Towards Execution Time Estimation in Abstract Machine-Based Languages *

E. Mera [1]
[1] Complutense University of Madrid
edison@fdi.ucm.es

P. Lopez-Garcia[2,3]
[2] IMDEA-Software
[3] CSIC
pedro.lopez.garcia@imdea.org

M. Carro[4], M. Hermenegildo[2,4,5]
[4] Technical U. of Madrid
[5] U. of New Mexico
{mcarro, herme}@fi.upm.es

Abstract

Abstract machines provide a certain separation between platform-dependent and platform-independent concerns in compilation. Many of the differences between architectures are encapsulated in the specific abstract machine implementation and the bytecode is left largely architecture independent. Taking advantage of this fact, we present a framework for estimating upper and lower bounds on the execution times of logic programs running on a bytecode-based abstract machine. Our approach includes a one-time, program-independent profiling stage which calculates constants or functions bounding the execution time of each abstract machine instruction. Then, a compile-time cost estimation phase, using the instruction timing information, infers expressions giving platform-dependent upper and lower bounds on actual execution time as functions of input data sizes for each program. Working at the abstract machine level makes it possible to take into account low-level issues in new architectures and platforms by just reexecuting the calibration stage instead of having to tailor the analysis for each architecture and platform. Applications of such predicted execution times include debugging/verification of time properties, certification of time properties in mobile code, granularity control in parallel/distributed computing, and resource-oriented specialization.

Categories and Subject Descriptors D.4.8 [*Performance*]: Modeling and prediction;
F.3.2 [*Semantics of Programming Languages*]: Program analysis;
D.1.6 [*Programming Techniques*]: Logic programming

General Terms Languages, performance

Keywords Execution Time Estimation, Cost Analysis, Profiling, Resource Awareness, Cost Models, Logic Programming.

* The authors have been partially supported by EU projects 215483 *S-Cube*, IST-15905 *MOBIUS*, Spanish projects ITEA2/PROFIT FIT-340005-2007-14 *ES_PASS*, ITEA/PROFIT FIT-350400-2006-44 *GGCC*, MEC TIN2005-09207-C03-01 *MERIT/COMVERS*, Comunidad de Madrid project S-0505/TIC/0407 *PROMESAS*. Manuel Hermenegildo is also partially funded by the Prince of Asturias Chair in Information Science and Technology.

PPDP'08, July 15–17, 2008, Valencia, Spain.

1. Introduction

Cost analysis has been studied for several declarative languages (7; 16; 11; 13). In logic programming previous work has focused on inferring upper (12; 11) or lower (13; 8) bounds on the cost of programs, where such bounds are *functions on the size (or values) of input data*. This approach captures well the fact that program execution cost in general depends on input data sizes. On the other hand the results of these analyses are given in terms of *execution steps*. While this measure has the advantage of being platform independent, it is not straightforward to translate such steps into execution time.

Estimation of *worst case execution times* (WCET) has received significant attention in the context of high-level imperative programming languages (24). In (18; 6) a portable WCET analysis for Java is proposed. However, the WCET approach only provides absolute upper bounds on execution time (i.e., bounds that do not depend on program input arguments) and often requires annotating loops manually.

Our objective is to infer automatically more precise bounds on execution times that are in general functions that depend on input data sizes. In (19) a static analysis was proposed in order to infer such platform-dependent time bounds in logic programs. This approach is based on a high-level analysis of certain syntactic characteristics of the program clause text (sizes of terms in heads, sizes of terms in bodies, number of arguments, etc.). Although promising experimental results were obtained, the predicted execution times were not very precise. In this paper we propose a new analysis which, in order to improve the accuracy of the time predictions, on one hand takes into account lower level factors and on the other makes the model richer by directly taking into account the inherently variable cost of certain low-level operations.

Regarding the choice of this lower level, rather than trying for example to model directly the characteristics of the physical processor, as in WCET, and given that most popular logic programming implementations are based on variations of the Warren abstract machine (WAM) (23; 1), we chose to model cost at the level of abstract machine instructions. Abstract machines have been used as a basic implementation technique in several programming paradigms (functional, logic, imperative, and object-oriented) (14) with the advantage that they provide an intermediate layer that separates to a certain extent the many low-level details of real (hardware) machines from the higher-level language, while at the same time making compilation easier. This property can be used to facilitate the design of our framework.

Within this setting, we present a new framework for the static estimation of execution times of programs. The basic ideas in our approach follow:

1. Measure the execution time of each of the instructions in a lower-level L_B (bytecode) language (or approximate it with a function if it depends on the value of an argument) in some specific abstract machine implementation when executed on a given processor / O.S.
2. Make the information regarding instruction execution time available to the timing analyzer. This is, in our proposal, done by means of *cost assertions* (written in a suitable assertion language) which are stored in a module accessible to the compiler/analyzer.
3. Given a concrete program P written in the source language L_H, compile it into L_B and record the relationship between P and its compiled counterpart.
4. Automatically analyze program P, taking into account the instruction execution time (determined in item 1 above) to infer a cost function C_P. This function is an expression which returns (bounds on) the actual execution time of P for different input data sizes for the given platform.

Points (1) and (2) are performed in a one-time profiling phase, independent from program P, while the rest are performed once for each P in the static (compile-time) cost analysis phase. We would like to point out that, in general, the basic ideas underlying our work can be applied to any language L_H as long as (i) cost estimation can be derived for programs written in L_H, (ii) the translation of L_H to some other (usually lower-level) language L_B is accessible, and (iii) the execution time of the instructions in L_B can be timed accurately enough. We will, however, focus herein on logic languages, so that we assume L_H to be a Prolog-like language and L_B some variant of the WAM bytecode.

The proposed framework has been implemented as part of the CiaoPP (17) system in such a way that any abstract machine properly instrumented can be analyzed. To the best of our knowledge, this is the first attempt at providing a timing analysis producing upper- and lower-bound time *functions* based on the cost of lower-level machine instructions.

2. Mappings Between Program Segments and Bytecodes

Let $OpSet = \{b_1, b_2, \ldots, b_n\}$ be the set of instructions of the abstract machine under consideration. We assume that each instruction is defined by a numeric identifier and its arity, i.e., $b_i \equiv f_i/n_i$, where f_i is the identifier and n_i the arity. Each program is compiled into a sequence of expressions of the form $f(a_1, a_2, \ldots, a_n)$ where f is the instruction name and the a_i's are its arguments. For conciseness, we will use I_i to refer to such expressions. The sequences of expressions into which a program is compiled are generally encoded using bytecodes. In the following we will often refer to sequences of abstract machine instructions or sequences of bytecodes simply as "bytecodes."

