ary predicate `foo` is to be profiled. A new predicate `prof$foo`, also of arity *n*, is generated, and its entry point is made the same as that of `foo`. At this point, therefore, both `foo` and `prof$foo` refer to the same executable code. Next, a new clause *C* is generated:

(*C*) $\quad \texttt{foo}(X_1, \ldots, X_n)\ \texttt{:-}\ \texttt{\$profile(prof\$foo}(X_1, \ldots, X_n)\texttt{)}.$

The final step in setting up the interception of calls to `foo` is to change its entry point in the symbol table to be that of the clause *C*. Thus, any call to `foo` is directed, via *C*, to the system predicate

[3] This problem is not unique to profiling, but is also encountered in other contexts that require call interception, e.g. in the tracing and goal caching tools supported by SB-Prolog [7].

$profile, which carries out the appropriate bookkeeping activities and eventually calls its argument, prof$foo, with the same arguments as in the original call to foo. Since the entry point of prof$foo is the original entry point for foo, this results in the code for foo being executed. Once execution succeeds through the original code for foo, i.e. on return from prof$foo, control is returned to $profile, which can take such further action as necessary. It is worth noting that a very similar mechanism is used in SB-Prolog's tracing and goal cacheing packages for call interception. Notice also that if clauses are added to foo using assert, or deleted using retract, the entry point is unaffected, which allows the profiling system to handle such dynamic predicates without any trouble.

5.3. Table Management

Statistics about the runtime behavior of predicates are maintained in tables. For improved speed of access, each table is implemented as a set of Prolog facts, with a single unary fact corresponding to each entry. The predicate name for the table entry is derived from the name of the source predicate it corresponds to and the name of the table. Thus, for example, the call count for a source predicate foo is maintained by a table entry given by a unary predicate '_$count$foo' defined by a single unit clause.[4] Given a call to the predicate foo, therefore, the corresponding entry in the call count table, '_$count$foo'(N), can be located quickly by hashing on the predicate name. The table entries are manipulated by the globalset primitive mentioned earlier. Actually a little bit of name manipulation is necessary to do this, because the call passed to the profiler is not the original call foo(t_1, ..., t_n), but rather the call prof$foo($t_1$, ..., t_n). It is not difficult to extract the name of the original predicate foo from this, and thence construct the predicate name for the appropriate table. This is done by the predicate get_pred_name:[5]

```
get_pred_name(Call, Pred, Arity) :-
        functor(Call, P0, Arity),
        name(P0, P0Name),               % prof$
        append("prof$", PredName, P0Name),
        name(Pred, PredName).
```

However, this involves the construction and manipulation of ''name lists'', which is repeated at each call and can end up being quite expensive. A simpler and more efficient solution is to construct a relation that associates the name of each predicate being profiled with that of its ''prof$-'' relative. Tuples are added to this relation at the same time as setting up the code for call interception for predicates being

---

[4] The '_$' prefix distinguishes it as a predicate internal to the system; this is an attempt to minimize the chance of name conflicts with user-defined predicates.

[5] It is easy to see that the call to append can be optimized away; however, we retain it here for ease of understanding.

profiled, so that when processing the call, it is enough to simply look up this relation (in systems like SB-Prolog supporting `assert` with argument indexing, the lookup can be made quite fast).

This way of managing these tables is dictated, in part, by the fact that in an interactive system, the predicates to be profiled are not generally known in advance by the system, so that the tables have to be created dynamically at runtime. While the cost of creating such table entries is not insignificant, it is done only once, when the user specifies that a predicate is to be profiled. Subsequent accesses to the table entries, through a hash table, and updating through the `globalset` primitive, can both be made fairly efficient. Since we expect these tables to be accessed and updated much more frequently than they are created, such an approach appears reasonable for this application.

5.4. Reducing the Runtime Overhead

The prototype profiler described typically incurs a runtime overhead of a factor of 10 to 20, which is about the same as for the Icon profiler [Griswold personal communication]. This does not affect the counts of calls, failures etc., but it can affect call timings (though this effect is less for larger programs). Our experience indicates, however, that the relative times, i.e. fraction of the total execution time spent in a predicate, is affected to a much smaller extent. Thus, hot spots can still be reliably identified.

Despite this, a runtime overhead of a factor of 10 may sometimes be unacceptable. The overhead can be reduced substantially by compiling in the profiling code, e.g. by first effecting a source-to-source transformation to splice in the profiling code. The biggest benefit accruing from this is that the table management code can be made substantially more efficient, since the addresses of the appropriate table entries are known at compile time: this enables indexed access to the tables, which is significantly faster than the process of getting the ‘‘`prof$-`’’ predicate name from the call at runtime, extracting the name of the original predicate from it, then constructing the name of the corresponding table predicate, and hashing on the name of this predicate. The compiled approach also permits profiling to be carried out at the clause level, allowing a more discriminating examination of the backtracking behavior of predicates. Finally, compiling in the profiling code permits the compiler to carry out certain subroutine linkage optimizations which have to be turned off when calls are intercepted at runtime; this serves to further reduce the runtime overhead. Compiling in the profiling code loses some of the flexibility of interactive profiling, but the benefits of more efficient execution and clause-level profiling may be worth this loss to the user. One problem with compiling in the profiling code is that dynamic predicates, i.e. those that have clauses added or deleted at runtime via *assert* or *retract*, cannot be handled very well: however, many recent Prolog implementations require that such predicates be declared by the user to be dynamic, and in such cases the compiler can generate code that causes calls to dynamic predicates to be trapped to the profiler at runtime, much as has been described earlier.

