Combining Static Analysis and Profiling for Estimating Execution Times in Logic Programs

Edison Mera[1], Pedro López-García[1], Germán Puebla[1], Manuel Carro[1], and Manuel Hermenegildo[1,2]

[1] Technical University of Madrid
edison@clip.dia.fi.upm.es, {pedro.lopez,german,mcarro,herme}@fi.upm.es
[2] University of New Mexico, herme@unm.edu

Abstract. Effective static analyses have been proposed which allow inferring functions which bound the number of resolutions or reductions. These have the advantage of being independent from the platform on which the programs are executed and such bounds have been shown useful in a number of applications, such as granularity control in parallel execution. On the other hand, in certain distributed computation scenarios where different platforms come into play, with each platform having different capabilities, it is more interesting to express costs in metrics that include the characteristics of the platform. In particular, it is specially interesting to be able to infer upper and lower bounds on actual execution time. With this objective in mind, we propose a method which allows inferring upper and lower bounds on the execution times of procedures of a program in a given execution platform. The approach combines compile-time cost bounds analysis with a one-time profiling of the platform in order to determine the values of certain constants for that platform. These constants calibrate a cost model which from then on is able to compute statically time bound functions for procedures and to predict with a significant degree of accuracy the execution times of such procedures in the given platform. The approach has been implemented and integrated in the `CiaoPP` system.

Keywords: Execution Time Estimation, Cost Analysis, Profiling, Resource Awareness, Cost Models, Mobile Computing.

1 Introduction

Predicting statically the running time of programs has many applications ranging from task scheduling in parallel execution to proving the ability of a program to meet strict time constraints in real-time systems. A starting point in order to attack this problem is to infer the computational complexity of such programs (or fragments thereof). This is one of the reasons why the development of static analysis techniques for inferring cost-related properties of programs has received considerable attention. However, in most cases such cost properties are expressed using platform-independent metrics. For example, [4, 5] present a method for automatically inferring functions which capture an upper bound on the number of resolution steps or reductions that a procedure will execute as a function of the

size of its input data. In [10, 11] the method of [4, 10] was fully automated in the context of a practical compiler and in [6, 10] a similar approach was applied in order to also obtain lower bounds, which are specially relevant in parallel execution. Such platform-independent cost information (bounds on number of reductions) has been shown to be quite useful in various applications. This includes, for example, scheduling parallel tasks [8, 10, 11], where such cost bounds allow an approximate comparison among tasks (and such tasks will generally be executed in the same, parallel, machine).

However, in distributed execution and other mobile/pervasive computation scenarios, where different platforms come into play with each platform having different computing power, it becomes necessary to express costs in metrics that can be later instantiated to different architectures so that actual running time can be compared using the same units. With this objective in mind, we present a framework which combines cost analysis with profiling techniques in order to infer functions which yield bounds on platform-dependent *execution times* of procedures. Platform-independent cost functions are first inferred which are parameterized by certain constants. These constants aim at capturing the execution time of certain low-level operations on each platform. For each execution platform, the value of such constants is determined experimentally once and for all by running a set of special-purpose synthetic benchmarks and measuring their running times with a profiling toolkit that we have also developed. Once these constants are determined, they are fed into the model with the objective of predicting with a certain accuracy execution times. We have studied a relatively large number of cost models, involving different sets of constants in order to explore experimentally which of the models produces the most precise results, i.e., which parameters model and predict best the actual execution times of procedures. In doing this we have taken into account the trade-off between simplicity of the cost models (which implies efficiency of the cost analysis and also simpler profiling) and the precision of their results. With this aim, we have started with a simple model and explored several possible refinements.

In addition to cost analysis, the implementation of profilers in declarative languages has also been considered by various authors. Debray [3] showed the basic considerations to have in mind when profiling Prolog programs: handling backtracking and failure. Ducassé [7] showed a trace analyzer for Prolog, which can be applied to profiling. Sansom and Peyton Jones [13] focused on profiling of functional languages using a semantic approach and highlighted the difficulty in profiling such kind of languages. Jarvis and Morgan [12] showed how to profile lazy functional programs like those in Haskell. Brassel et al. [1] solved part of the difficulty in profiling when considering special features in functional logic programs, like sharing, laziness and non-determinism. All this work focuses on providing tools and techniques to help discover why a part of a program does not exhibit some expected performance, whereas our aim is to *predict* performance. Also, regarding the use of the profiler, instead of only profiling a program with fixed input arguments, we use the profiler to calibrate the values for the constants that appear in the cost functions, which will be instrumental in yielding execution times of procedures for a given platform and cost model.