Let C be a clause H :- $L_1, \ldots, L_m$. Let $E(C)$ be a function that returns the sequence of bytecodes resulting from the compilation of clause C:

$$E(C) =< I_1, I_2, \ldots, I_p >$$

Let $E(C, H)$ be a function that maps the clause head H to the sequence of bytecodes in $E(C)$ starting from the beginning up to the first `call/execute` instruction or to the end of the sequence $E(C)$ if there are no more `call/execute` instructions (i.e., to the end of the bytecode sequence resulting from the compilation of clause C). Let $E(C, L_i)$ be the function that maps literal L_i of clause C to the sequence of bytecodes in $E(C)$ which start at the `call` bytecode instruction corresponding to this literal and up to the next `call/execute` instruction or to the end of the sequence $E(C)$ if there are no more `call/execute` instructions. If $\uplus$ represents the concatenation of sequences of bytecodes, then:

$$E(C) = E(C, H) \biguplus (\biguplus_{i=1}^{m} E(C, L_i))$$

Note that functions $E(C, H)$ and $E(C, L_i)$ do not necessarily return the bytecodes that one would normally associate to the clause head H and literal L_i respectively. Instead, the definition of those functions associates the instructions corresponding to argument preparation for a given call with the (success of the) *previous* call (or head). This is to cater for the fact that, in the context of backtracking, the WAM argument preparation occurs only one time per call to a literal, even if such call is retried more times before failing definitively. As a result, the cost of argument preparation for a given `call/execute` instruction needs to be associated with the previous literal to that `call/execute`, in order not to count it every time the call is retried.

Table 1 shows how predicate append/3 is compiled into bytecodes, and identifies the result of calling the $E(C, H)$ and $E(C, L_i)$ functions for each clause head and body literal. H^1 represents the head of the first clause (C_1), and H^2 and L_1^2 the head of the second (recursive) clause (C_2) and the first literal in such clause body (the only body literal).

$E(C_1, H^1)$	`append([], X, X).`	
	append/3/1:	try_me_else append/3/2 allocate get_constant([],A0) get_variable(V0,A1) get_value(V0,A2) deallocate proceed
$E(C_2, H^2)$	`append([X|Xs], Y, [X|Zs]) :-`	
	append/3/2:	trust_me allocate get_variable(V0,A0) set_variable(V1) set_variable(V2) set_variable(V3) get_list(V1,V3) set_variable(V4) unify_variable(V2,V4) unify_variable(V0,V3) set_variable(V5,A1) get_variable(V6,A2) set_variable(V7) set_variable(V8) get_list(V1,V8) set_variable(V9) unify_variable(V7,V9) unify_variable(V6,V8) put_value(V2,A0) put_value(V5,A1) put_value(V7,A2) deallocate
$E(C_2, L_1^2)$	`append(Xs, Y, Zs).`	
		execute append/3

Table 1. Sequences of bytecodes assigned to clause heads and body literals of the clauses C_1 and C_2 of predicate append by the functions $E(C, H)$ and $E(C, L)$.

3. Modeling the Execution Time of Instructions

We define a function $t(I)$ (the *timing model*), which takes a bytecode instruction I and returns another function which estimates the execution time for it depending on the input data sizes of the bytecode. This is similar to the approach described in (5), where, however, $t(I)$ was a constant.

In many cases we can assume that the time to execute a bytecode is constant. However there are some instructions for which this does not hold because their definitions involve loops. In many of these cases the timing model consists of an initial constant time t_0 plus another additional constant time t_{iter} to cater for the cost of each iteration, and a simple linear model can be used: $t_0 + n \times t_{iter}$. Consider for example the `unify_void` n instruction, which pushes n new unbound cells on the heap (1), and whose execution time is a linear function on n. In some other cases instructions have different execution times depending on the (fixed) values a given argument can take from some finite set. In such cases, execution time is an arbitrary function on the argument. Specific constants are assigned for each possible argument value by means of profiling (Section 5).

Since the cost of a given instruction is different when it succeeds and when it fails, we will have two costs for each instruction that can fail: one for the success case and another for the failure case. Finally, and besides lower-level factors such as cache behavior, there are some additional variable factors (such as, e.g., the length of dereferencing chains) which may affect execution times. These factors are in principle not impossible to cater for via a combination of static and dynamic analysis, but, given the additional complication involved, we will ignore them herein and explore what kind of precision of timing prediction can be achieved with this first level of approximation.

Another factor that we are not taking into account at this moment is garbage collection (GC). GC makes programs run slower, which, at profiling time, increases the (estimated) cost of every instruction. Therefore, turning it off at profile time (which gives a smaller estimation of instruction cost) is safe when finding out lower bounds: if the program whose execution time is to be predicted is run with GC turned on, then it would run slower w.r.t. an execution with GC turned off (as it was when profiling), and the estimated bounds will still be lower bounds, albeit more conservative. An inverse reasoning applies to upper bounds, and the technique herein presented is equally valid. However, for the sake of simplicity, we have taken all the measurements (both for profiling and executions to be predicted) with GC disconnected.

4. Static Cost Analysis

We now present the compile-time component of our combined framework: the static cost analysis. This analysis has been implemented and integrated in `CiaoPP` (17).

4.1 Overview of the Approach

Since the work done by a call to a recursive procedure often depends on the "size" of its input, knowing this size is a prerequisite to statically estimate such work. Our basic approach is as follows: given a call p, an expression $\Phi_p(n)$ is *statically* computed that (i) is relatively simple to evaluate, and (ii) it approximates $\mathtt{Time}_p(n)$, where $\mathtt{Time}_p(n)$ denotes the cost (in time units) of computing p for an input of size n on a given platform. Various measures are used for the "size" of an input, such as list-length, term-size, term-depth, integer-value, etc. It is then evaluated at run-time, when the size of the input is known, yielding (upper or lower) bounds on the execution time required by the computation of the call on a given platform. In the following we will refer to the compile-time computed expressions $\Phi_p(n)$ as *cost functions*.

Certain program information (such as, for example, input/output modes and size metrics for predicate arguments) is first automatically inferred by other analyzers which are part of `CiaoPP` and then provided as input to the size and cost analysis. The techniques involved in inferring this information are beyond the scope of this paper —see, e.g., (17) and its references for some examples. Based on this information, our analysis first finds bounds on the size of input arguments to the calls in the body of the predicate being analyzed, relative to the sizes of the input arguments to this predicate, using the inferred metrics. The size of an output argument in a predicate call depends in general on the size of the input arguments in that call. For this reason, for each output argument we infer an expression which yields its size as a function of the input data sizes. To this end, and using the input-output argument information, data dependency graphs (namely the *argument dependency graph* and the *literal dependency graph*) are used to set up difference equations whose solution yields size relationships between input and output arguments of predicate calls. The *argument dependency graph* is a directed acyclic graph used to represent the data dependency between argument positions in a clause body (and between them and those in the clause head). The *literal dependency graph* is constructed from the *argument dependency graph* (grouping nodes) and represents the data dependencies between literals.

The information regarding argument sizes is then used to set up another set of difference equations whose solution provides bound functions on predicate calls (execution time). Both the size and cost difference equations must be solved by a difference equation solver. Although the operation of such solvers is beyond the scope of the paper, our implementation does provide a table-based solver which covers a reasonable set of difference equations such as first-order and higher-order linear difference equations in one variable with constant and polynomial coefficients,[1] divide and conquer difference equations, etc. In addition, the system allows the use of external solvers (such as, e.g., Purrs (4), Mathematica, Matlab, etc.) and is currently being extended to interface with other interesting solvers that have been recently developed (2). Note also that, since we are computing upper/lower bounds, it suffices to compute upper/lower bounds on the solution of a set of difference equations, rather than an exact solution. This allows obtaining an *approximate* closed form when the exact solution is not possible.

