Profiling for Run-Time Checking of Computational Properties and Performance Debugging in Logic Programs

Edison Mera[1], Teresa Trigo[2],
Pedro Lopez-García[2,3], and Manuel Hermenegildo[2,4]

[1] Complutense University of Madrid (UCM), Spain
[2] IMDEA Software Institute, Spain
[3] Spanish Research Council (CSIC), Spain
[4] School of Computer Science, Technical University of Madrid (UPM), Spain
edison@fdi.ucm.es, herme@fi.upm.es,
{teresa.trigo,pedro.lopez,manuel.hermenegildo}@imdea.org

Abstract. Although several profiling techniques for identifying performance bottlenecks in logic programs have been developed, they are generally not automatic and in most cases they do not provide enough information for identifying the root causes of such bottlenecks. This complicates using their results for guiding performance improvement. We present a profiling method and tool that provides such explanations. Our profiler associates cost centers to certain program elements and can measure different types of resource-related properties that affect performance, preserving the precedence of cost centers in the call graph. It includes an automatic method for detecting procedures that are performance bottlenecks. The profiling tool has been integrated in a previously developed run-time checking framework to allow verification of certain properties when they cannot be verified statically. The approach allows checking global computational properties which require complex instrumentation tracking information about previous execution states, such as, e.g., that the execution time *accumulated* by a given procedure is not greater than a given bound. We have built a prototype implementation, integrated it in the Ciao/CiaoPP system and successfully applied it to performance improvement, automatic optimization (e.g., resource-aware specialization of programs), run-time checking, and debugging of global computational properties (e.g., resource usage) in Prolog programs.

Keywords: profiling, run-time checking, performance debugging, resource usage estimation/verification, logic programming.

1 Introduction

Profilers have been developed in the context of several programming paradigms: imperative [5,16] (including object oriented [7]), functional [15,14], logic [3,9,4], or integrations of some of them, such as the functional logic languages Curry and Toy [1]. In this paper we focus our attention on profilers for logic programs, and in particular for the Prolog language. The implementation of Prolog profilers has

the added complexity w.r.t. more traditional paradigms of having to deal with its specific features such as non-determinism and the possibility of failure, which makes it necessary to deal with backtracking (and, hence, with choice points), and search pruning operators (like the cut). There exist some implementations of profilers for the Prolog language (e.g., [4,3]). However, in order to fill some gaps and to broaden the range of applications, we have developed a profiler for Prolog that has the following original features:

1. *It is based on the concept of cost center.* We have adapted the cost center definition of Morgan [14], developed in the context of functional programming, to support the unique features of logic programming. A cost center, as we will explain later in detail, is a program point (such as a procedure or a call in a clause body) where data about computational events is accumulated each time the point is reached by the program execution control flow. This allows measuring accumulated execution time of program procedures that do not overlap, i.e., the total resource usage of a program can be computed in a compositional way, by adding the execution time associated to each cost center. A cost center-based profiler with this property has been developed for functional programming [15], however, as far as we know, no implementation of this kind of profiler has been developed for logic programs.
2. It allows *preserving the precedence of cost centers in the call graph.* It provides separate accumulated resource usage information for a given procedure *depending on where it is called from*, i.e., it is a *call graph profiler* for Prolog. We have taken the call graph profiling approach of [16] as starting point and we have adapted it in order to deal with the more complex execution model of Prolog, taking failure, backtracking, and pruning operators into account. The SWI profiler is to our knowledge the only Prolog profiler that keeps the precedence between the caller and the callee, but it does not support the concept of cost center.
3. It *can measure a wide range of computational properties and events*, such as execution time, execution steps, numbers of calls, failures, exits, redos, choice point creations, cut executions, choice points removed by the cut operator, or the percentage of the accumulated cost of a predicate with respect to the total cost of the program. We use in the rest of the paper the term "resources" to refer to any of these properties. Although the current implementation is not fully parametric w.r.t. resources, it can be easily generalized as it was done with the static resource analysis integrated in `CiaoPP` [12].
4. It is *used for run-time checking of computational properties.* For this purpose, it is *tightly integrated in an advanced program development framework* which incorporates in a uniform way run-time checking, static verification, unit testing, debugging, and optimization. To our knowledge, no profiler has been used for this purpose or integrated in such an environment to date.
5. It includes a (configurable) *automatic method for detecting procedures that are performance bottlenecks* following several heuristics. The method automatically associates cost centers to procedures in an iterative process. Previous approaches are not automatic (e.g., [3,15,1]), so that the programmer is responsible for configuring cost centers iteratively based on the information

returned by the profiler until the root cause of the bottleneck is detected. We show that the configuration of cost centers can be automated, as we will explain further, by exploring a (static or dynamically) generated call graph until the root cause of the bottleneck is detected.

6. It is able to *point at the part of the program that is responsible for the bottleneck*, guided by any arbitrary resource (like time, event counts, etc.) and to provide explanations at different granularity levels. This information includes an automatically generated picture of (a sub-graph of) the call graph (see Section 6). Existing profilers only provide information about where the bottlenecks of the programs are without any kind of explanation about the root causes, requiring that additional techniques be applied in order to identify such causes.
7. It *combines time profiling with count profiling*, which has proved to be non-trivial [9], and supports modularity, allowing the specification of which modules should be instrumented for profiling. This feature of our profiler is possible thanks to the usage of `Ciao`'s module system and the automatic code transformation provided through `Ciao`'s semantic packages.
8. It *uses global static analysis* to reduce the overhead of the profiling process.