2 Static Platform-Dependent Cost Analysis

In this Section we present the compile-time cost bounds analysis component of our combined framework. This analysis has been implemented and integrated in CiaoPP [9] by extending our previous implementations of reduction-counting cost analyses. The inferred (upper or lower) bounds on cost are expressed as functions on the sizes of the input arguments as well as several platform-dependent parameters. Once these platform-dependent parameters are instantiated with values for a given platform, such functions yield bounds on the execution times required by the computation on such platform. The analyzer can use several metrics for computing the "size" of an input, such as list-length, term-size, term-depth, integer-value, etc. Types, modes, and size measures are first automatically inferred by other analyzers which are part of CiaoPP and then used in the size and cost analysis.

2.1 Platform-Independent Static Cost Analysis

As mentioned before, our static cost analysis approach is based on that developed in [4, 5] (for estimation of upper bounds on resolution steps) and further extended in [6] (for lower bounds). In these approaches the time complexity of a clause can be bounded by the time complexity of head unification together with the time complexity of each of its body literals. For simplicity, the discussion that follows is focused on the estimation of upper bounds. We refer the reader to [6] for details on lower bounds analysis. Consider a clause $\mathtt{C}$ defined as "$\mathtt{H} :- \mathtt{L_1}, ..., \mathtt{L_m}$". Because of backtracking, the number of times a literal will be executed depends on the number of solutions that the literals preceding it can generate. Assume that $\overline{n}$ is a vector such that each element corresponds to the size of an input argument to clause $\mathtt{C}$ and that each $\overline{n}_i$, $i = 1 \ldots n$, is a vector such that each element corresponds to the size of an input argument to literal $\mathtt{L}_i$, τ is the cost needed to resolve the head $\mathtt{H}$ of the clause with the literal being solved, and $\mathtt{Sols}_{\mathtt{L}_j}$ is the number of solutions literal $\mathtt{L}_j$ can generate. Then the cost complexity of clause $\mathtt{C}$, $\mathtt{Cost}_{\mathtt{C}}(\overline{n})$, can be expressed as:

$$\mathtt{Cost}_{\mathtt{C}}(\overline{n}) \leq \tau + \sum_{i=1}^{m} (\prod_{j \prec i} \mathtt{Sols}_{\mathtt{L}_j}(\overline{n}_j)) \mathtt{Cost}_{\mathtt{L}_i}(\overline{n}_i), \tag{1}$$

Here we use $j \prec i$ to denote that $\mathtt{L}_j$ precedes $\mathtt{L}_i$ in the literal dependency graph for the clause.

Our current implementation also considers the cost of the terms created for the literals in the body of predicates, which can affect the cost expression significantly. To further simplify the discussion that follows, we restrict ourselves to the simple case where each literal is determinate, i.e., produces at most one solution. This does not mean however that our implementation is limited to deterministic programs, and our system system in fact handles non determinism (presence of several solutions for a given call) in the cost analysis. In this case, equation (1) simplifies to:

$$\mathtt{Cost}_{\mathtt{C}}(\overline{n}) \leq \tau + \sum_{i=1}^{m} \mathtt{Cost}_{\mathtt{L}_i}(\overline{n}_i). \tag{2}$$

A difference equation is set up for each recursive clause, whose solution (using as boundary conditions the cost of non-recursive clauses) is a function that yields the cost of a clause. The cost of a predicate is then computed from the cost of its defining clauses. Since it is generally not known in advance how many of the solutions generated by a predicate will be demanded, a conservative upper bound on the computational cost of a predicate can be obtained by assuming that all solutions are needed, and that all clauses are executed (thus the cost of the predicate is assumed to be the sum of the costs of its defining clauses). Taking mutual exclusion into account in order to obtain a more precise estimate of the cost of a predicate is relatively easy: the complexity for deterministic predicates can be approximated with the maximum of the costs of mutually exclusive groups of clauses.

Although the aim of the analysis of [4, 5] was the estimation of number of resolution steps, it was also pointed out that it is possible to use a number of alternative metrics as the unit of cost in the analysis, so that instead of the number of resolution steps for example the number of unifications or the number of instructions executed could be counted. In the rest of this section we explore this open issue more deeply and study how the original cost analysis can be extended in order to infer cost functions using more refined and parametric cost models, which in turn will allow achieving accurate execution time bound analysis.