4.2 Estimating the Execution Time of Clauses and Predicates

Our cost analysis approach is based on that developed in (12; 11) (for estimation of upper bounds on resolution steps) and further extended in (13) (for lower bounds). More recently, in (19) the analysis was extended to work with *vectors* of cost components, with each component considering a known aspect that affects the total cost of the program. In these approaches the cost of a clause can be bounded by the cost of head unification together with the cost 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 (13) for details on lower-bounds cost analysis.

Consider a predicate defined by r clauses $\mathtt{C}_1, \ldots, \mathtt{C}_r$. We take into account that a given clause $\mathtt{C}_k$ will be tried only if clauses $\mathtt{C}_1, \ldots, \mathtt{C}_{k-1}$ fail to yield a solution. Consider clause $\mathtt{C}_k$ defined as $\mathtt{H}^k$:- $\mathtt{L}^k_1, \ldots, \mathtt{L}^k_m$. Because of backtracking, the number of times a literal will be executed depends on the number of solutions of the previous literals. Assume that $\overline{n}$ is a vector such that each element corresponds to the size of an input argument to clause $\mathtt{C}_k$ 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 $\mathtt{L}^k_i$. Assume also that $\tau(\mathtt{H}^k, \overline{\mathtt{n}})$ is the execution time needed to resolve the head $\mathtt{H}^k$ of

[1] Note that it is always possible to reduce a system of linear difference equations to a single linear difference equation in one variable.

the clause $\mathtt{C}_k$ with the literal being solved, $\mathtt{Sols}_{\mathtt{L}_j^k}$ is the number of solutions literal $\mathtt{L}_j^k$ can generate, and $\beta(\mathtt{L}_i^k, \overline{n}_i)$ the time needed to prepare the call to literal $\mathtt{L}_i^k$ in the body of the clause $\mathtt{C}_k$. Because of space constraints, we refer the reader to (11; 13) for details about the algorithms used to estimate the number of solutions that a literal can generate, and the sizes of input arguments. Then, an upper bound $\mathtt{Cost}_{\mathtt{C}^k}(\overline{n})$ on the cost of clause $\mathtt{C}^k$ (assuming all solutions are required) can be expressed as:

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

Here we use $j \prec i$ to denote that $\mathtt{L}_j^k$ precedes $\mathtt{L}_i^k$ in the literal dependency graph for the clause $\mathtt{C}_k$ (described in Section 4.1). We have that:

$$\tau(\mathtt{H}^k, \overline{n}) = \delta_k(\overline{n}) + \sum_{I \in E(\mathtt{C}^k, \mathtt{H}^k)} t(I)(\overline{n})$$

where $\delta_k(\overline{n})$ denotes the execution time necessary to determine that clauses $\mathtt{C}_1, \ldots, \mathtt{C}_{k-1}$ will not yield a solution and that $\mathtt{C}_k$ must be tried: the function δ_k obviously takes into account the type and cost of the indexing scheme being used in the underlying implementation. Also:

$$\beta(\mathtt{L}_i^k, \overline{n}_i) = \sum_{I \in E(\mathtt{C}, \mathtt{L}_i^k)} t(I)(\overline{n}_i), i = 1, \cdots, m$$

with $E(\mathtt{C}, \mathtt{L}_i^k)$ and $t(I)$ defined as in Sections 2 and 3 respectively.

A difference equation is set up for each recursive clause, whose solution (using as boundary conditions the execution times of non-recursive clauses) is a function that yields the execution time of a clause. The execution time of a predicate is then computed from the execution time of its defining clauses. Since the number of solutions which will be required from a predicate is generally not known in advance, a conservative upper bound on the execution time of a predicate can be obtained by assuming that all solutions are needed, and, thus, all clauses are executed and the execution time of the predicate will be the sum of the execution times of its defining clauses. When the clauses of a predicate are mutually exclusive, a more precise estimation of the execution time of such a deterministic predicate can be obtained as the maximum of the execution times of the clauses it is composed of.

Note that our approach allows defining via assertions the execution time of external predicates, which can then be used for modular composition. This includes also predicates for which the code is not available or which are even written in a programming language that is not supported by the analyzer. In addition, assertions also allow describing by hand the execution time of any predicate for which the automatic analysis infers a value that is not accurate enough, and this can be used to prevent inaccuracies in the automatic inference from propagating. The description of the assertion language used is out of the scope of this paper, and we refer the reader to (21) for details.

5. Estimating Instruction Execution Times via Profiling

In this section we will see how data regarding the expected execution time of each instruction in the abstract machine (Section 3) can be accurately measured in a realistic environment.

5.1 Instruction Profiling

Profiling aims at calculating a function $t(I)$ for each bytecode instruction I. An approach is to instrument the WAM implementation so that time measures are taken and recorded at appropriate points in the execution (18). In practice, a number of issues have to be taken into account in order to obtain accurate enough measurements. These include the selection of the places where the instrumentation code will be inserted, how to minimize the effects of such instrumentation on the execution (not only execution time but also, e.g., cache behavior), and how to work around the complex instruction scheduling performed by modern processors, which may lead to large variance in the results, especially since we aim at measuring very small fragments of code.

```
while (op != END) { /* WAM emulation loop */
  ...
   record_profile_info (op); /* op is the current bytecode */

  switch(op) {
     ...
  }
  ...
  op = get_next_op();
}
```

Figure 1. A simple WAM emulation loop instrumented.

A first approximation is to add profiling-related calls in designated parts of the bytecode interpreter main loop. Figure 1 shows a piece of code illustrating this. The `record_profile_info(op)` operation records the start time for the bytecode `op`. The end time is processed when the next opcode is fetched. The data for each bytecode is maintained in memory during execution (and in raw form in order to impact execution as little as possible) and later saved to an external file.

A benchmark-based analysis is also proposed in (18), which describes how the instrumented code can be reused effectively on various platforms without modifying it, and how the execution time of a specific set of bytecodes can be measured.

However, the methods mentioned above have drawbacks. For example, the first one (instrumenting the main loop) depends on the existence of very precise, non-intrusive, low-overhead timing operations which, unfortunately, are not always available in all platforms. Portable O.S. calls, besides having a typically high associated overhead, are in general not accurate enough for our purposes. Even if a very fast timing operation is available (which is not the case in platforms such as mobile and embedded devices), its introduction may affect the behavior of the machine being analyzed if the abstract machine loop is very optimized. For example, if the new code changes register and variable allocation, program behavior will be affected in unforeseen ways.

We will, however, use an instrumented loop like that of Figure 1 to count the number of bytecodes executed in a calibration step.

