Combining Static Analysis and Profiling for Estimating Execution Times

Edison Mera[1], Pedro López-García[1], Germán Puebla[1], Manuel Carro[1], and Manuel V. 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 infer bounds on the number of resolutions. These have the advantage of being independent from the platform on which the programs are executed and have been shown to be useful in a number of applications, such as granularity control in parallel execution. On the other hand, in distributed computation scenarios where platforms with different capabilities come into play, it is necessary 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 times. With this objective in mind, we propose an approach which combines compile-time analysis for cost bounds with a one-time profiling of a given platform in order to determine the values of certain parameters for that platform. These parameters 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 that concrete 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. 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 [11, 12] the method of [5, 11] was fully automated in the context of a practical compiler and in [6, 11] 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, 11, 12]. In a typical scenario, these tasks will be executed in a single parallel machine, where all processors are typically identical. Therefore, the deduced number of reductions can actually be used as a relative measure in order to compare to a first degree of approximation the amount of work under the tasks.

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. This applies also to heterogeneous parallel computing platforms. 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 parametrized 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 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, with the aim of helping to discover why a part of a program does not exhibit the expected performance. Debray [3] showed the basic considerations to have in mind when profiling Prolog programs: handling backtracking and failure. Ducassé [7] designed and implemented a trace analyzer for Prolog which can be applied to profiling. Sansom and Peyton Jones [14] focused on profiling of functional languages using a semantic approach and highlighted the difficulty in profiling such kind of languages. Jarvis and Morgan [13] showed how to profile lazy functional programs. 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. We will use also profiling but, since our aim is to *predict* performance, profiling will in our case be aimed at calibrating the values for some constants that appear in the cost functions, and which will be instrumental to forecast execution times for a given platform and cost model. Therefore we will not use profiling with just some fixed input arguments, but with a set of programs and input arguments which we hope will be representative enough to derive meaningful characteristics of an execution platform.

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 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 and use several platform-dependent parameters. Once these 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 $\overline{n}_i$, $i = 1 \ldots m$, 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, an upper bound on the cost of clause $\mathtt{C}$ (assuming all solutions are required), $\mathtt{Cost}_{\mathtt{C}}(\overline{n})$, 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 term creation for the literals in the body of clauses, 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. In this case, equation (1) simplifies to:

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

However, it should be pointed out that our implementation is not limited to deterministic programs: our cost analysis system indeed handles non determinism, i.e., the presence of several solutions for a given call.

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 the number of solutions generated by a predicate that will be demanded is generally not known in advance, 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). If we take mutual exclusion among clauses into account, we can obtain a more precise estimate of the cost of a predicate: the complexity for deterministic predicates can be approximated by the maximum of the costs of mutually exclusive groups of clauses.

The analysis in [4, 5] was primarily aimed at estimating resolution steps. However, the basic metric is open and can be tailored to alternative scenarios: more sophisticated and accurate measures can be used in place of the initially proposed ones (by, e.g., decomposing arbitrary unifications into simpler steps). 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. These will in turn make it possible to generate expressions which capture execution time (or, typically, a bound thereof) more accurately.

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 include the time needed to resolve a literal `G` against the corresponding clause head `H`, but also the cost associated with selecting alternatives, the cost coming from setting up the body literals for execution, allocating activation records, etc. In the following, we will still refer to τ as the *clause head cost function* (but understanding that it now includes all these costs), and we will consider different definitions 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 generalize τ to be a function parametrized by the cost model so that:

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

$time(\Omega)$ returns the time associated to a resolution step, including the aforementioned additional overheads. The parameter $\Omega = (\omega_1, \ldots, \omega_v)$ is a vector denoting which characteristics we want to take into account: every ω_i looks at

a different indicator of the execution time. The family of cost models we will study assumes that $time(\Omega)$ is defined as follows:

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

where each $time(\omega_i)$ contributes with the part of the execution time which depends on the feature ω_i. We also 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. I.e., K_{ω_i} expresses the cost of each unit of $I(\omega_i)$ in terms of time. Equation (4) can be written in vector notation as

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

where $\overline{K}_{\Omega} = (K_{\omega_1}, \ldots, K_{\omega_v})$ and $\overline{I(\Omega)} = (I(\omega_1), \ldots, I(\omega_v))$ are vectors of platform-dependent constants and of platform-independent cost functions, respectively. Accordingly, we generalize equation (2) by introducing the clause head cost function τ as a parameter:

$$\mathtt{Cost_C}(\Omega, \overline{n}) \leq \tau(\Omega) + \sum_{i=1}^{m} \mathtt{Cost_{L_i}}(\Omega, \overline{n}_i). \tag{7}$$

A cost model, of which we have tested several, is given by a particular definition of the parameter Ω. Every cost model is defined by the program characteristics taken into account by it. While a large number of indicators can be used, we have identified some of them as specially interesting. We list them below, giving a mnemonic to every ω_i and explaining the meaning of each $I(\omega_i)$.