2 A Cost Center-Based Approach to Profiling

Fundamental to our approach to profiling is the concept of *cost center*, which is inspired by the one defined by Morgan [14] in the context of functional languages.

A *cost center* for us is a program point where data about computational events is accumulated each time the point is reached by the program execution control flow. In our current implementation both predicates and literals in body clauses can be marked as cost centers. However, for the sake of brevity, in this paper we will only describe cost centers at the predicate level. We also introduce a special cost center, named *remainder cost center* (denoted *rcc*), which is used for accumulating data about events not corresponding to any defined cost center.

In order to deal with the control flow of Prolog, we adopt the "box model" of Byrd [2], where predicates (procedures) are seen as "black boxes" in the usual way. Since the simple call/return view of procedures is not enough to capture backtracking, this model uses a "4-port box view." Namely, given a *goal* (i.e., a unique run-time call to a predicate), the four ports (events) in Prolog execution are: (1) *call* (start to execute the goal), (2) *exit* (succeed in producing a solution to the goal), (3) *redo* (attempt to find an alternative solution to the goal), and (4) *fail* (exit with failure, if no further solutions to the goal are found). Thus, there are two ports for "entering" the box (*call* and *redo*), and two ports for "leaving" it (*exit* and *fail*).

Definition 1 (Calls relation). *We define the* calls *relation between predicates in a program as follows: p calls q, written $p \rightsquigarrow q$, if and only if a literal with predicate symbol q appears in the body of a clause defining p. Let $\rightsquigarrow^+$ denote the transitive closure of $\rightsquigarrow$.*[1]

[1] For simplicity we provide a static definition of the call graph. However, in practice, it is dynamically built, and thus it deals safely with meta-calls.

Definition 2 (Cost center set). *Given a program P to be profiled, the cost center set for P (denoted C_P), is defined as $C_P = \{p \mid p$ is a predicate of P marked as a cost center$\} \cup \{rcc\}$, where rcc is the remainder cost center.*

Definition 3 (Cost center graph). *The* cost center graph *of a program P (denoted G_P) is the graph defined by the set of nodes C_P and the set of edges $E = E' \cup \{(rcc, rcc)\}$, such that $(p, q) \in E'$ iff:*

1. *p is not the remainder cost center (i.e., $p \neq rcc$), $q \neq rcc$, and $p \rightsquigarrow^+ q$ through some path where all of its nodes (except the origin and destination) are not in C_P; or*
2. *$p = rcc$ and: (a) q is an entry point of program P such that $q \in C_P$, or (b) for some predicate r being an entry point of P, $r \rightsquigarrow^+ q$ through some path where all of its nodes (except the destination) are not in C_P.*

Definition 4 (Edge-accumulated resource usage). *Each edge $(c, d) \in G_P$ has a data structure R_{cd}, which contains the addition of resource usages over all the times that the cost center d was entered from cost center c, until a new cost center is entered or the computation finishes. This allows giving separate resource usage information for a given procedure depending on where it is called from.*

Our profiler is *parametric* w.r.t. the enter/leave ports, i.e., R_{cd} contains matrices of the form $Resource[enter][leave]$ ($enter \in \{call, redo\}$, $leave \in \{exit, fail\}$), whose elements are counters to keep track of the usage of several resources for the four possible "enter/leave" port combination (cf. the "4-port box" of node d). For example we keep track of the number of times that each of the four "enter/leave" port combination happens during program execution in $Counts[enter][leave]$. Execution times are also tracked in $Ticks[enter][leave]$.

Example 1. We are going to illustrate how the resource usage information is stored in the edges of the cost center graph during the profiling process. At any time in this process, only one edge is active. When execution enters a predicate which is defined as a cost center, the resource usage monitored so far is stored in the active edge, it is deactivated, and then another edge is activated. Consider program `p`, and its call graph and cost center graph in Figure 1. Before starting program execution, the active edge is `(rcc, rcc)`. Then, when execution starts, the partial counters are reset and `p` is called. Since `p` is defined as a cost center, the resource usage monitored so far in the partial counters is accumulated in the active edge `(rcc, rcc)`, the partial counters are reset, and the active edge changes to `(rcc, p)`. Then, the execution of the body of `p` starts by executing `q`. Since `q` is not defined as a cost center, the active edge remains the same as before, `(rcc, p)` (and the partial counters are not reset). When the execution of `q` finishes, `r` is called. Since `r` is defined as a cost center, the resource usage monitored so far in the partial counters is accumulated in the active edge `(rcc, p)`, the partial counters are reset, and the active edge changes to `(p, r)`. Since `r` is the last call in the definition of `p`, when the execution of `r` finishes, the resource usage monitored so far in the partial counters is accumulated in `(p, r)` and program execution finishes.

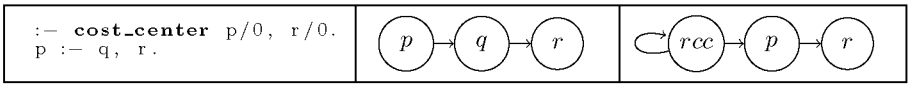

Fig. 1. Source code, call graph and cost center graph for Example 1

Definition 5 (Accumulated resource usage of a cost center). *The accumulated resource usage of a given cost center d (denoted R_d) is the sum of the resource usage for all times cost center d is entered either in forwards (i.e., via the* call *port) or backwards (i.e., via the* redo *port) execution, until a new cost center is entered or the computation finishes.*

The accumulated resource usage of a cost center can be obtained as the sum of the accumulated resource usages of its incoming edges: $R_d = \sum_{(c,d)\in E} R_{cd}$.