5.2 Measuring Time Accurately

In order to do portable time measurements in platforms where high resolution timing is difficult or impossible to achieve, workarounds have to be used. The approach that we have followed is based on using synthetic benchmarks which on purpose repeatedly execute the instructions under estimation for a large enough time, and later divide the total execution time by the number of times the instructions were executed. A complication in this process is that it is in general not possible to run a single instruction repeatedly within the abstract machine, since the resulting sequence would not be legal and may "break" the abstract machine, run out of memory, etc. In general, more complex sequences of instructions must be constructed and repeated instead.

Therefore, the approach we have followed involves designing a set of *legal* programs which cover all the bytecode instructions,

Programs	Instructions	Trace
c1_5 :- c1_5_0.	00 : `execute 01`	00 : `execute` 01
c1_5_0 :- c1_5_1.	01 : `execute 02`	01 : `execute` 02
c1_5_1 :- c1_5_2.	02 : `execute 03`	02 : `execute` 03
c1_5_2 :- c1_5_3.	03 : `execute 04`	03 : `execute` 04
c1_5_3 :- c1_5_4.	04 : `execute 05`	04 : `execute` 05
c1_5_4 :- c1_5_5.	05 : `execute 06`	05 : `execute` 06
c1_5_5.	06 : `proceed`	06 : `proceed`
c1_0 :- c1_0_0.	01 : `execute` 02	01 : `execute` 02
c1_0_0.	02 : `proceed`	02 : `proceed`

Table 2. Programs used in order to get the execution time of the `execute` instruction.

relate the execution time of these programs with the individual instruction execution times with a system of equations, and solving such a system.

5.3 Getting Instruction Execution Time

We now discuss how to set up calibration programs in order to get the cost of bytecodes. In this section, and in order to simplify the discussion, we deal with those bytecodes whose execution time is bound by a constant. In the following section we extend our technique to manage instructions whose execution time is unbound.

Let $C_i, i = 0, 1, \ldots, n$ be a set of synthetic calibration programs, each of them returning the execution time of a block of code. Each C_i, which we will refer to as calibrator, is generated in such a way that it repeats such block a given number of times, say r. Let us assume, for example, that we want to calibrate the WAM instruction "`execute`" when it does not fail and that we want to repeat its execution 5 times (i.e., $r = 5$). Table 2 shows a set of programs which can be used to calibrate this WAM instruction. Columns **Instructions** and **Trace** show the WAM code as generated by the compiler and the sequence of instructions executed when running the program starting from the first clause respectively. In general, in our approach, rather than a concrete program, calibrators are program generation templates which take r as an input and return, e.g., the programs in Table 2 for that value of r. The actual calibration program includes an entry point which calls the programs in Table 2 and returns the value of the execution time of the `execute` instruction, subtracting the time spent in the entry calls (e.g., `c1_5` for Table 2). In this case the calibration time is easy to compute as the difference between the execution time of `c1_5` and `c1_0` divided by r. The result of the calibration should ideally be invariant with respect to r; in practice this is however not true due, among other factors, to timing imprecision. Thus, r needs to be determined for each case: it has to be a large enough value to ensure stability of the time measured by the calibrator for the particular platform and the method used to measure time, but not excessively large, as this would make calibration impractical.

In some cases we cannot isolate the behavior of only one bytecode. This is specially the case in the calibrators of instructions which alter the program flow, such as `call`, `proceed`, `trust_me`, `try_me_else`, `retry_me_else`, `allocate`, `deallocate`. It is also the case when measuring the cost of failure for any of the instructions which can fail (generally the `get_` and `unify_` instructions). All these instructions need to be always executed together with other bytecodes in order to make the calibration program legal. As a result, and due to interactions between the costs of the different instructions, the equations are not as easy to configure in all cases as the simple case for the `execute` instruction above.

As a simple example, the calibrator that returns the cost of `call` and the `proceed` instructions uses the programs in Table 3 (where we have turned off the optimization of register / variable allocation in the compiler for simplicity). In order to be able to separate the

Programs	Instructions	Trace
c2_5 :-	00 : `allocate`	00 : `allocate`
c_5,	01 : `call 09`	01 : `call 09`
c_5,	02 : `call 09`	09 : `proceed`
c_5,	03 : `call 09`	02 : `call 09`
c_5,	04 : `call 09`	09 : `proceed`
c_5,	05 : `call 09`	03 : `call 09`
c_5,	06 : `call 09`	09 : `proceed`
c_5.	07 : `deallocate`	04 : `call 09`
	08 : `execute 09`	09 : `proceed`
		05 : `call 09`
c_5.	09 : `proceed`	09 : `proceed`
		06 : `call 09`
		09 : `proceed`
		07 : `deallocate`
		08 : `execute 09`
		09 : `proceed`
c2_0 :-	00 : `allocate`	00 : `allocate`
c_0,	01 : `call 04`	01 : `call 04`
c_0.	02 : `deallocate`	04 : `proceed`
	03 : `execute 04`	02 : `deallocate`
		03 : `execute` 04
c_0.	04 : `proceed`	04 : `proceed`

Table 3. Programs used to get the execution time of the `call` and `proceed` instructions.

cost of `call` and `proceed` an idea might be to find a calibrator that isolates the cost of `proceed` by itself and subtract from the value given by the calibrator for `call` and `proceed` and obtain the cost of `call`. However, that is in general not possible since in all legal calibrators `proceed` and `call` must always appear combined with other bytecodes. In general we need to set up a system of equations in which the known values are the costs given by our calibrators and the unknown values are the costs of the individual bytecodes. Such equations can be configured automatically, by executing the calibration programs in a special version of the WAM with the bytecode dispatch loop instrumented as in Figure 1 so that the profiler keeps an account of the executed bytecodes.

Let c_i, $0 \leq i \leq n$, be the time calibrator C_i has returned, and let β_j, $0 \leq j \leq m$, $m \geq n$, be the cost of a bytecode B_j, distinguishing between the case of a fail or a success in the execution of such bytecode. In other words, $B_j \in I \times \{fail, success\}$, where I the set of all possible bytecodes and *fail* and *success* represent the failure or success of the execution of a bytecode. We can then set up the following system of equations:

$$\begin{array}{rcl} c_1 & = & a_{11}\beta_1 + a_{12}\beta_2 + \cdots + a_{1m}\beta_m \\ c_2 & = & a_{21}\beta_1 + a_{22}\beta_2 + \cdots + a_{2m}\beta_m \\ & \cdots & \\ c_n & = & a_{n1}\beta_1 + a_{n2}\beta_2 + \cdots + a_{nm}\beta_m \end{array} \quad (1)$$

which we can rewrite such using matrix notation:

$$C = AB \quad (2)$$

where $B = (\beta_i)$ is the vector of execution times for the bytecodes. In order to obtain B we ideally need to configure as many calibrators as bytecodes. Finding a solution to this system of equations requires, in principle, independence among the equations (i.e., there is no other linear independent equation but those in (1)), and to have as many equations as variables. However, that is not always possible due to dependencies between the number of times a bytecode is executed. For example, in the WAM under analysis, the following invariant holds:

PROPOSITION 1. *For any program, the number of times* `retry_me_else` *is called plus the number of times* `trust_me` *is called is equal to the number of failures.*

This holds since a failure always causes backtracking to the next choice point, which always implies executing either a `retry_me_else` or a `trust_me` instruction. As the coefficients a_{ij} in the equation above are precisely the number of times every bytecode is executed, it turns out that, for a given execution, some coefficients are dependent on some other coefficients, therefore breaking the initial independence assumption: the system of equations is underdetermined and it does not have a unique solution.