In what follows we will say that an argument of a literal is an *output argument* if the term being passed by the calling literal is known to be a variable at run-time, and an *input argument* if it is not a variable. Run-time arguments can be classified as either input or output using well-known techniques for mode analyses (in our case, those provided by `CiaoPP`).

$I(step) = 1$ Every successful *head traversal* has a constant weight in the execution. I.e., in equation (5), we have:

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

$I(vounif) =$ *the number of variables in the clause head which correspond to "output" argument positions.* This describes a component of the execution time that is directly proportional to the number of cases where both a goal argument and the corresponding head argument are variables. This should boil down to assignment (maybe with trailing).

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

$I(viunif) =$ *the number of variables in the clause head which correspond to "input" argument positions.* This component corresponds to the number of non-variable goal arguments which are unified with a variable in the head. The unification for such arguments is also similar to an assignment with a small, constant cost. We assume that the cost of creating the input argument is constant. Given these assumptions:

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

$I(gounif) =$ *The number of function symbols and constants in the clause head which appear in output arguments.* We are capturing here the size of the terms that are created when a variable in a goal is unified with a non-variable in the clause head.

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

$I(giunif) =$ *The number of function symbols and constants in the clause head which appear in input arguments.* We assume that there is a component of the execution time which depends on the number of arguments in which neither the goal nor the clause head arguments are variables. For each of these arguments, we take into account the number of symbols in the clause head.

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

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

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

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 [4, 6] for computing bounds on cost of predicates from the computed values for the cost of the clauses defining it. However, we introduce the cost model τ as a parameter of these cost functions.

Let $\mathtt{Cost}_{\mathtt{p}}(\Omega, \overline{n})$ be a function which gives the cost of the computation of a call to predicate $\mathtt{p}$ for an input of size $\overline{n}$ (recall that the cost units depend on the definition of Ω). Given a predicate $\mathtt{p}$, and a clause head cost function $time(\Omega)$ as defined in equation (6), we have that:

$$\mathtt{Cost}_{\mathtt{p}}(\Omega, \overline{n}) = \overline{K}_{\Omega} \bullet \overline{\mathtt{Cost}_{\mathtt{p}}}(\Omega, \overline{n}) \tag{9}$$

where

$$\overline{\mathtt{Cost}_{\mathtt{p}}}(\Omega, \overline{n}) = (\mathtt{Cost}_{\mathtt{p}}(I(\omega_1), \overline{n}), \ldots, \mathtt{Cost}_{\mathtt{p}}(I(\omega_v), \overline{n}))$$

Equation (9) gives the basis for computing values for constants K_{ω_i} via profiling (as explained in Section 3). 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.

2.3 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 (external predicates). We will refer to all of them as builtins for brevity. We assume that a cost function is available (expressed via `trust` assertions [9]) for each such predicate. This cost function can be a constant in simple cases but more generally it will be a function that depends on sizes of the (input) arguments of the predicate. As an example, the cost of arithmetic predicates (such as `=:=/2`, `=\=/2`, or `>/2`) is approximated by a function that depends on the size (and types) of the arithmetic expressions that will appear as arguments.

Note that this is a significant change with respect to the cost analysis proposed in [4] since one of the simplifying assumptions made in that analysis was to not count calls to certain builtin as resolution steps (which meant that they were simply ignored in the cost analysis). While such an assumption made sense for inferring *number of resolution steps*, the assumption is not realistic for estimating *execution times*, since the time involved in executing such builtins is not negligible in general and thus has to be taken into account.

We have modeled this by assuming that each builtin contributes with a new component of the cost model to the execution time as expressed in Equation (4). Then, a new $time(\omega_i)$ is added for each builtin predicate `b/n` as follows:

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

We now consider in more detail the case of arithmetic operators and discuss several possibilities. For the sake of accuracy, every arithmetic operator can be dealt with separately: let $\odot/n$ be an arithmetic operator. As usual, the execution time due to the total number of times that this operator is evaluated is given by:

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

where $K_{\odot/n}$ approximates the time taken by the evaluation of the arithmetic operator $\odot/n$. $I(\odot/n)$ could be the number of times that the arithmetic operator is evaluated. With these assumptions, equation (9) (in Section 2.2) also holds for programs that perform calls to builtin predicates, say, for example, a builtin b/n, by introducing b/n and $\odot/n$ as new cost components of Ω.