Our definition of accumulated resource of a cost center is compositional, in the sense that the total resource usage of a program P, denoted R_P, is the addition of the accumulated resource usage of all its cost centers: $R_P = \sum_{c\in C_P} R_c$. In contrast, in traditional profilers, the accumulated execution times for different predicates may overlap (and thus adding them may yield a result greater than their actual resource usage).

3 Integrating Profiling with Verification and Debugging

In this section we explain how our profiler is integrated within the `Ciao/CiaoPP` verification/debugging framework, which incorporates in a uniform way run-time checking, static verification, unit testing, debugging, and optimization [6,10]. The run-time checking of program state properties such as traditional types or modes can be performed relatively easily. This is in part due to the fact that properties are written in the source language and *runnable* (facilitated by the underlying logic engine), which simplifies the program transformation that adds run-time checks. However, the run-time checking of global computational properties requires monitoring, which is performed by our profiler. Figure 2 gives an overall

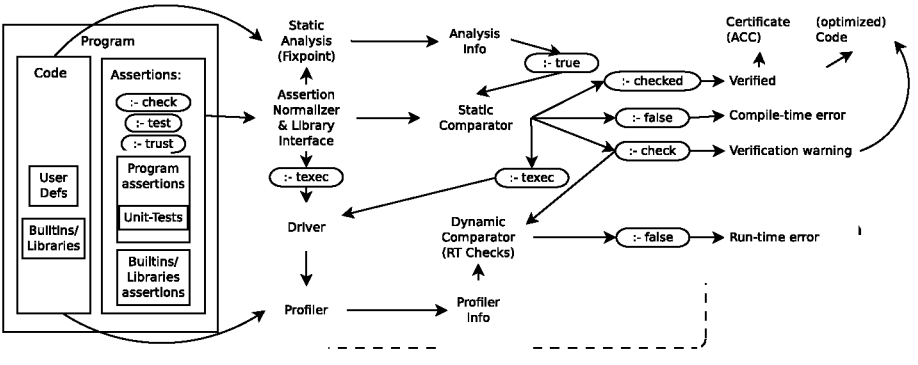

Fig. 2. The `Ciao` assertion framework (`CiaoPP`'s verification/testing architecture)

```
:- cost_center qsort1/2, qsort2/2.

qsort1(A,B) :- qsort(A,B).
qsort2(A,B) :- qsort(A,B).

qsort([],[]).
qsort([X|L],R):-
        partition(L,X,L1,L2),
        qsort1(L1,R1),
        qsort2(L2,R2),
    append(R1,[X|R2],R).
```

```
partition([],_,[],[]).
partition([H|L],X,[H|L1],L2):-
    H < X, !,
    partition(L,X,L1,L2).
partition([H|L],X,L1,[H|L2]):-
    H >= X,
    partition(L,X,L1,L2).

append([],B,B).
append([H|A],B,[H|C]) :-
    append(A,B,C).
```

Fig. 3. Source code for qsort with cost center declarations (at predicate level)

view of such framework, placing the profiling tool in context. Hexagons represent the tools involved while arrows indicate the communication paths among them. The process input is the user program, *optionally* including a set of assertions that always includes the assertions present for predicates exported by any libraries used (left part of Figure 2), and, optionally, it can include unit tests.

In this paper we are interested in a subset of the versatile `Ciao` assertion language which allows expressing global computational properties whose runtime checking requires the use of our profiler. A detailed description of the full assertion language can be found in [6]. For brevity, we only introduce the class of **`pred` assertions**, which describes a particular predicate and, in general, follows the schema:

`:- pred` *Pred* [`:` *Precond*] [`=>` *Postcond*] [`+` *Comp-Props*]`.`

where *Pred* is a predicate symbol applied to distinct free variables and *Precond* and *Postcond* are logic formulae about execution states. An execution state is defined by the bindings of values to variables in a given execution step (in logic programming terminology, a substitution). *Precond* is the precondition under which the `pred` assertion is applicable. *Postcond* expresses that in any call to *Pred*, if *Precond* holds in the calling state and the computation of the call succeeds, then *Postcond* also holds in the success state. Finally, the *Comp-Props* field is used to describe properties of the whole computation of the calls to predicate *Pred* that meet *Precond* (e.g., resource usage properties). For example, the following assertion for the quick-sort program in Figure 3:

```
:- pred qsort(A,B) :( list(A,num),var(B)) => (list(A,num),list(B,num))
        +(cost(ub,steps,length(A)*log(length(A))), not_fails, is_det).
```

states that for any call to predicate `qsort/2` with the first argument bound to a list of numbers and the second one a free variable, if the call succeeds, then the second argument is also bound to a list of numbers. It also states that (for any of such calls) an upper bound on the number of resolution steps required to execute qsort/2, is $length(A) \times log(length(A))$, a function on the length of list A. This is of course false, but we will see later in this section how we can detect it using our profiler. Additionally, `not_fails` and `is_det` express that the previous calls do not finitely fail (i.e., they produce at least one solution or do not terminate) and are deterministic (i.e., they produce at most one solution at

most once), respectively. The `cost` construct for expressing resource usages, as illustrated in the previous sample assertion, follows the schema:

cost(*Approx*, *Res_Name*, *Arith_Expr*)

where the *Res_Name* field expresses which resource the assertion refers to. It is a user-provided identifier which gives a name to each particular resource that needs to be tracked, verified, or checked. *Arith_Expr* is an arithmetic function that expresses the resource usage of the predicate as a function of input data sizes. The *Approx* field states, for example, whether *Arith_Expr* is providing an exact value (**eq**), an upper bound (**ub**), or a lower bound (**lb**).