2.2 Proposed Platform-Dependent Cost Analysis Models

Since the cost metric which we want to use in our approach is execution time, we take τ (in expression 2) to be the time needed to resolve the head `H` of the clause with the literal being solved plus some possible costs associated to the resolution of the clause, which we will assimilate to the cost of the head (and we will also associate to it the cost of body literals). In the following, we will refer to τ as the *clause head cost function*, under the assumption that these other costs are also taken into account. We will consider different values for τ, each of them yielding a different cost model. These cost models make use of a vector of platform-dependent constants, together with a vector of platform-independent metrics, each one corresponding to a particular low-level operation related to program execution. Examples of such low-level operations considered by the cost models are unifications where one of the terms being unified is a variable and thus behave as an "assignment", or full unifications, i.e., when both terms being unified are not variables, and thus unification performs a "test" or produces new terms, etc. Thus, we assume that τ is a function parameterized by the cost model, so that:

$$\tau = time(\Omega) \tag{3}$$

where $time(\Omega)$ is a function that gives the time needed to resolve the head `H` of the clause with the literal being solved (plus some possible costs associated to the execution of the clause such as, e.g., whether an activation record is allocated) for the cost model named Ω. We study a family of cost models such that $time(\Omega)$ is a function defined as follows:

$$time(\Omega) = time(\omega_1) + \cdots + time(\omega_v),\ v > 0 \tag{4}$$

where each $time(\omega_i)$ provides that part of the execution time which depends on the metric ω_i. We assume that:

$$time(\omega_i) = K_{\omega_i} \times I(\omega_i) \tag{5}$$

where K_{ω_i} is a platform-dependent constant, and $I(\omega_i)$ is a platform-independent cost function.

Since $time(\Omega)$ is a linear combination of platform-independent cost functions, we can write equation (4) as:

$$time(\Omega) = \overline{K}_{\Omega} \bullet \overline{I(\Omega)} \tag{6}$$

where $\overline{K}_{\Omega}$ is a vector of platform-dependent constants, $\overline{I(\Omega)}$ is a vector of platform-independent cost functions, and $\bullet$ is the dot product.

Accordingly, we generalize the definition of equation (2) introducing the clause head cost function τ as a parameter:

$$\mathtt{Cost}_{\mathtt{C}}(\tau, \overline{n}) \leq \tau + \sum_{i=1}^{m} \mathtt{Cost}_{\mathtt{L}_i}(\overline{n}_i). \tag{7}$$

A particular definition of $\overline{I(\Omega)}$ yields a cost model. We have tried with several cost models, by using different vectors $\overline{I(\Omega)}$ constructed by choosing some (or all) of the following $I(\omega_i)$ cost functions (for example, the cost model that uses *all* such functions is $\overline{I(\Omega)} = (I(step), I(viunif), I(vounif), I(giunif), I(gounif))$). In the following an *input argument* is one for which the term being passed by the calling literal is known to be non-var at the time of head unification. An *output argument* is one for which the term being passed by the calling literal is known to be a variable at the time of head unification. Whether unifications are input or output can be inferred using well-known techniques for mode analyses (in our case, those provided by `CiaoPP`).

- $I(step) = 1$.
  Here we assume that there is a constant component of the execution time when a clause is resolved (a clause neck "`:-`" is crossed). I.e., following equation (5), we are assuming for this component that:

  $$time(step) = K_{step}$$

- $I(vounif) =$ *the number of variables in the clause head which correspond to "output" argument positions.*
  Here we assume that there is a component of the execution time that is directly proportional to the number of cases where we know that both terms being unified are variables and thus unification really implies a simple assignment with a (presumably small) constant cost:

  $$time(vounif) = K_{vounif} \times I(vounif)$$

- $I(viunif) =$ *the number of variables in the clause head which correspond to "input" argument positions.*

Here we assume that there is a component of the execution time that is directly proportional to the number of cases where we know that the incoming term is non-var and the argument position in the clause is a variable. In this case the head unification for that argument is also an assignment with a small, constant cost, and there is also a cost associated with creating the input argument at the calling point, which for simplicity we will also consider constant. Given these assumptions:

$$time(viunif) = K_{viunif} \times I(viunif)$$

- $I(gounif)$ = *The number of function symbols, constants, and variables in the clause head which appear in output arguments.*
  We are assuming that there is a component of the execution time that is directly proportional to the size of the terms that have to be written into variables passed in by the calling literal, and which is proportional to the size of the number of function symbols, constants, and variables which appear in output arguments in the clause head:

$$time(gounif) = K_{gounif} \times I(gounif)$$

- $I(giunif)$ = *The number of function symbols, variables, and constants in the clause head which appear in input arguments.*
  Here we are assuming that there is a component of the execution time that is directly proportional to the number of "input" unifications, i.e., when both terms being unified are not variables, and thus unification performs a "test," and which is actually proportional to the number of function symbols, variables, and constants in the clause head which appear in input arguments (this is obviously an approximation):

$$time(giunif) = K_{giunif} \times I(giunif)$$

- $I(nargs) = arity(H)$.
  Here we are assuming that there is a component of the execution time that depends on the number of arguments in the clause head:

$$time(nargs) = K_{nargs} \times arity(H) \tag{8}$$

  This component is obviously redundant with respect to the previous ones, but we have included it as a statistical control: the experiments should show (and do show) that it is irrelevant when the others are used.