Alternatively, $I(\odot/n)$ can be a cost function defined as:

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

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.:

$$\texttt{EvCost}(\odot/n, A) = \begin{cases} 0 & \text{if } \texttt{atomic}(A) \vee \texttt{var}(A) \\ 1 + \sum_{i=1}^{n} \texttt{EvCost}(\odot/n, A_i) & \text{if } A = \odot(A_1, ..., A_n) \\ \sum_{i=1}^{m} \texttt{EvCost}(\odot/n, A_i) & \text{if } A = \hat{\odot}(A_1, ..., A_m) \wedge \hat{\odot} \neq \odot \end{cases}$$

For simplicity we can make the assumption 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., to ignore the cost of evaluating t). This can be a good approximation if in most cases t is a number and thus no evaluation of a complex expression is needed for it. This is the case in our simple benchmarks and our experimental results show good time predictions for arithmetic builtin predicates using just the simple cost model. On the other hand, a more refined cost model which assumes that cost is a function on the size of t will be needed for programs which evaluate symbolic arithmetic expressions.

Note that the simple models that we have discussed ignore the possible optimizations that the compiler might perform. We can take into account those performed by source-to-source transformation by placing our analyses in the last stage of the front-end, but at some point the language the compiler works with would be different enough as to require different considerations in the cost model.

3 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 in general cases. 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., both upper an lower bounds are the same: $\texttt{Cost}_\texttt{p}{}^{l}(I(\omega_i), \overline{n}) = \texttt{Cost}_\texttt{p}{}^{u}(I(\omega_i), \overline{n}) = \texttt{Cost}_\texttt{p}(I(\omega_i), \overline{n})$,
   - $\texttt{Cost}_\texttt{p}(I(\omega_i), \overline{n})$ 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 with 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.
   - $\overline{C}_{p_j} = \overline{\texttt{Cost}_\texttt{p}}(\Omega, \overline{n_j})$, where $\overline{n_j}$ is the size of the j-th input data d_j.

4. Using the set of pairs $(\overline{C}_{p_j}, T_{p_j})$ to set 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 resulting from putting together all the equations (10) corresponding to all the calibration programs.
6. Solving the above system of equations using the least squares method (see, e.g., [15]). A solution to this system gives 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.

4 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: Dell Optiplex, 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.

Equation (10) is, in general, overdetermined, and we plan to find an approximation which is "best" in some sense, by using the least squares method. We used the Householder transformation [10], which decomposes the $m \times n$ matrix $C = \{\overline{C}_{p_j}\}$ into the product of two matrices Q and U such that $C = Q \bullet U$, where Q is an orthonormal matrix (i.e., $Q^T \bullet Q = I$, the $m \times m$ identity matrix) and U an upper triangular $m \times n$ matrix. Then, multiplying both sides of equation (10) by Q^T and simplifying we can get:

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

where, for clarity, we denote $K = \overline{K}_{\Omega}$, $T = T_{p_j}$ and $Q^T \bullet T = B$. We can take advantage of the structure of U and define V as the first n rows of U, n being 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, where m and n are the number of rows and columns of the matrix C respectively, and finally let $S = \sqrt{MRSS}$ be the estimation of the standard error of the model, S. In order to evaluate experimentally which

No.	Model
1	step nargs giunif gounif viunif vounif
2	step giunif gounif viunif vounif
3	step giunif gounif vounif
4	step

Table 1. List of cost models being applied.

Plat.	Model	S (μs)	$\overline{K}_\Omega$
Intel	1	6.2475	(21.27, 9.96, 10.30, 8.23, 6.46, 5.69)
	2	9.3715	(26.56, 10.81, 8.60, 6.17, 6.39)
	3	13.7277	(27.95, 11.09, 8.77, 7.40)
	4	68.3088	108.90
PPC	1	4.7167	(41.06, 5.21, 16.85, 15.14, 9.58, 9.92)
	2	5.9676	(43.83, 17.12, 15.33, 9.43, 10.29)
	3	16.4511	(45.95, 17.55, 15.59, 11.82)
	4	116.0289	183.83

Table 2. Values (in nanoseconds) for vector constants in several cost models, sorted by standard error.

models generate the best approximation of the observed time, we have compared the values of $MRSS$ (or S) for several proposed models.