Each assertion can be in a particular *verification status*, marked with the keyword prefixes `check`, `checked`, `false`, `trust` or `true` (see the ellipses in Figure 2). The (default) status `check` determines that the assertion is to be checked. `checked` and `false` express that the assertion has already been proved correct or incorrect respectively by the system (a compile-time error is reported in the last case). `trust` expresses that the assertion is to be trusted (it provides information coming from the programmer), and `true` that the provided information is the result of static analysis and thus correct (safely approximated). We herein introduce a new status, `obs`, which means that an assertion expresses observed information (in this case, by the profiler).

In this paper we focus on the run-time checking of computational (resource-related) properties within the `CiaoPP` unified framework, giving an intuitive short description using the following example.

Example 2. Assume that we want the `CiaoPP` system to check whether the following assertion, which gives a logarithmic upper bound on the number of resolution steps of `qsort/2` as a function of the length of the input list, holds or not:

```
:- pred qsort(A,B):(list(A,num),var(B))
                  + cost(ub,steps,length(A)*log(length(A))).
```

First, the `CiaoPP` system tries to statically verify the assertion. This is done by running a static resource usage analysis (see [12]) that computes *safe* lower and upper bounds on the resource usage (number of resolution steps in this case), and then by comparing the analysis results with the specification given in the assertion. A full description of the static verification of computational/resource-related properties is given in [8]. The quick-sort program is of the kind of divide-and-conquer programs that may cause the analysis to lose precision. As a consequence, the assertion cannot be proved to be false, since the lower bound resource usage function derived by the analysis (which is linear) is not greater than the upper bound function given in the assertion. Conversely, the assertion cannot be proved to hold, because the upper bound resource usage function derived by the analysis (which is exponential) is not less or equal than the upper bound function given in the assertion. Thus, the outcome of the static verification process is "unknown" and the assertion status remains as `check`.[2] However, if the run-time checking option is selected, `CiaoPP` instruments the program with checks to be performed at run-time

[2] This can optionally produce a *verification warning* (also known as an "alarm").

for (parts of) assertions which cannot be verified statically. Failure of these checks raises run-time errors referring to the corresponding assertion. In our example, using input data automatically generated (or taken from existing unit tests [10]) the profiler performs different calls to the quick-sort program. If for some of these calls the computed number of steps is greater than the one specified in the assertion, then such assertion is false (in fact, `CiaoPP` was easily able to prove it).

4 Proposing New Computational Properties

In order to support cumulative properties, we extend the set of properties used in the assertion language, starting with the addition of `rel_cost`, which expresses relative resource usages. For example, assuming that the `qsort/2` procedure is part of a given main program, the assertion (with no postcondition):

```
:- pred qsort(A,B) : (list(A,num),var(B)) + rel_cost(ub,exectime,20).
```

expresses that the execution time of `qsort/2` is at most 20% of the total execution time of the main program. The `rel_cost` construct follows the schema:

rel_cost(*Approx*,*Res_Name*,*Percentage***)**

where *Approx* is as before, denoting an upper bound, a lower bound, or an exact value on the *Percentage* of the procedure resource (*Res_Name*) usage with respect to the total resource usage of the whole main program (from which the predicate is called) respectively.

We have also extended the `cost` and `rel_cost` property constructs with an extra argument *Type* specifying the kind of cost information we are interested in:

{`cost`,`rel_cost`}(*Approx*, *Type*, *Res_Name*, *Arith_Expr*)

defined as follows:

- **sol(I):** The cost of obtaining the `I`-th solution without considering the cost of obtaining the previous one. By definition, if `I` is greater than the number of solutions, then the related cost is zero.
- **allsols:** The cost of obtaining all the solutions. It is equivalent to the cost of applying `findall/3` over the given predicate, but subtracting the cost of `findall/3` itself.
- **call:** The cost of calling the predicate, regardless of whether it fails or succeeds (this is the value by default).
- **call_exit:** The cost of calling the predicate when it succeeds.
- **call_fail:** The cost of calling the predicate when it fails.
- **redo**: The cost of backtracking over the predicate, regardless of whether it fails or succeeds.
- **redo_exit:** The cost of backtracking over the predicate when it succeeds.
- **redo_fail:** The cost of backtracking over the predicate when it fails.

The following example illustrates how the `CiaoPP` system (with our profiler integrated and our extended run-time checking operations), monitors and checks relative resource usages at run-time.

Example 3. Consider again the `qsort/2` predicate in Figure 3, and assume that we want to know how the execution times of its recursive calls are distributed. Although as mentioned before it is possible to define cost centers at literal level, for the sake of clarity we have defined two bridge predicates (`qsort1/2` and `qsort2/2`) that are used in place of the recursive calls of `qsort/2`, and have marked them as cost centers using the following declaration:

```
:- cost_center qsort1/2, qsort2/2.
```

Assume that we profile the execution of `qsort/2` with an input list of 2500 randomly generated elements, and that our profiler outputs the assertions:

```
:- obs pred qsort1/2 + rel_cost(eq, exec_time, 48).
:- obs pred qsort2/2 + rel_cost(eq, exec_time, 47).
```

which mean that the observed execution times of `qsort1/2` and `qsort2/2` are 48% and 47% of the total execution time respectively.