Clearly, other components can be included (such as whether activation records are created or not) but our objective is to see how far we can go with the components outlined above.

We adopt the same approach as [5, 6] for computing bounds on cost of predicates from the computed values for the cost of the clauses defining it. However, we introduce the clause head cost function τ as a parameter of these cost functions.

Let $\mathtt{Cost}_{\mathtt{p}}(\tau, \overline{n})$ be a function which gives the cost of the computation of a call to predicate `p` for an input of size $\overline{n}$ (recall that the cost units depend on the

definition of τ). Given a predicate $\mathtt{p}$, and a clause head cost function $time(\Omega)$ of the form defined in equation (6), we have that:

$$\mathtt{Cost_p}(time(\Omega), \overline{n}) = \overline{K}_{\Omega} \bullet \overline{\mathtt{Cost_p}}(\overline{I(\Omega)}, \overline{n}) \tag{9}$$

where $\overline{K}_{\Omega}$, $\overline{I(\Omega)}$ and $\overline{\mathtt{Cost_p}}(\overline{I(\Omega)}, \overline{n})$ are vectors of the form:
$\overline{K}_{\Omega} = (K_{\omega_1}, \ldots, K_{\omega_v})$,
$\overline{I(\Omega)} = (I(\omega_i), \ldots, I(\omega_v))$, and
$\overline{\mathtt{Cost_p}}(\overline{I(\Omega)}, \overline{n}) = (\mathtt{Cost_p}(I(\omega_1), \overline{n}), \ldots, \mathtt{Cost_p}(I(\omega_v), \overline{n}))$

Equation (9) gives the basis for computing values for constants K_{ω_i} via profiling (as explained in Section 4). Also, it provides a way to obtain the cost of a procedure expressed in a platform-dependent cost metric from another cost expressed in a platform-independent cost metric.

3 Refining the Cost Model: Dealing with Builtins

In this section we present our approach to the cost analysis of programs which call builtins, or more generally, predicates whose code is not available to the analyzer. We assume that there is a cost function (expressed via trust assertions [9]) for builtin predicates. In some cases, this cost function for each builtin predicate is approximated by a constant value, and in others, it is approximated by a function that depends on properties of the (input) arguments of the predicate. In particular, the cost of arithmetic builtin predicates (such as `=:=/2`, `=\=/2`, or `>/2`) is approximated by a function that depends on the number and type of arithmetic operands appearing in its arguments.

Note that this is an important improvement over the cost analysis proposed in [5] (which infers *number of resolution steps*), since one of the assumptions made in such analysis, is that calls to certain builtin predicates are not counted as a resolution step, and are thus completely ignored by cost analysis. This assumption is not realistic if we want to estimate execution times, since the cost of executing such builtins has to be taken into account.

Going into more detail, we assume that each builtin contributes with a new component to the execution time as expressed in Equation (4), that is, our cost model will have a new component $time(\omega_i)$ for each builtin predicate and arithmetic operator. Let $\odot/n$ be an arithmetic operator. The execution time due to the total number of times that such operator is evaluated is given by:

$$time(\odot/n) = K_{\odot/n} \times I(\odot/n)$$

where $K_{\odot/n}$ is a platform-dependent constant, and $I(\odot/n)$ is a platform-independent cost function. $K_{\odot/n}$ approximates the cost (in units of time) of evaluating the arithmetic operator $\odot/n$. $I(\odot/n)$ could be the number of times that the arithmetic operator is evaluated. Alternatively, it can be a cost function defined as:

$$I(\odot/n) = \sum_{a \in S} \mathtt{EvCost}(\odot/n, a)$$

and where S is the set of arithmetic expressions appearing in the clause body which will be evaluated; and $\texttt{EvCost}(\odot/n, a)$ represents the cost corresponding to the operator $\odot/n$ in the evaluation of the arithmetic term a, i.e.:

$$\texttt{EvCost}(\odot/n, A) = \begin{cases} 0 & \text{if } A \text{ is a constant} \\ & \quad \text{or a variable} \\ 1 + \sum_{i=1}^{n} \texttt{EvCost}(\odot/n, A_i) & \text{if } A = \odot(A_1, ..., A_n) \\ \sum_{i=1}^{m} \texttt{EvCost}(\odot/n, A_i) & \text{if } A \neq \odot(A_1, ..., A_n) \\ & \quad \wedge\ A = \hat{\odot}(A_1, ..., A_m) \\ & \quad \text{for some operator } \hat{\odot}/m \end{cases}$$

For simplicity, we assume that the cost of evaluating the arithmetic term t to which a variable appearing in A will be bound at execution time is zero (i.e. we ignore the cost of evaluating t). This is a good approximation if in most cases t is a number and thus no evaluation is needed for it. However, a more refined cost model could assume that this cost is a function on the size of t.