Table 1 shows the considered models. Table 2 shows the estimated values for the vector K using the calibration programs in Table 3, as well as the standard error of the model, sorted from the best to the worst model. Note that the estimation of K only needs to be done once per platform. This took 15.62 seconds for the Intel platform and 17.84 seconds for the PPC, repeating the experiment 250 times for each calibration program. Our approach has been tested on the programs used in the calibration process itself for the considered models. Table 3 shows the error incurred in when an observed value is compared against an estimated value using the models in Table 1. It can be observed that the simpler models incur in significant errors while the more complex ones are more accurate

Program	Error (%)			
Model	1	2	3	4
Environment creation	20	16	12	73
Predicates with no arguments	10	6	2	85
Traverse a list without last call optimization	20	20	11	80
Traverse a list with last call optimization	53	50	32	88
Program (unifying deep terms) for which $I(giunif)$ is known	16	18	18	474
Program (unifying deep terms) for which $I(gounif)$ is known	0	4	2	409
Program (unifying flat terms) for which $I(giunif)$ is known	16	18	18	472
Program (unifying flat terms) for which $I(gounif)$ is known	5	10	8	386
Program for which $I(viunif)$ is known	9	11	36	735
Program for which $I(vounif)$ is known	1	2	11	227
Unify two list element by element	34	29	20	26
Predicate with many arguments	17	16	9	159

Table 3. Calibration programs used to estimate the constants and the estimation error.

(understandable since these calibrators exercise just particular implementation aspects and are thus expected to deviate from any "normal" behaviour).

5 Assessment of the Prediction of Execution Times

We have tested the proposed cost models in a set programs not used in the calibration process in order to assess how well their execution time is predicted, without performing any runtime profiling on them. We have performed experiments with the 63 possible cost models resulting from selecting one or more of the components described in Section 2.2. For space reasons we only show the three most accurate cost models (according to a global accuracy comparison that will be presented later) plus the step model (number 4), which, despite its simplicity, has a special interest, as we will also see later. Experimental results are shown in Table 4, where the analyzers integrated in the `CiaoPP` system infer the exact platform-independent cost function for all the programs in that table, which means that the upper and lower bound are the same, i.e., $\mathtt{Cost_p}^l(I(\omega_i), \overline{n}) = \mathtt{Cost_p}^u(I(\omega_i), \overline{n}) = \mathtt{Cost_p}(I(\omega_i), \overline{n})$. The first three rows for each test program show the three more accurate predictions along with the model used. The fourth row shows the prediction obtained by the cost model *step*, which assumes that the execution time is directly proportional to the number of resolution steps performed. Note that $\mathtt{Cost_C}(I(step), \overline{n})$ gives the number of resolution steps performed by clause `C`. The row tagged as **Observed** corresponds to the actual measured timings, and the last row details the analysis time (roughly the same in all benchmarks, and which includes mode, type, and cost analysis).

The first column is the program name, the second is the cost model Ω (= vector of characteristics taken into account) and the third and fourth are the timing estimations corresponding to the "Intel" and "PPC" platforms. These are computed by using the average value of the constant $\overline{K}_\Omega$ as estimated in Table 2 with the formula:

$$\mathbf{Estimate_P} = \overline{K}_\Omega \bullet \overline{\mathtt{Cost_p}}(\Omega, \overline{n})$$

Deviations respect to the measured values are also shown between parenthesis in the column $\mathbf{Estimate_P}$.

The observed execution times have been measured by running the programs with input data of a fixed size. We generated randomly 10 input data sets of fixed size, and for each data set we run 5 times every program. The observed execution time for the (fixed) input size was computed as the average of all runs.

Table 5 compares the overall accuracy of the four cost models already shown in Table 4, 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 4. By considering both platforms in combination we can conclude that the more accurate cost model is $\Omega = (steps, giunif, gounif, viunif, vounif)$. This cost model has an overall error of 14.66 % in the PPC platform and 31.06 % in