For this reason, since the coefficients a_{ij} where obtained by summarizing legal programs (i.e., the calibrators), and they will be affected by the linear dependency mentioned above, the undetermined system (2) will not have a unique solution. However, note that when several bytecodes in a block must be executed together (because of constraints in the WAM compilation and execution scheme) knowing the execution time of each of them in isolation is not needed: knowing the total execution time of the whole block is enough. This intuitive idea can be formalized and generalized with the following result:

PROPOSITION 2. *Given a set of n calibration programs C_i, that define n linear independent equations with β_i variables (corresponding to the m bytecodes, with both success and failure cases included), if we have that for all programs there exist $m-n$ linear independent relationships between the number of bytecodes that are always fulfilled, then the estimated execution time is invariant with respect to the choice of any arbitrary element of the solution set of such linear system.*

Proof : Let B be an arbitrary solution of $C = AB$. Let X be a vector which represents the number of times each bytecode has been executed for a given program. The estimated execution time is $E = X^T B$, i.e., the sum of the time for each bytecode multiplied by the number of times it has been executed.

By linear algebra, and considering that each calibrator defines a linear independent equation, we have that the range of A is n, and the kernel (or nullspace) of A is given by the set of all λ such that $A\lambda = 0$, a vector space of dimension $m - n$ (0 represents the null vector of dimension n). In other words, we have that:

$$C = AB = AB + 0 = AB + A\lambda = A(B + \lambda) \tag{3}$$

Then, $B+\lambda$ is a solution of (2), and it is also a representative of the solution set of such equation system. What we should prove now is that $X^T(B + \lambda) = X^T B$, that is, canceling common terms and transposing the equations:

$$\lambda^T X = 0 \tag{4}$$

On the other hand, we have a set of $m-n = k$ linear dependencies between the number of bytecodes executed of the form:

$$\begin{array}{rcl} 0 & = & v_{11}x_1 + v_{12}x_2 + \cdots + v_{1m}x_m \\ 0 & = & v_{21}x_1 + v_{22}x_2 + \cdots + v_{2m}x_m \\ & \cdots & \\ 0 & = & v_{k1}x_1 + v_{k2}x_2 + \cdots + v_{km}x_m \end{array}$$

Or, rewriting them using matrices:

$$0 = VX \tag{5}$$

The result of multiplying an arbitrary vector d by V is a vector $\mu^T = (dV)$ and for the equation above, it follows that $\mu^T X = 0$.

But note that the rows of A were obtained executing a program that meets the linear dependencies too, that is, $\mu^T A^T = 0$. Transposing, we have:

$$A\mu = 0 \tag{6}$$

For this reason, we can see that as λ, μ is a member of the kernel of A, and considering the uniqueness of the kernel of a matrix, and that μ is an element of a space of dimension $m - n$, we can choose μ such that $\lambda = \mu$, that is, we can express λ as the product $(dV)^T$, as result of basic theorems of linear algebra. Therefore, we have that:

$$\lambda^T X = \mu^T X = (dV)X = d(VX) = d(0) = 0 \tag{7}$$

□

Then, the method we follow to select a representative solution B is simply to complete the equation systems with $m - n$ arbitrary equations in order to make them become determined. Such equations should be selected in such a way that the β_i values be positive, for example, by setting the cost to 0 as the time of the bytecodes that are faster, avoiding negative solutions.

5.4 Dealing with unbound instructions

We now consider the case of bytecode instructions whose execution time depends on the specific values that certain parameters can take at run time. In such cases the accuracy of our analysis can be increased by taking advantage of static term-size analysis and the addition of cost-related assertions for such instructions. Such assertions make it possible to introduce ad-hoc functions giving the size of the input parameters of the bytecode.

In fact, our system is able to deal with several metrics (e.g., value, length, size, depth, ...) as shown in (12; 11; 13), but for brevity, in the following paragraphs we will describe an example unifying lists.

Let us take, the instruction `unify_variable(V, W)` and let us assume that we want to calculate an upper bound for its execution time upon success and for the case where the two arguments to unify are lists of numbers. We assume that an upper bound to the execution time is proportional to the number of iterations necessary to scan the lists. The timing model for such instruction is thus $K_1 + K_2 * length(V)$, because if the instruction succeeds, the length of both V and W should be equal. The value of constants K_1 and K_2 is calculated by setting up two benchmarks which unify lists of different length l_1 and l_2. If the cost of `unify_variable` for these two list lengths is, respectively, B_1 and B_2, then we set up the following system of linear equations:

$$\begin{array}{rcl} B_1 & = & K_1 + K_2 \times l_1 \\ B_2 & = & K_1 + K_2 \times l_2 \end{array} \tag{8}$$

Note that B_1 and B_2 can be added to the system of equations (2) to get its values in one step, and later, we solve K_1 and K_2 in the system of linear equations (6).

6. Experimental results

In order to evaluate the techniques presented so far we need to choose a concrete bytecode language and an implementation of its abstract machine to execute and profile with. As mentioned before, the de-facto target abstract machine for most Prolog compilers is the WAM (23; 1) or one of its derivatives. In order to evaluate the feasibility of the approach we have chosen a relatively simple WAM design, which is quite close to the original WAM definition. It is based on (9), but has been ported from Java to C/C++ to achieve similar performance of other Prolog systems. The use of a relatively simple abstract machine allows evaluating the technique while avoiding the many practical complications present in modern implementations, such as having complex instructions resulting from merging other, simpler ones, or specializations of instruction and argument combinations. This of course does not preclude the application of our technique to the more complex cases.

In our concrete abstract machine, we have considered 42 equations for 43 bytecodes, differentiating the success and failure cases.

As we have seen in Proposition **??**, there exists a linear relationship between the number of bytecodes that a program will call which can be stated as:

$$\begin{aligned} 0 = \quad & x_{30} + x_{38} - x_{13} - x_{15} - x_{17} - x_{22} - x_{41} \\ & -x_{43} - x_{49} - x_{50} - x_{51} - x_{52} - x_{53} \end{aligned}$$

where the x_i represent the number of times the bytecode tagged as β_i has been executed for any program being analyzed (see Tables 4 and 6).