Note that this model ignores the possible optimizations that the compiler might perform. However, experimental results show that our simplified cost model gives a good approximation of the execution times for arithmetic builtin predicates. With these assumptions, equation (9) (in Section 2.2) also holds for programs that perform calls to builtin predicates, by introducing b/n and $\odot/n$ as new cost components of Ω.

A similar approach can be used for other (non-arithmetic) builtins b/n using the formula:

$$time(b/n) = K_{b/n} \times I(b/n)$$

4 Calibrating Constants via Profiling

In order to compute values for the platform-dependent constants which appear in the different cost models proposed in Section 2.2, our calibration schema takes advantage of the relationship between the platform-dependent and -independent cost metrics expressed in Equation (9). In this sense, the calibration of the constants appearing in $\overline{K}_{\Omega}$ is performed by solving systems of linear equations (in which such constants are treated as variables).
Based on this expression, the calibration procedure consists of:

1. Using a selected set of calibration programs which aim at isolating specific aspects that affect execution time of programs in general. For these calibration programs it holds that $\texttt{Cost}_\texttt{p}(I(\omega_i), \overline{n})$ is known for all $1 \leq i \leq v$. This can be done by using any of the following methods:
   - The analyzers integrated in the `CiaoPP` system infer the exact cost function, i.e. $\texttt{Cost}_\texttt{p}{}^{l}(I(\omega_i), \overline{n}) = \texttt{Cost}_\texttt{p}{}^{u}(I(\omega_i), \overline{n}) = \texttt{Cost}_\texttt{p}(I(\omega_i), \overline{n})$,
   - $\texttt{Cost}_\texttt{p}(I(\omega_i), \overline{n})$ is computed by a profiler tool, or

- $\texttt{Cost}_\texttt{p}(I(\omega_i), \overline{n})$ is supplied by the user together with the code of program `p` (i.e., the cost function is not the result from any automatic analysis but rather `p` is well known and its cost function can be supplied in a trust assertion).

2. For each benchmark p in this set, automatically generating a significant amount m of input data for it. This can be achieved by associating to each calibration program a data generation rule.
3. For each generated input data d_j, computing a pair $(\overline{C}_{p_j}, T_{p_j})$, $1 \leq j \leq m$, where:
   - T_{p_j} is the j-th observed execution time of program p with this generated input data.
   - $\overline{C}_{p_j} = \overline{\texttt{Cost}_\texttt{p}}(\overline{I(\Omega)}, \overline{n_j})$, where $\overline{n_j}$ is the size of the j-th input data d_j.
4. Using the set of pairs $(\overline{C}_{p_j}, T_{p_j})$ for setting up the equation:

$$\overline{C}_{p_j} \bullet \overline{K}_\Omega = T_{p_j} \tag{10}$$

   where $\overline{K}_\Omega$ is considered a vector of variables.
5. Setting up the (overdetermined) system of equations composed by putting together all the equations (10) corresponding to all the calibration programs.
6. Solving the above system of equations using the least square method. A solution to this system gives concrete values to the vector $\overline{K}_\Omega$ (and hence, to the constants K_{ω_i} which are the elements composing it).
7. Calculating the constants for builtins and arithmetic operators by performing repeated tests in which only the builtin being tested is called, accumulating the time, and dividing the accumulated time by the number of times the repeated test has been performed.

5 Assessment of the Calibration of Constants

We have assessed both the constant calibration process and the prediction of execution times using the previously proposed cost models in two different platforms:

- "intel" platform: Optiplex Dell, Pentium 4 (Hyper threading), 2GHz, 512MB RAM memory, Fedora Core 4 operating System with Kernel 2.6.
- "ppc" platform: Apple iMac, PowerPC G4 (1.1) 1.5GHz, 1GB RAM memory, with Mac OS X 10.4.5 Tiger.

In section 4 we presented equation 10, and we mentioned that it can be solved using the least squares method. We used the householder algorithm, which consists in decomposing the matrix $C = \{\overline{C}_{p_j}\}$, which has m rows and n cols into the product of two matrices Q and U such that $C = QU$, where Q is an orthonormal matrix (i.e. , $Q^T Q = I$, the m x m identity matrix) and U an upper triangular m x n matrix. Then, we can rewrite equation 10 as

$$U \bullet K = Q^T \bullet T = B$$

Program
Environment test
Call test
Recursion not optimized
Verify a list
Input deep ground terms
Output deep ground terms
Input flat ground terms
Output flat ground terms
Build a list of input var terms
Build a list of output var terms
Unify two lists
Head with many arguments