Assume now that we want the `CiaoPP` system to check at run time whether the two (recursive) calls in the body of the (original) `qsort/2` are balanced (i.e., whether each recursive call consumes more or less 50% of the total execution time). For this purpose, we write the following assertions:

```
:- check pred qsort1/2 + rel_cost(ub, exec_time, 55).
:- check pred qsort2/2 + rel_cost(ub, exec_time, 55).
```

Assume that we call `qsort/2` with a non-uniformly distributed input list, and that the execution accumulates 65.01% and 8.16% of the time in the two cost centers associated to the two calls respectively. In this case, the `CiaoPP` dynamic comparator will throw a run-time checking error informing that the assertion for cost center `qsort1`/2 is violated (because the monitored execution time is greater than the one expressed in the assertion), and, thus the two calls in the body of `qsort/2` are not balanced.

In contrast to non-cumulative global properties, the previously illustrated kind of cumulative properties cannot be checked immediately at run-time, but rather at the proper time instant in the program execution. In the current implementation, such checking is done at the end of the program execution (when the program control reaches an output port where there are no pending choice points). However, some scenarios require other rules for expressing the time instant in which the checking is performed. Consider for example a service that requires the check to be made periodically at certain time intervals, or when a certain number of client requests has been reached. Also, so far the operation for accumulating resource usages has been addition. However, it is desirable to have more complex operations. For example, old measurements could be discarded, or the events weighted according to their ages or other properties.

5 Program Transformation for Profiling

Source-to-Source Transformation for (High-Level) Profiling. A predicate marked as a cost center is transformed into an equivalent one that preserves its semantics while intercepting occurrences of events inside it, by using some

Program	Cost center transformation for profiling

`:- module(ap,[append/3],[profiler]).`
`:- cost_center append/3.`
`append([],B,B).`
`append([H|A],B,[H|C]) :-`
`  append(A,B,C).`
	`'$cc$'(ap,append,3).`
	`append(E, L, R) :-`
	`  hcc_call('ap:append',3,PrevCCE,CutTo),`
	`  hcc_fail(PrevCCE,ChPt0),`
	`  '$cc$append'(A,B,C),`
	`  hcc_exit(PrevCCE,ActiveCCE,ChPt1),`
	`  hcc_redo(ActiveCCE,ChPt0,ChPt1,CutTo).`
	`'$cc$append'([],B,B).`
	`'$cc$append'([H|A],B,[H|C]) :-`
	`  '$cc$append'(A,B,C).`

Fig. 4. Cost center transformation for profiling (at predicate level)

instrumentation procedures introduced by the transformation. For example, the predicate `append/3` in Figure 4 is marked as a cost center (left hand side), and, in its transformation (right hand side), it is uniquely renamed to `'$cc$append'/3`. In order to avoid calls to instrumentation procedures along all recursive calls to `append/3`, the body of the recursive clause of `'$cc$append'/3` is transformed so that it calls `'$cc$append'/3` instead of `append/3` (this also avoids the destruction of last call optimization.).

A brief description of the instrumentation predicates follows. They operate on the cost center graph. Any edge in such graph (CC-edge in the following), contains the already described (non backtrackable) arrays $Counts[enter][leave]$ and $Ticks[enter][leave]$ (Section 2). An implicit stack whose elements are pairs of CC-edges (variables `PrevCCE` and `ActiveCCE`) is used to keep the active CC-edge, and to restore the previous CC-edge when the control flow leaves the active one (so that the precedence of cost centers in the call graph is preserved):

- `hcc_call(+Name,+Arity,-PrevCCE,-CutTo)`: activates the CC-edge whose destination is `Name/Arity` and origin the destination of the previous CC-edge. Unifies `PrevCCE` with a pointer to the previous CC-edge. Sets the flag named "*entryport*" (associated to the active CC-edge) to the value "`call`", in order to track that the predicate `Name/Arity` has been entered through the `call` port. Unifies `CutTo` with a pointer to the top of the current choice point stack.
- `hcc_fail(+PrevCCE,-ChPt0)`: pushes a choice point on the stack in order to execute instrumentation code upon backtracking (after failure occurs), and unifies `ChPt0` with a pointer to such choice point. The instrumentation code executed upon backtracking increments by one the value of $Counts[entryport][\texttt{fail}]$ associated to the active CC-edge, [3] and changes the active CC-edge to `PrevCCE`.
- `hcc_exit(+PrevCCE,-ActiveCCE,-ChPt1)`: increments by one the value of $Counts[entryport][\texttt{exit}]$ associated to the active CC-edge. Unifies `ActiveCCE` with a pointer to the active CC-edge and `ChPt1` with a pointer to the top of the current choice point stack. Changes the active CC-edge to `PrevCCE`.
- `hcc_redo(+ActiveCCE,+ChPt0,+ChPt1,+CutTo)`: pushes a choice point on the stack to execute instrumentation code upon backtracking. Checks whether

[3] Note that the *entryport* flag can take the values `call` or `redo`.

`ChPt0` and `ChPt1` point to the same choice point, in which case the goal is deterministic (i.e., no choice points have been created during its execution), and all choice points up to `CutTo` are removed (namely, the ones introduced by `hcc_fail/2` and `hcc_redo/4` itself). The instrumentation code executed upon backtracking sets the "*entryport*" flag (associated to the active CC-edge) to the value "`redo`," and changes the active CC-edge to `ActiveCCE`.