By Proposition 1, we are free to select any arbitrary solution of the linear system. The proposed solution has been found by setting arbitrarily the cost of fail to zero. Then, our set of linear equations, discarding those whose calibrators are composed only with one bytecode, is as follows:

$$\begin{array}{ll|ll} 0 & = \beta_{13} & c_{01} & = \beta_{01} + \beta_{07} \\ c_{20} & = \beta_{20} + \beta_{33} + \beta_{43} & c_{09} & = \beta_{09} + \beta_{24} \\ c_{11} & = \beta_{01} + \beta_{11} + 2\beta_{28} + \beta_{30} & c_{15} & = \beta_{15} + \beta_{38} \\ c_{46} & = \beta_{01} + 2\beta_{28} + \beta_{30} + \beta_{50} & c_{17} & = \beta_{17} + \beta_{30} \\ c_{42} & = \beta_{01} + 2\beta_{27} + \beta_{30} + \beta_{52} & c_{07} & = \beta_{07} + \beta_{24} \\ c_{22} & = \beta_{01} + \beta_{22} + \beta_{23} + \beta_{30} & c_{29} & = \beta_{01} + \beta_{17} + \beta_{30} \\ c_{34} & = \beta_{01} + \beta_{23} + \beta_{30} + \beta_{35} & c_{37} & = \beta_{17} + \beta_{38} \\ c_{36} & = \beta_{01} + 2\beta_{28} + \beta_{30} + \beta_{37} & c_{38} & = \beta_{07} + \beta_{24} + \beta_{39} \\ c_{40} & = \beta_{01} + \beta_{23} + \beta_{30} + \beta_{41} & c_{19} & = \beta_{19} + \beta_{33} \\ c_{43} & = \beta_{01} + \beta_{27} + \beta_{28} & c_{13} & = \beta_{01} + \beta_{13} + \beta_{30} \\ & \quad +\beta_{30} + \beta_{49} & & \\ c_{49} & = \beta_{01} + 2\beta_{19} + 2\beta_{27} & c_{51} & = \beta_{01} + 2\beta_{20} \\ & \quad +\beta_{30} + 2\beta_{31} + 2\beta_{33} + \beta_{51} & & \quad +\beta_{30} + 2\beta_{31} + \beta_{53} \end{array} \tag{9}$$

Solving this linear system we get:

$$\begin{array}{lcl} \beta_{01} & = & c_{29} - c_{17} \\ \beta_{07} & = & -c_{29} + c_{17} + c_{01} \\ \beta_{09} & = & -c_{29} + c_{17} + c_{09} - c_{07} + c_{01} \\ \beta_{11} & = & -2c_{27} - c_{13} + c_{11} \\ \beta_{13} & = & 0 \\ \beta_{15} & = & -c_{37} + c_{29} + c_{15} - c_{13} \\ \beta_{17} & = & c_{29} - c_{13} \\ \beta_{19} & = & c_{19} - c_{32} \\ \beta_{20} & = & -c_{44} - c_{32} + c_{20} \\ \beta_{22} & = & -c_{23} + c_{22} - c_{13} \\ \beta_{24} & = & c_{29} - c_{17} + c_{07} - c_{01} \\ \beta_{30} & = & -c_{29} + c_{17} + c_{13} \\ \beta_{35} & = & c_{34} - c_{23} - c_{13} \\ \beta_{37} & = & c_{36} - 2c_{27} - c_{13} \\ \beta_{38} & = & c_{37} - c_{29} + c_{13} \\ \beta_{39} & = & c_{38} - c_{07} \\ \beta_{41} & = & c_{40} - c_{23} - c_{13} \\ \beta_{49} & = & c_{43} - c_{27} - c_{26} - c_{13} \\ \beta_{50} & = & c_{46} - 2c_{27} - c_{13} \\ \beta_{51} & = & c_{49} - 2c_{30} - 2c_{26} - 2c_{19} - c_{13} \\ \beta_{52} & = & c_{42} - 2c_{26} - c_{13} \\ \beta_{53} & = & c_{51} + 2c_{44} + 2c_{32} - 2c_{30} - 2c_{20} - c_{13} \end{array} \tag{10}$$

The leftmost column of Tables 4 and 6 summarizes the calibration data for the instructions of our WAM implementation. For brevity, we actually only show those being used in the examples tested, although we have calibrated all of them. In the second column there is a tag that is the variable name in the linear equations system. In the examples we deal with a subset of Prolog which only has operations on integers, atoms, lists, and terms. Likewise, we obviate issues like modules or syntactic sugar which can be dealt with at the Prolog level. A few additional built-in predicates are required to have a minimal functionality including `write/1`, `consult/1`, etc. They are profiled separately and their timing is given to the system through assertions. This is also a valid solution in order to be able to analyze larger programs.

Bytecode	Tag	Intel (ns)	N810 (ns)	Sparc (ns)
allocate	β_{01}	29	366	1055
arith_add	β_{02}	29	489	1438
arith_div	β_{03}	29	580	1541
arith_mod	β_{04}	29	641	1553
arith_mul	β_{05}	28	519	1468
arith_sub	β_{06}	28	519	1438
call	β_{07}	11	183	261
cut	β_{08}	13	183	581
deallocate	β_{09}	7	305	142
execute	β_{12}	15	152	574
get_constant_atom	β_{14}	38	518	1211
get_constant_int	β_{16}	28	396	1157
get_level	β_{18}	28	213	1054
get_list	β_{19}	20	275	763
get_struct	β_{20}	52	642	1766
get_value	β_{21}	43	488	1457
get_variable	β_{23}	43	549	1658
proceed	β_{24}	17	61	699
put_a_constant_atom	β_{25}	20	122	594
put_a_constant_int	β_{26}	20	122	506
put_constant_atom	β_{27}	37	274	1085
put_constant_int	β_{28}	37	274	997
put_value	β_{29}	21	183	910
retry_me_else	β_{30}	33	336	999
set_constant_atom	β_{31}	26	213	861
set_constant_int	β_{32}	25	183	767
set_variable	β_{33}	29	213	850
trust_me	β_{38}	29	336	973
try_me_else	β_{39}	30	457	1132
unequal	β_{40}	21	244	1021
unify_variable(nvar,var)	β_{42}	35	396	1309
unify_variable(var,nvar)	β_{43}	35	397	1309
unify_variable(int,int)	β_{44}	32	275	1179
unify_variable(atm,atm)	β_{46}	44	427	1413
unify_variable(str(1),str(1))	β_{47}	77	885	2560
unify_variable(list(1),list(1))	β_{45}	96	1068	3291
unify_variable(list(100),list(100))	β_{48}	4062	42511	217975

Table 4. Timing model for the WAM instructions. Cost of bytecodes when they succeed.

The experiments were made on the following representative platforms:

- Ultra**Sparc**-T1, 8 cores x 1GHz (4 threads per core), 8GB of RAM, SunOS 5.10.
- **Intel** Core Duo 1.66GHz, 2GB of RAM, Ubuntu Linux 7.04.
- Nokia **N810**. 400MHz processor, 128MB of RAM, Internet Tablet OS, Maemo Linux based OS2008 51.3

In order to reduce noise in the data because of spurious results, we have repeated each experiment 20 times and present the lowest results. In the calibration step 1000 repetitions were made (i.e., $r = 1000$). When possible, the tests were performed with the machines in single-user mode, stopping unnecessary processes. System tasks such as garbage collection, which, as mentioned before, is not considered in our model at the moment, were turned off.