Program	Model		**Estimate$_P$**	
			Intel	PPC
			(μs) (%)	(μs) (%)
evpol	1	step nargs giunif gounif viunif vounif	89.72 (44)	77.4 (23)
	2	step giunif gounif viunif vounif	85.06 (38)	74.96 (26)
	3	step giunif gounif vounif	82 (35)	70.28 (33)
	4	step	90.12 (45)	85.07 (13)
	Observed		58.43	97.08
	Analysis time T_{ca} (s)		2.002	4.461
hanoi	1	step nargs giunif gounif viunif vounif	319 (31)	398.5 (4)
	2	step giunif gounif viunif vounif	243.3 (3)	358.8 (7)
	3	step giunif gounif vounif	205.6 (14)	301.3 (25)
	4	step	340.7 (38)	538.6 (34)
	Observed		235.3	384.2
	Analysis time T_{ca} (s)		2.145	4.903
nrev	1	step nargs giunif gounif viunif vounif	131.3 (68)	179.4 (26)
	2	step giunif gounif viunif vounif	101.1 (39)	163.6 (16)
	3	step giunif gounif vounif	82.51 (18)	135.2 (3)
	4	step	144.4 (80)	243.8 (59)
	Observed		69.25	139.2
	Analysis time T_{ca} (s)		2.022	4.691
palind	1	step nargs giunif gounif viunif vounif	131.8 (18)	179.8 (5)
	2	step giunif gounif viunif vounif	101 (9)	163.7 (5)
	3	step giunif gounif vounif	86.91 (24)	142.1 (19)
	4	step	167.2 (43)	282.2 (52)
	Observed		110	171.6
	Analysis time T_{ca} (s)		2	4.7
powset	1	step nargs giunif gounif viunif vounif	537.5 (59)	727.9 (17)
	2	step giunif gounif viunif vounif	404.5 (28)	658.3 (7)
	3	step giunif gounif vounif	323.8 (5)	534.9 (14)
	4	step	448.7 (38)	757.4 (21)
	Observed		308.2	615
	Analysis time T_{ca} (s)		2.07	4.636
append	1	step nargs giunif gounif viunif vounif	50.29 (75)	68.72 (24)
	2	step giunif gounif viunif vounif	38.69 (44)	62.65 (15)
	3	step giunif gounif vounif	31.36 (22)	51.45 (5)
	4	step	54.56 (85)	92.1 (56)
	Observed		25.16	53.92
	Analysis time T_{ca} (s)		1.932	4.441

Table 4. Evaluation of execution time predictions.

Platform	Intel				PPC			
Model	1	2	3	4	1	2	3	4
Error (%)	53.17	31.06	21.48	58.45	18.72	14.66	19.44	43.04

Table 5. Global comparison of the accuracy of cost models.

the Intel platform. In the latter (obviously more challenging) architecture the model $\Omega = (steps, giunif, gounif, vounif)$ appears to be the best.

This is in line with the intuition that taking into account a comparatively large number of lower-level operations should improve accuracy. However, such components should contribute significantly to the model in order to avoid noise introduction. It is also interesting to see that including *nargs* in the cost model does not further improve accuracy, as expected, 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 both platforms, due to the noise introduced in the model. 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.

The disparity in the accuracy for both platforms can be attributed to a number of reasons, among them the difference in the internal architectures (number of registers, orthogonality in their usage, etc.), which make predicting execution characteristics in Intel processors harder. The weight of some constants can also differ from the calibration programs to the benchmarks due to, e.g., the state of the internal processor pipelines and state of registers. In our experience, the PPC architecture offers a more homogeneous behavior performance-wise.

Overall we believe that the results are 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.

6 Applications

The experimental results presented in Section 5 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 query, the number of reductions by itself was too rough a 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 [8].

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 to their costs [2].

The use of a source-level characterization of the execution profile, which undoubtedly carries some lack of accuracy with it, can be applied not only to different architectures, but also to different compilation / execution schemes. By identifying a rich enough cost model, and using the calibration programs under a given execution model (and architecture), predictions about this execution model / architecture can be made. The advantage lies in that instrumenting the low-level representation used by the execution algorithm (e.g., WAM code & emulator, C code / assembler, or interpreters or virtual machines for other bytecode representations) is not needed: $\overline{K}_\Omega$ should get instantiated to the cost (or an approximation thereof) of every identified *basic* feature in the execution model under study.

7 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.

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 and N. W. Lin. Cost analysis of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, November 1993.
5. 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.
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. Alston S. Householder. Unitary Triangularization of a Nonsymmetric Matrix. *Journal ACM*, 5(4):339–342, October 1958. DOI:10.1145/320941.320947.
11. 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.
12. 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.
13. S. A. Jarvis R. G. Morgan. Profiling large-scale lazy functional programs. *Journal of Functional Programing*, 8(3):201–237, May 1998.
14. 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.
15. D. Wackerly, W. Mendenhall, and R. Scheaffer. *Mathematical Statistics With Applications 5th Edition.* P W S Publishers, 1995.