Static Cost Center Optimization using `CiaoPP`. The overhead introduced by the transformation of cost centers described before can be reduced by using static analysis. There are situations where it can be ensured that some of the instrumentation predicates (or combinations of them) introduced by such transformation will never be reached. For example, when a predicate (or literal) marked as a cost center does not introduce choice points, always succeeds, or always fails. Thus, such unreachable instrumentation predicates can be removed. Our profiler detects these situations by using the information inferred by the `CiaoPP` analyzers [6] (such as non-determinism and non-failure). It also introduces specialized versions for reachable combinations of instrumentation predicates. Although these specialized versions increase the size of the instrumented program, they can significantly reduce the overhead introduced by the profiler. Figure 5 shows (right hand side) some of the optimized cost center transformations (which introduce specialized versions of the instrumentation predicates) performed by using information inferred by `CiaoPP` analyzers, that is expressed as assertions (left hand side).

Assertion	Specialized Cost Center Transformation
`:- true pred Goal + (no_choicepoint, not_fails).`	`hcc_call_ncnf(Name,Arity,PrevCCE), call(Goal), hcc_exit_ncnf(PrevCCE).`
`:- true pred Goal + no_choicepoint.`	`hcc_call(Name,Arity,PrevCCE,CutTo), hcc_fail_nc(PrevCCE), call(Goal), hcc_exit_nc(PrevCCE,CutTo).`
`:- true pred Goal + not_fails.`	`hcc_call_nf(Name,Arity,PrevCCE,CutTo), call(Goal), hcc_exit(PrevCCE,ActiveCCE,ChPt1), hcc_redo_nf(ActiveCCE,ChPt1,CutTo).`

Fig. 5. Cost center transformation optimization

Enriching Information with Low-level Profiling. We set up several hooks at some relevant points in the engine. Their implementation is located in a separate module. To avoid run-time overhead, such hooks are made available by compiling the engine with an option that enables them. For example, there are hooks that are called when a fail causes the next choice point to be tried (`lph_fail_redo(wam)`), when a cut is executed (`lph_cut(wam)`), and when a given predicate `pred` is called (`lph_exit_call(wam, pred)`, where the variable `wam` is a structure that represents the current state of the virtual machine). Such hooks remain uninstantiated until the procedure `profile/1` is used over a given goal, in which case they are instantiated to actual functions that perform the profiling itself. The end of the profiling leaves the hooks uninstantiated

again. When performing low-level profiling, each edge of the cost center graph contains the following (non backtrackable) data structures: (1) the already described ones used in high-level profiling; (2) two matrices, $Cuts[enter][leave]$ and $SCuts[enter][leave]$, that keep the number of cut executions that remove or do not remove choice points respectively (which allows for example detecting useless pruning operations and checking that a cut actually prunes branches); and (3) a hash table used to track the execution of predicates. The key of each entry in the table is a predicate name/arity, and its fields are: (a) two matrices similar to the already described $Counts[enter][leave]$ and $Ticks[enter][leave]$), but referred to "predicate heads," and (b) a counter ($Skips$) to keep the number of choice points that are removed for that predicate by some pruning operator (cut) execution.

The low-level profiling allows tracking information on predicates that have not been marked as cost centers (e.g., library predicates), and therefore, to detect that certain low-level or library predicates are being used by our program without us being aware (which could happen if syntactic expansions are used). It also allows detecting backtracking in predicate heads (useful to detect predicates that do not succeed in the first clause, or that are not indexed by the first argument).

6 Automatic Performance Bottleneck Detection

Defining cost centers by hand in order to detect performance bottlenecks is a time-consuming task. As mentioned before, one of the original features of our profiling tool is a method for identifying performance bottlenecks in an automatic way, which uses an iterative process that defines cost centers at each iteration. For space reasons, we give a high-level description of the algorithm and refer the reader to [11] for details and examples. The method provides the sub-graph (a tree in fact) of the cost center graph that is responsible for the performance leak. It can be applied to modular programs and allows providing a list of modules whose predicates must be taken into account. The input call graph to the method is dynamically constructed (defining cost centers for all predicates in the selected modules, and executing once with profiling activated).

Starting with the initial goal as the current predicate, at each iteration the children of the current predicate in the call graph (i.e., its called predicates) are computed. They and *the previous cost centers* in the current branch of the cost center graph (including the current predicate), are marked as cost centers. Then, the goal is profiled, and, after that, the set of cost centers called by the current predicate and the amount of resource that each one consumes are computed. To ensure termination, any predicate previously defined as a cost center (including the current predicate) is removed from this set. If after this removal there are no cost centers left in the set, then the process finishes returning the graph built so far. Otherwise, it selects the relevant cost centers of the called cost centers set, according to a heuristic (which is a parameter of the method), provided by the user. Some examples of heuristic selection rules are: (1) select the N predicates that consume more resources, (2) select the ones whose resource consumption is larger than a given percentage of the total resource usage, or (3) select the predicates whose number is not larger than a percentage X of the number of program predicates, and which together consume a percentage of the total resource

usage greater than a given bound Y. Independently of the heuristic used, a given predicate is selected at most once (and thus, the sub-graph returned is a tree).

We have also developed a method for drawing automatically the sub-graph of the cost center graph that is responsible for the performance leak, where different colors and sizes are used to express the accumulated resource usage in each cost center.