Platform	Timing Model (ns)
Intel	$44 + 40.62 * length(X)$
N810	$427 + 425.11 * length(X)$
Sparc	$1413 + 2179.75 * length(X)$

Table 5. Timing model given by a linear function, for `unify_variable(X,Y)` when the arguments are lists of integers, and the instruction does not fail.

Bytecode	Tag	Intel (ns)	N810 (ns)	Sparc (ns)
fail	β_{13}	0	0	0
get_constant_atom	β_{15}	32	457	1256
get_constant_int	β_{17}	26	366	1169
get_value	β_{22}	25	244	1106
unequal	β_{41}	11	61	651
unify_variable	β_{43}	121	1065	3867
unify_variable(const1,const2) const1 $\neq$ const2	β_{49}	41	154	697
unify_variable(int,int)	β_{50}	122	1035	3830
unify_variable(list(1),list(1))	β_{51}	338	3227	12229
unify_variable(atm,atm)	β_{52}	127	1126	4282
unify_variable(str(1),str(1))	β_{53}	223	2381	9239

Table 6. Timing model for the WAM instructions. Cost of bytecodes when they fail.

Tables 4 and 6 show the timing model for this WAM and the architectures studied. In the benchmarks used the `is/2` instruction is compiled into basic operations over pairs of numbers. The table shows the corresponding instructions named `arith_*`. We also have separated the cost of the instructions `put_constant`, `get_constant` when they are called for an atom or an integer. Note however, that their cost is very similar in most cases, but this will still help to reduce errors in the estimation. For the `unify_variable` instruction we have also included calibrations for several cases depending on the type and size of the input arguments in order to increase precision. In other cases, as mentioned in 5.4, the execution time of this instruction is not bounded by any a-priori known constant. Since, as also shown in Section 5.4, in our implementation it is possible to use functions instead of constants as timing model for a given instruction, in this table we include in the calibrations two data points for the case when the arguments are lists of integers, and for lists of size (length) 1 and 100 (β_{45} and β_{48} in Table 4). The value for an empty list is the same as for unifying any two equal atoms, i.e., β_{46} in Table 4. Table 5 shows the resulting timing model for `unify_variable` using these values to fit our linear model for this instruction.

Using the timing model shown in Tables 4, 5, and 6, we have performed some experiments with a series of programs on the three platforms (Intel, N810, and Sparc) in order to assess the accuracy of our technique for estimating execution times. The results of these experiments are shown in Tables 8 (Intel), 9 (N810), and 10 (Sparc).

Column **Pr. No.** lists the program identifiers, whose association with the programs and the input data sizes used is shown in Table 7. Column **Cost App.** indicates the type of approximation of the automatically inferred cost functions which estimate execution times (as a function on input data size): upper bound (**U**), lower bound (**L**), or exact (**E**). Such cost functions are shown in column **Cost Function** for the three different platforms considered

No.	Program	Data size
1	append(+A,+,-)	x=length(A)=150
2	evalpol(+A,+X,-)	x=length(A)=100
3	fib(+N,-)	x=N=16
4	hanoi(+N,+,+,+,-)	x=N=8
5	nreverse(+L,-)	x=length(L)=83
6	palindro(+A,-)	x=length(A)=9
7	powset(+A,-)	x=length(A)=11
8	list_diff(+L,+D,-)	x=length(L)=65 y=length(D)=65
9	list_inters(+L,+D,-)	x=length(L)=65 y=length(D)=65
10	substitute(+A,+B,-)	x=term_size(A)=67 y=term_size(B)=80
11	derive(+E,+,-)	x=term_size(E)=75

Table 7. List of program examples used in the experimental assessment.

in our experiments. The variables x and y represent the sizes of the input arguments to the programs which are relevant for the inference of the cost functions. The type of approximation directly depends on the one used by the static analysis described in Section 4 for estimating the number of executed instructions (as a function on input data size). The value **E** means that the lower and upper bound cost functions are the same, and thus, since the analysis is safe, this means that the exact cost function was inferred. Using the cost functions shown in column **Cost Function**, and in order to assess their accuracy, we have also estimated execution times for particular input data for each program and compared them with the observed execution times. These execution times are shown in columns **Est.** and **Obs.** respectively. Column **D.** shows the relative harmonic difference between the estimated and the observed time [2]. The source of inaccuracies in the execution time estimations of our technique come mainly from two sources: the timing model (which gives the execution time estimation of bytecodes, as shown in Tables 4 and 6)) and the static analysis (described in Section 4, which estimates the number of times that the bytecodes are executed, depending on the input data size). Since we are interested in identifying the source(s) of inaccuracies, we have also introduced the column **Prf.** It shows the result of estimating execution times using the timing model and assuming that the static analysis was perfect and obtained a function which provides the *exact* number of times that the bytecodes are executed. This obviously represents the case in which all loss of accuracy must be assigned to the timing model. The "perfect" cost model is obtained from an actual execution by instrumenting the profiler so that it records the number of times each instruction is executed for the application and the particular input data. Column **Pr.D.** shows the relative harmonic difference between **Prf.** and the observed execution time **Obs.**

The upper part of Tables 8, 9, and 10, up to the double line corresponds to examples where an exact cost function for the number of executed bytecodes was automatically inferred by the static analysis (note that, as expected, the values **Est.** and **Prf.** are the same). We can see that with an exact static analysis, the estimated execution times **Est.** are quite precise, which in turn supports the accuracy of our timing model.

It is particularly interesting to compare these results with those which were obtained using a variety of higher-level models in (19). Table 11 provides the standard deviation of the four high-level models of (19) as well as that of the abstract machine-based model presented in this paper, for the Intel platform and our set of bench-

[2] $rel\_harmonic\_diff(x, y) = (x - y)(1/x + 1/y)/2$.

Pr. No.	Cost. App.	Intel (μs) Cost Function	Est.	Prf.	Obs.	D. %	Pr.D. %
1	E	$0.73x + 0.21$	110	110	113	-2.4	-2.4
2	E	$0.69x + 0.19$	69	69	71	-2.3	-2.3
3	E	$0.69 \cdot 1.6^x + 0.21(-0.62)^x - 0.72$	1525	1525	1576	-3.3	-3.3
4	E	$-0.0042 \cdot 2^x + 0.73x \cdot 2^x - 0.86$	1501	1501	1589	-5.7	-5.7
5	E	$0.37x^2 + 0.49x + 0.12$	2569	2569	2638	-2.7	-2.7
6	E	$0.36 \cdot 2^x + 0.37x \cdot 2^x - 0.24$	1875	1875	2027	-7.8	-7.8
7	E	$0.91 \cdot 2^x + 0.87x - 0.6$	1868	1868	1931	-3.3	-3.3
8	L	$0.66x + 0.2$	43	68	81	-67.2	-17.8
	U	$0.78xy + 1.7x + 0.4$	3414	3569	3640	-6.4	-2.0
9	L	$0.83x + 0.2$	54	79	91	-54.6	-14.8
	U	$0.78xy + 1.7x + 0.4$	3414	3694	4011	-16.2	-8.2
10	L	$2x$	135	142	124	8.6	13.7
	U	$1.4xy + 1.4y + 6.1x + 4.1$	7922	2937	2858	120.6	2.7
11	L	$2.9x$	216	138	111	72.3	22.5
	U	$3x + 3$	226	216	162	34.0	29.5

Table 8. Observed and estimated execution time with cost functions, Intel platform (microseconds).