Table 1. Description of calibration programs used in the estimation of constants.

where, for clarity of exposition, we denote $K = \overline{K}_{\Omega}$ and $T = T_{p_j}$. We can take advantage of the structure of U and define V as the first n rows of U, being n the number of columns of C and b the first n rows of B, then K can be estimated solving the following upper triangular system, where $\hat{K}$ stands for the estimate for K:

$$V \bullet \hat{K} = Q^T \bullet T = b$$

Since this method is being used to find an approximate solution, we define the residual of the system as the value

$$R = T - C\hat{K}$$

Let

$$RSS = R \bullet R$$

be the residual square sum, and let

$$MRSS = \frac{RSS}{m - n}$$

be the mean of residual square sum, and finally let

$$S = \sqrt{MRSS}$$

be the estimation of the model standard error, S. In order to experimentally evaluate which models better approximate the observed time in practice, we have compared the values of $MRSS$ (or S) for several proposed models. Table 2 shows the estimated values for the vector K using the calibration programs in Table 1, as well as the standard error of the model, sorted from the best to the worst model. For example, the first row in the table shows the model that has as components step, nargs, giunif, gounif, viunif, vounif for the intel platform. It has a standard error of 6.2475 μs and the values for each of the constants are 21.27, 9.96, 10.30, 8.23, 6.46, and 5.69, respectively.

Note that the estimation of K is done just once per platform. In the case of the intel platform it took 15.62 seconds and in ppc 17.84 seconds, repeating the experiment 250 times by each program.

Plat.	Model	S (μs)	$\overline{K}_\Omega$
intel	step nargs giunif gounif viunif vounif	6.2475	(21.27, 9.96, 10.30, 8.23, 6.46, 5.69)
	step giunif gounif viunif vounif	9.3715	(26.56, 10.81, 8.60, 6.17, 6.39)
	step giunif gounif vounif	13.7277	(27.95, 11.09, 8.77, 7.40)
	step	68.3088	108.90
ppc	step nargs giunif gounif viunif vounif	4.7167	(41.06, 5.21, 16.85, 15.14, 9.58, 9.92)
	step giunif gounif viunif vounif	5.9676	(43.83, 17.12, 15.33, 9.43, 10.29)
	step giunif gounif vounif	16.4511	(45.95, 17.55, 15.59, 11.82)
	step	116.0289	183.83

Table 2. Global values for vector constants in several cost models (in nanoseconds), sorted by S, the standard error of the model.

6 Assessment of the Prediction of Execution Times

We have tested our implementation of the proposed cost models in order to assess how they predict the execution time of other programs (not used in the calibration process) statically, without performing any runtime profiling with them. We have performed experiments with all the 63 possible cost models that result of the combination of one or more of the components described in Section 2.2. However, for space reasons and for clarity, we only show the three most accurate cost models (according to a global comparison that will be presented later) plus the step model, which has special interest as we will also see later. Experimental results are shown in Table 3. **Prog.** lists the program names. The analyzers integrated in the `CiaoPP` system infer the exact cost function for all the programs in that table under the $I(\omega_i)$ metric, which means that the upper and lower bound are the same, i.e. $\mathtt{Cost_p}^l(I(\omega_i), \overline{n}) = \mathtt{Cost_p}^u(I(\omega_i), \overline{n}) = \mathtt{Cost_p}(I(\omega_i), \overline{n})$. There are several rows for each program in the table. The first three rows show results corresponding to the prediction of execution times with the three more accurate cost models. The fourth row shows the prediction obtained by the cost model *step* that only considers resolution steps, i.e., it assumes that the execution time of a procedure call is directly proportional to the number of resolution steps performed by the call. This means that for this simple cost model we are assuming that $time(step) = K_{step}$, since $I(step) = 1$, for a constant K_{step}, which represents the time taken by a resolution step. Note that $\mathtt{Cost_C}(I(step), \overline{n})$ gives the number of resolution steps performed by clause `C`. The last row per benchmark program presents the observed execution times (i.e. measured execution times) and allows measuring the accuracy of the different predictions. In this sense, values in the **Model** column are the names of the four cost models. The value **observed** identifies the row corresponding to the observed values. The following two columns show results corresponding to the "intel" execution platform.

Column **Estimate** shows execution times computed by using the average value of the constant $\overline{K}_\Omega$ as estimated in Table 2:

$$\textbf{Estimate} = \overline{K}_\Omega \bullet \overline{\mathtt{Cost_p}}(\overline{I(\Omega)}, \overline{n})$$

Deviations respect to the observed values (in the **observed** row) are also shown between parenthesis in the column **Estimate**.