7 Experimental Results

We have performed an experimental assessment of our profiler. The results are shown in Table 1 for two different platforms with different processors and OS: an **Intel** Core i7, 4 cores x 2.67GHz (2 threads per core), 12GB of RAM, Ubuntu Linux 10.10 (kernel 2.6.35) and an Ultra**Sparc**-T1, 8 cores x 1GHz (4 threads per core), 8GB of RAM, SunOS 5.10. In both platforms, the execution has been *locked* to a single core in order to avoid erroneous execution time measurements. The profiler measures execution times using a high-resolution timer, which allows giving relevant values in situations where other methods would get a zero value. The first and second columns of the table show the benchmarks used[4] and the number of predicates defined in them respectively. For each platform, the `Obs` column shows the observed execution time without profiling (given in milliseconds). The following two columns grouped under (`Est.Dev.`) are meant to assess the accuracy of our profiler in monitoring execution times. They show the ratio between the execution time estimated by the profiler and `Obs` for two levels of profiling instrumentation: `hl`, which only performs the high level source-to-source transformation, and `ll`, which besides performing this transformation, also introduces hooks in the engine (i.e., it also performs low-level profiling). The columns grouped under `Bot.D.` refer to the automatic performance bottleneck detection process (described in Section 6), where the performance is measured in terms of execution time, the heuristic of selecting the goal with the largest execution time in each iteration has been followed, and the profiling has been performed without engine hooks (since they do not improve execution time measurements). Column `#it` shows the number of iterations needed to complete such process, and column $\texttt{ov}_b$ shows its overhead, calculated as $\texttt{ov}_b = \frac{T_{tot} - T_{pr} - T_{co}}{\texttt{\#it} \times \texttt{Obs}}$, where the total time due to executing the program with profiling ($T_{pr} = \sum_{i=1}^{\texttt{\#it}} Prof_i$) and the total time due to its compilation ($T_{co} = \sum_{i=1}^{\texttt{\#it}} Comp_i$) have been subtracted from the total time (T_{tot}) in order to isolate the time due to the bottleneck detection process itself. Averages are also provided in the last row of the table.[5]

The columns under `Profiling Ov.` relate to the run-time overhead introduced by the different program transformations/instrumentations described in Section 5. They are grouped into two sub-columns, showing the results when the instrumentation has been optimized using `CiaoPP`'s static analyzers (`Optim.`), and without such optimization (`N.Op.`). In both cases we present the results with (`ll`) and without (`hl`) engine hooks activated respectively. The overheads

[4] Source code for the examples is available at `http://www.clip.dia.fi.upm.es/profiling/`

[5] Weighted average taking the observed execution time as weight in all cases.

Table 1. Experimental assessment of the profiler

Program	# P.	Intel									Sparc								
		Obs (ms)	Est. Dev.		Bot.D.		Profiling Ov.				Obs (ms)	Est. Dev.		Bot.D.		Profiling Ov.			
					#	ov_b	Optim.		N.Op.					#	ov_b	Optim.		N.Op.	
			hl	ll	it	(%)	hl	ll	hl	ll		hl	ll	it	(%)	hl	ll	hl	ll
mem	5	67	1.0	1.4	4	2.40	1.0	4.6	1.0	4.6	377	1.0	1.4	4	4.65	1.0	7.8	1.0	7.8
guardians	9	182	1.0	1.6	4	0.95	1.0	3.8	1.0	3.8	959	1.0	1.3	4	1.31	1.0	6.1	1.0	6.1
color_map	5	99	1.0	1.8	2	1.26	1.0	4.3	1.0	4.3	558	1.0	1.3	2	2.24	1.0	6.4	1.0	6.4
bignums	4	102	1.0	2.4	3	1.46	1.0	2.6	1.0	2.6	2178	1.0	1.0	3	0.33	1.0	1.1	1.0	1.1
wumpus	65	211	1.0	1.6	4	1.53	1.0	5.6	1.0	5.6	1018	1.0	1.7	4	1.56	1.0	11.4	1.0	11.4
solve_jugs	6	255	1.0	1.6	4	0.52	1.0	4.0	1.0	4.0	1237	1.0	1.5	4	0.98	1.0	7.1	1.0	7.1
qsort	4	76	1.1	2.1	3	1.82	1.3	6.3	1.3	6.3	402	1.1	2.2	3	1.46	1.4	10.5	1.5	10.6
sudoku	12	72	1.3	1.7	7	2.42	1.5	5.3	1.6	5.6	359	1.2	1.5	7	4.44	1.7	8.7	1.8	9.4
zebra	5	40	1.3	1.5	3	3.53	1.5	5.7	1.5	5.7	184	1.4	1.4	3	7.16	1.9	10.6	1.9	10.6
hanoi	3	128	1.2	1.9	3	1.12	1.6	8.3	2.1	9.6	665	1.2	1.7	3	1.57	1.9	14.6	2.4	16.9
flat	4	65	1.5	1.8	4	2.41	2.9	10.0	4.2	14.2	323	1.5	1.6	4	2.54	3.8	17.2	5.4	24.7
substitute	3	187	2.0	2.3	3	0.71	2.9	12.7	2.9	12.6	1102	2.0	2.0	3	0.51	3.4	19.6	3.4	19.7
queens	16	92	2.0	2.2	6	1.79	3.1	15.7	3.1	15.7	429	2.3	1.9	6	4.30	4.4	28.4	4.4	28.4
Average		121	1.2	1.8	3	1.34	1.5	6.7	1.6	7.0	753	1.2	1.5	3	1.59	1.6	9.5	1.7	10.0

(`ll` and `hl`) are given as a ratio $\frac{Prof}{\texttt{Obs}}$, where $Prof$ refers to the execution time when the profiler is activated, with the cost centers assigned by the automatic bottleneck detection process reported in the columns grouped under `Bot.D.` (the number of selected cost centers is `#it` -1), and `Obs` is the value in the third column described before. The overhead ratio `hl` is very close to 1 (i.e., almost no overhead is introduced) for the first six programs in the table, while it is larger for the rest. This is because the latter perform recursive calls between cost centers. As expected, the overhead ratios (for both platforms) grow as we increase the degree of information that we want to obtain from the profiler.