6. Conclusions

Profilers are very useful for detecting those parts of a program that are executed frequently, or in which the program spends a lot of its time. This information can then be used to recode and optimize such ‘‘hot spots’’. As such, profilers are an essential tool for software developers. With the growing interest in very high level languages like Prolog, the development of tools for such languages also becomes necessary. It turns out, however, that profiling techniques for traditional languages are inadequate for languages like Prolog. This is because control flow in Prolog, which involves backtracking and failure, can be much more complicated than in traditional languages. Another problem is that it is difficult to predict, *a priori*, the amount of time that Prolog's primitive operation, unification, may take, because this depends on the size of the input. Finally, programming environments for languages like Prolog demand interactive tools where profiling can be turned on or off interactively, modules linked in dynamically, etc., without laborious and time consuming recompilation or relinking. In this paper we describe a simple interactive profiler for Prolog. Our main purpose is to highlight how backtracking and failure can be handled correctly. However, the principles illustrated here can be adapted, without much difficulty, to have the compiler generate profiling code for the procedures or modules to be profiled: this would be somewhat less flexible, but would incur less overhead during profiling.

Acknowledgements

David S. Warren pointed out potential problems in the profiler's interaction with *cut*. The author is also grateful to Suzanne Dietrich and the anonymous referees for numerous suggestions which led to a significant improvement in the presentation of the paper.

Appendix: Code for Some of the Main Predicates Involved

```
$profile(Call) :-
     get_pred_name(Call, Pred, Arity),
     ( (counting(Pred, Arity) ; timing(Pred, Arity)) →
           $process_call(Pred, Arity, Call) ;
        call(Call)                    % not being counted or timed
     ).

$process_call(Pred, Arity, Call) :-
     entry_update(Pred, Arity, Rval),
     ( (call(Call),
        ( (cputime(T) ;                           % success
           backtrack_update(Pred, Arity, Rval)   % backtrack
           ),
           exit_update(Pred, Arity, T, Rval)
         ) ;
         failure_update(Pred, Arity, Rval)       % failure
     ).

entry_update(Pred, Arity, NoTime) :-
     (counting(Pred, Arity) →
       increment_callcount(Pred, Arity) ;
       true                           % not being counted */
     ),
     (timing(Pred, Arity),
      get_reccall_bit(Pred, Arity, NoTime), NoTime = 0) →
           (set_reccall_bit(Pred, Arity, 1)
            cputime(T),
            set_TimeLastEntered(Pred, Arity, T)
           ) ;
           NoTime = 1                 % not being timed */
      ).

backtrack_update(Pred, Arity, NoTime) :-
     cputime(T),
     (NoTime = 0 →
        set_TimeLastEntered(Pred, Arity, T) ;
       true
     ),
     (counting(Pred, Arity) →
        increment_bktrack_count(Pred, Arity) ;
```

```
        true
    ),
    fail.

exit_update(Pred, Arity, T, NoTime) :-
    NoTime = 0 ->
       update_tables(Pred, Arity, T) ;
       true.

failure_update(Pred, Arity, NoTime) :-
    cputime(T),
    (NoTime = 0 ->
         update_tables(Pred, Arity, T);
         true
    ),
    (counting(Pred, Arity) ->
       increment_failure_count(Pred, Arity) ;
       true
    ),
    fail.

exit_update(Pred, Arity, T1) :-
    set_reccall_bit(Pred, Arity, 0),
    get_pred_time(Pred, Arity, OldTime),
    get_TimeLastEntered(Pred, Arity, LastEntry),
    NewTime is OldTime + T1 - LastEntry,
    set_pred_time(Pred, Arity, NewTime).
```

References

1. J. Bentley, Profilers, *Communications of the ACM 30*, 7 (July 1987), 587-592.
2. M. Bruynooghe and L. M. Pereira, Deduction revision by intelligent backtracking, in *Implementations of Prolog*, J. A. Campbell (ed.), Ellis Horwood Ltd., Chichester, 1984, pp. 194-215.
3. M. Carlsson, On Compiling Indexing and Cut for the WAM, Research Report R86011, Swedish Institute of Computer Science, Spanga, Sweden, Dec. 1986.
4. J. Chang and A. M. Despain, Semi-Intelligent Backtracking of Prolog Based on Static Data Dependency Analysis, in *Proc. 1985 Symposium on Logic Programming*, Boston, July 1985, pp. 10-21.
5. W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, New York, 1981.
6. S. K. Debray, The SB-Prolog System, Version 2.3.2: A User Manual, Tech. Rep. 87-15, Department of Computer Science, University of Arizona, Tucson, AZ, Dec. 1987. (Revised March 1988).
7. S. W. Dietrich, Extension Tables: Memo Relations in Logic Programming, in *Proc. Fourth IEEE Symposium on Logic Programming*, San Francisco, CA, Sep. 1987, pp. 264-272.
8. M. M. Gorlick and C. F. Kesselman, Timing Prolog Programs Without Clocks, in *Proc. Fourth IEEE Symposium on Logic Programming*, San Francisco, CA, Sep. 1987.
9. S. L. Graham, P. B. Kessler and M. K. McKusick, An Execution Profiler for Modular Programs, *Software Practice and Experience 13*(1983), pp. 671-685.
10. R. Griswold and M. Griswold, *The Icon Programming Language*, Prentice Hall, Inc., 1983.
11. L. R. Power, Design and Use of a Program Execution Analyzer, *IBM Systems Journal 22*, 3 (1983), pp. 271-294.
12. R. L. Sites, Programming Tools: Statement Counts and Procedure Timings, *SIGPLAN Notices 13*, 12 (Dec. 1978), 98-101.