Prog.	Model	intel		ppc	
		Estimate	T_{ca}	Estimate	T_{ca}
		(μs) (%)	(s)	(μs) (%)	(s)
evpol	step nargs giunif gounif viunif vounif	89.72 (44)	2.002	77.4 (23)	4.461
	step giunif gounif viunif vounif	85.06 (38)		74.96 (26)	
	step giunif gounif vounif	82 (35)		70.28 (33)	
	step	90.12 (45)		85.07 (13)	
	observed	58.43		97.08	
hanoi	step nargs giunif gounif viunif vounif	319 (31)	2.145	398.5 (4)	4.903
	step giunif gounif viunif vounif	243.3 (3)		358.8 (7)	
	step giunif gounif vounif	205.6 (14)		301.3 (25)	
	step	340.7 (38)		538.6 (34)	
	observed	235.3		384.2	
nrev	step nargs giunif gounif viunif vounif	131.3 (68)	2.022	179.4 (26)	4.691
	step giunif gounif viunif vounif	101.1 (39)		163.6 (16)	
	step giunif gounif vounif	82.51 (18)		135.2 (3)	
	step	144.4 (80)		243.8 (59)	
	observed	69.25		139.2	
palind	step nargs giunif gounif viunif vounif	131.8 (18)	2	179.8 (5)	4.7
	step giunif gounif viunif vounif	101 (9)		163.7 (5)	
	step giunif gounif vounif	86.91 (24)		142.1 (19)	
	step	167.2 (43)		282.2 (52)	
	observed	110		171.6	
powset	step nargs giunif gounif viunif vounif	537.5 (59)	2.07	727.9 (17)	4.636
	step giunif gounif viunif vounif	404.5 (28)		658.3 (7)	
	step giunif gounif vounif	323.8 (5)		534.9 (14)	
	step	448.7 (38)		757.4 (21)	
	observed	308.2		615	
append	step nargs giunif gounif viunif vounif	50.29 (75)	1.932	68.72 (24)	4.441
	step giunif gounif viunif vounif	38.69 (44)		62.65 (15)	
	step giunif gounif vounif	31.36 (22)		51.45 (5)	
	step	54.56 (85)		92.1 (56)	
	observed	25.16		53.92	

Table 3. Experiments on programs considering builtins.

Platform	Model	Error (%)
intel	step giunif gounif vounif	21.48
	step giunif gounif viunif vounif	31.06
	step nargs giunif gounif viunif vounif	53.17
	step	58.45
ppc	step giunif gounif viunif vounif	14.66
	step nargs giunif gounif viunif vounif	18.72
	step giunif gounif vounif	19.44
	step	43.04

Table 4. Global comparative of the accuracy of cost models.

The observed execution times have been measured by running the programs with input data of a fixed size. 10 input data sets of such fixed size have been

generated randomly. 5 runs of the program have been performed for each of such input data sets. The observed execution time for such input size has been computed as the average of all runs.

Column T_{ca} shows the total (static) cost analysis time (in seconds) needed to perform the execution time estimation (and includes mode, type and cost analysis). The following columns show results corresponding to the "ppc" execution platform, and have the same structure than the "intel" platform.

Table 4, compares the overall accuracy of the four cost models already shown in Table 3, for the two considered platforms. The last column shows the global error and it is an indicator of the amount of deviation of the execution times estimated by each cost model with respect to the observed values. As global error we take the square mean of the errors in each example being considered in Table 3. By considering both platforms in combination we can conclude that the more accurate cost model is the one consisting of steps, giunif, gounif, viunif, and vounif. This cost model has an overall error of 14.66 % in platform "ppc" and 31.06 % in "intel". In "intel" (obviously a more challenging platform) the model consisting of steps, giunif, gounif, and vounif appears to be the best. This coincides with our intuition that taking into account a comparatively large number of lower-level operations should improve accuracy. It is also interesting to see that including *nargs* in the cost model does not further improve accuracy, since nargs is not independent from the four components giunif, gounif, viunif, vounif. In fact, including this component results in a less precise model in the "ppc" platform, and in the case of "intel". Also, the cost model step deserves special mention, since it is the simplest one and at least for the given examples, the error is smaller than we expected and better than more complex cost models not shown in the tables.

Overall we believe that the results are very encouraging in the sense that our combined framework predicts with an acceptable degree of accuracy the execution times of programs and paves the way for even more accurate analyses by including additional parameters.

7 Applications

The experimental results presented in Section 6 above show that the proposed framework can be relevant in practice for estimating platform dependent cost metrics such as execution time. We believe that execution time estimates can be very useful in several contexts. As already mentioned, in certain mobile/pervasive computation scenarios different platforms come into play, with each platform having different capabilities. More concretely, the execution time estimates could be useful for performing resource/granularity control in parallel/distributed computing. This belief is based on previous experimental results, where it appeared, from the sensitivity of the results observed in such experiments, that while it is not essential to be absolutely precise in inferring the best time estimates for a problem, the number of reductions by itself was a rough measure and the current time estimation approach could presumably improve on previous results.