It can be argued that the overhead introduced by our profiler is small for a reasonable level of profiling information, and that global static analysis indeed reduces such overhead. Interestingly, if we compare our results with those reported in [16] (which is the closest related previous work that we are aware of for which there is available data, although applied to imperative programs), the overheads of the `hl` columns under `Profiling Ov./Optim.` are of similar magnitude to those reported therein: 2.95 in the worst case, while in our results the worst overhead for the Intel platform is 3.1 (queens). However, our approach provides a richer (and more detailed) variety data.

8 Discussion and Future Work

Since its development our profiler has proved to be quite useful in practice by identifying the root causes of performance bottlenecks in several complex, real-life situations. For example, it was the key for identifying a difficult to locate performance bug in the (Ciao) CHR implementation (a complex and relatively large piece of code): a dereferencing chain for the attribute of a variable was constructed, instead of modifying the value of the attribute. Thus, the time needed for getting the value of such attribute was directly proportional to the

number of times that the attribute was modified. Our profiler has also been successfully applied to resource-aware poly-controlled partial evaluation [13]. This technique combines different control strategies to obtain optimizations that cannot be obtained using a single control technique. Once the optimizations have been obtained they are compared using some values (called *fitness values*). Our implementation has been successfully used for estimating such values.

Note that in some cases, bottlenecks can only be detected using the fine-grain information provided by our low-level profiling (via engine hooks). For example, assume that we have a read-only library which is responsible for lack of performance. In this case we are unable to define cost centers on it to perform high-level profiling. Alternatively, we can activate the engine hooks to track information about all the subroutines invoked in such library, and use it to diagnose the performance problem. Engine hooks can also profile more properties, like the number of cut executions that remove (or do not remove) choice points, failures during head unification, or choice points removed for a given predicate (see Section 5).

Although our profiler already supports several computational properties and events, these are predefined. However, as future work, it should be straightforward to extend it to allow measuring user-defined resources, in the sense of the static resource analysis currently integrated in the `CiaoPP` framework [12].

References

1. Brassel, B., Hanus, M., Huch, F., Silva, J., Vidal, G.: Run-Time Profiling of Functional Logic Programs. In: Etalle, S. (ed.) LOPSTR 2004. LNCS, vol. 3573, pp. 182–197. Springer, Heidelberg (2005)
2. Byrd, L.: Understanding the Control Flow of Prolog Programs. In: Tärnlund, S.-A. (ed.) Proceedings of the 1980 Logic Programming Workshop, Debrecen, Hungary, pp. 127–138 (July 1980)
3. Debray, S.K.: Profiling Prolog Programs. Software Practice and Experience 18(9), 821–839 (1983)
4. Ducassé, M., Noyé, J.: Tracing Prolog Programs by Source Instrumentation is Efficient Enough. Journal of Logic Programming 43, 157–172 (2000)
5. Graham, S.L., Kessler, P.B., Mckusick, M.K.: Gprof: a Call Graph Execution Profiler. In: SIGPLAN 1982: Proc. of the 1982 SIGPLAN Symp. on Compiler Construction, pp. 120–126. ACM, New York (1982)
6. Hermenegildo, M., Puebla, G., Bueno, F., López-García, P.: Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). Science of Computer Programming 58(1-2), 115–140 (2005)
7. Kazi, I.H., Jose, D.P., Ben-Hamida, B., Hescott, C.J., Kwok, C., Konstan, J.A., Lilja, D.J., Yew, P.-C.: JaViz: A Client/Server Java Profiling Tool. IBM Syst. J. 39(1), 96–117 (2000)

8. López-García, P., Darmawan, L., Bueno, F.: A Framework for Verification and Debugging of Resource Usage Properties. In: Technical Communications of ICLP. LIPIcs, vol. 7, pp. 104–113. Schloss Dagstuhl (July 2010)
9. Matos, A.B.: A matrix model for the flow of control in prolog programs with applications to profiling. Software Practice and Experience 24(8), 729–746 (1994)
10. Mera, E., López-García, P., Hermenegildo, M.: Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework. In: Hill, P.M., Warren, D.S. (eds.) ICLP 2009. LNCS, vol. 5649, pp. 281–295. Springer, Heidelberg (2009)
11. Mera, E., Trigo, T., López-García, P., Hermenegildo, M.: An Approach to Profiling for Run-Time Checking of Computational Properties and Performance Debugging. Technical Report CLIP3/2010.0, Technical University of Madrid (UPM), School of Computer Science, UPM (March 2010)
12. Navas, J., Mera, E., López-García, P., Hermenegildo, M.: User-Definable Resource Bounds Analysis for Logic Programs. In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, pp. 348–363. Springer, Heidelberg (2007)
13. Ochoa, C., Puebla, G.: Poly-Controlled Partial Evaluation in Practice. In: ACM Partial Evaluation and Program Manipulation (PEPM 2007), pp. 164–173. ACM Press, New York (2007)
14. Jarvis, S.A., Morgan, R.G.: Profiling large-scale lazy functional programs. Journal of Functional Programing 8(3), 201–237 (1998)
15. Sansom, P.M., Peyton Jones, S.L.: Formally Based Profiling for Higher-Order Functional Languages. ACM Transactions on Programming Languages and Systems 19(2), 334–385 (1997)
16. Spivey, J.M.: Fast, accurate call graph profiling. Software Practice and Experience 34(3), 249–264 (2004)