One of the good features of our approach is that we can translate platform-independent cost functions (which are the result of the analyzer) into platform-dependent cost functions (using the relationship in expression (9)). A possible

application for taking advantage of this feature is mobile code safety and in particular Proof-Carrying Code (PCC), a general approach in which the code supplier augments the program with a certificate (or proof). Consider a scenario where the producer sends a certificate with a platform-independent cost function (i.e. where the cost is expressed in a platform-independent metric) together with a calibration program. The calibration program includes a fixed set of calibration benchmarks. Then, the consumer runs (only once) the calibration program and computes the values for the constants appearing in the cost functions. Using these constants, the consumer can obtain platform dependent cost functions.

Another application of the proposed approach is resource-oriented specialization. The proposed cost-models, which include low-level factors for CLP programs, are more refined cost-models than previously proposed ones and thus can be used to better guide the specialization process. The inferred cost functions can be used to develop automatic program transformation techniques which take into account the size of the resulting program, its run time and memory usage, and other low-level implementation factors. In particular, they can be used for performing self-tuning specialization in order to compare different specialized version according with their costs [2].

8 Conclusions

We have developed a framework which allows estimating execution times of procedures of a program in a given execution platform. The method proposed combines compile-time (static) cost analysis with a one-time profiling of the platform in order to determine the values of certain constants. These constants calibrate a cost model from which time cost functions for a given platform can be computed statically. The approach has been implemented and integrated in the `CiaoPP` system. To the best of our knowledge, this is the first combined framework for estimating statically and accurately execution time bounds based on static automatic inference of upper and lower bound complexity functions plus experimental adjustment of constants. We have performed an experimental assessment of this implementation for a wide range of different candidate cost models and two execution platforms. The results achieved show that the combined framework predicts the execution times of programs with a reasonable degree of accuracy. We believe this is an encouraging result, since using a one-time profiling for estimating execution times of other, unrelated programs is clearly a challenging goal.

Also, we argue that the work presented in this paper presents an interesting trade-off between accuracy and simplicity of the approach. At the same time, there is clearly room for improving precision by using more refined cost models which take into account additional (lower level) factors. Of course, these models would also be more difficult to handle since on one hand they would require computing more constants and on the other hand they may require taking into account factors which are not observable at source level. This is in any case the subject of possibly interesting future work.

Acknowledgments

This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* project, by the Spanish Ministry of Education under the TIN-2005-09207 *MERIT* project, and the Madrid Regional Government under the *PROMESAS* project. Manuel Hermenegildo is also supported by the Prince of Asturias Chair in Information Science and Technology at UNM.

References

1. B. Brassel, M. Hanus, F. Huch, J. Silva, and G. Vidal. Run-time profiling of functional logic programs. In *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'04)*, pages 182–197. Springer LNCS 3573, 2005.
2. S.J. Craig and M. Leuschel. Self-tuning resource aware specialisation for Prolog. In *Proc. of PPDP'05*, pages 23–34. ACM Press, 2005.
3. S. K. Debray. Profiling prolog programs. *Software Practice and Experience*, 18(9):821–839, 1983.
4. S.K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation*, pages 174–188. ACM Press, June 1990.
5. S.K. Debray and N.W. Lin. Cost analysis of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, November 1993.
6. S.K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Lower Bound Cost Estimation for Logic Programs. In *1997 International Logic Programming Symposium*, pages 291–305. MIT Press, Cambridge, MA, October 1997.
7. Mireille Ducassé. Opium: An extendable trace analyzer for prolog. *J. Log. Program.*, 39(1-3):177–223, 1999.
8. M. Hermenegildo, E. Albert, P. López-García, and G. Puebla. Abstraction Carrying Code and Resource-Awareness. In *Proc. of PPDP'05*. ACM Press, July 2005.
9. M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.
10. P. López-García. *Non-failure Analysis and Granularity Control in Parallel Execution of Logic Programs*. PhD thesis, Universidad Politécnica de Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, June 2000.
11. P. López-García, M. Hermenegildo, and S.K. Debray. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *J. of Symbolic Computation, Special Issue on Parallel Symbolic Computation*, 22:715–734, 1996.
12. S. A. Jarvis R. G. Morgan. Profiling large-scale lazy functional programs. *Journal of Functional Programing*, 8(3):201–237, May 1998.
13. Patrick M. Sansom and Simon L. Peyton Jones. Formally based profiling for higher-order functional languages. *ACM Transactions on Programming Languages and Systems*, 19(2):334–385, March 1997.