Pr. No.	Cost. App.	N810 (μs) Cost Function	Est.	Prf.	Obs.	D. %	Pr.D. %
1	E	$7.8x + 2.7$	1169	1169	1037	12.0	12.0
2	E	$7.8x + 2.7$	786	786	641	20.6	20.6
3	E	$8.3 \cdot 1.6^x + 2.5(-0.62)^x - 8.4$	18333	18333	14496	23.7	23.7
4	E	$0.74 \cdot 2^x + 7.8x \cdot 2^x - 10$	16095	16095	16144	-0.3	-0.3
5	E	$3.9x^2 + 5.7x + 1.6$	27247	27247	28381	-4.1	-4.1
6	E	$4.4 \cdot 2^x + 3.9x \cdot 2^x - 2.9$	20167	20167	20416	-1.2	-1.2
7	E	$9.5 \cdot 2^x + 10x - 6$	19517	19517	19653	-0.7	-0.7
8	L	$7.3x + 2.8$	474	744	640	-30.4	15.1
	U	$8.2xy + 19x + 5.5$	35849	37162	29266	20.4	24.1
9	L	$8.7x + 2.8$	569	839	732	-25.4	13.7
	U	$8.2xy + 19x + 5.5$	35849	38076	29907	18.2	24.4
10	L	$21x$	1399	1475	1068	27.3	32.9
	U	$15xy + 15y + 64x + 43$	85893	30375	25543	153.3	17.4
11	L	$29x$	2190	1423	854	108.7	53.3
	U	$30x + 30$	2306	2193	1342	56.8	51.1

Table 9. Observed and estimated execution time with cost functions, Nokia N810 platform (microseconds).

Model		Deviation
High Level	1	51.17 %
	2	31.06 %
	3	21.48 %
	4	58.45 %
Abs. Machine		4.72 %

Table 11. Comparison between the higher level models and the abstract machine-based model, on the Intel platform.

marks. It can be observed that the results obtained with the abstract machine-based model are more than five times better on the same platform than those obtained using the higher-level models.

With the abstract machine-based model, and for this type of programs we believe that the remaining error comes simply from the accumulated loss of accuracy of the bytecode instruction profiling and expect that making the *timing* model more precise will increase precision even further.

The lower part of Tables 8, 9, and 10 shows programs for which there is no unique value for $\texttt{Time}_p(n)$, where $\texttt{Time}_p(n)$ (as described in Section 4.1) denotes the cost (in time units) of computing a call to program p for an input of size n on a given platform. The reason is that for such programs, the number of instructions executed does not only depend on the input data sizes, but also depends on other characteristics of the input data (e.g., their actual values). Thus, for a given data size, there are actual lower and upper bounds for the cost of the program calls. For this reason, the two observed execution times shown in column **Obs.** for each program have been obtained by running the program with the input data, of the size specified in Table 7, that yield the actual lower and upper bounds to the execution times for such size. In this case, the static analysis infers approximations to such actual lower and upper bound cost functions (**L** and **U** respectively). These predictions are understandably much less accurate in these cases than those in the first part of the table, but still reasonable. In any case, lower bounds and upper bounds tend to be reasonably smaller or bigger than the observed execution times respectively. In general, for the programs in the lower part of the tables with big (absolute) values for **D.**, the (absolute) value for **Pr.D.** is reasonably small. This means that, in those cases, most of the inaccuracy in the estimation of execution times (**Est.**) comes from the static analysis, which does not approximate actual lower and upper bound cost functions accurately enough, and that the timing model used for predicting

Pr. No.	Cost. App.	Sparc (μs) Cost Function	Est.	Prf.	Obs.	D. %	Pr.D. %
1	E	$26x + 7.4$	3906	3906	4670	-18.0	-18.0
2	E	$25x + 7.1$	2543	2543	2985	-16.1	-16.1
3	E	$26 \cdot 1.6^x + 7.8(-0.62)^x - 27$	56828	56828	59120	-4.0	-4.0
4	E	$1.2 \cdot 2^x + 26x \cdot 2^x - 33$	53504	53504	63156	-16.7	-16.7
5	E	$13x^2 + 17x + 4.3$	90973	90973	109849	-19.0	-19.0
6	E	$13 \cdot 2^x + 13x \cdot 2^x - 8.5$	66400	66400	78980	-17.4	-17.4
7	E	$32 \cdot 2^x + 32x - 22$	66224	66224	78151	-16.6	-16.6
8	L	$24x + 7.1$	1574	2458	2991	-68.7	-19.7
	U	$27xy + 62x + 14$	118269	123733	129951	-9.4	-4.9
9	L	$30x + 7.1$	1940	2824	3394	-58.9	-18.5
	U	$27xy + 62x + 14$	118269	127378	133703	-12.3	-4.8
10	L	$68x$	4545	4821	4634	-1.9	4.0
	U	$48xy + 48y + 207x + 140$	277175	101779	111829	103.8	-9.4
11	L	$95x$	7104	4628	4038	59.6	13.7
	U	$98x + 98$	7454	7147	6081	20.5	16.2

Table 10. Observed and estimated execution time with cost functions, Sparc platform (microseconds).

the execution time of bytecodes is reasonably precise. Thus, we believe that using a better static analysis for inferring cost functions which take into account other characteristics of the input data, besides their sizes, would significantly improve the predictions. In any case, there is always a reasonable slack in the precision of the estimations due to the timing measurements and the timing model.

7. Conclusions and Future Work

We have developed a framework for estimating upper and lower bounds on the execution times of logic programs running on a bytecode-based abstract machine. We have shown that working at the abstract machine level allows taking into account low-level issues without having to tailor the analysis for each architecture and platform, and allows obtaining more accurate estimates than with previous approaches, including comparatively accurate upper and lower bound estimations of execution time.

Although the framework has been presented in the context of logic programs, we believe the technique can easily be applied to other languages. This adaptation of the approach, while certainly not trivial, to some extent would actually imply some simplification, since backtracking does not need to be taken into account. For example, analyses have been recently developed for Java bytecode (3) which infer the number of execution steps using similar techniques to those used in logic programming (12; 11; 13). Such analyses could be adapted, following the techniques presented herein, to take into account the bytecode timing information and would then be able to estimate actual execution time for Java programs. Appropriate cost models for Java bytecode are already being developed in (22).

We believe that the more accurate execution time estimates that can be obtained with our technique can be very useful in several contexts including parallelism, compilation, real-time applications, pervasive systems, etc. More concretely, increased timing precision can improve the effectiveness of resource/granularity control in parallel/distributed computing. This belief is based on previous experimental results, where it appeared that, even if improved precision in timing estimates is not essential, it does yield increased speedups. Also, the inferred cost functions can be used to develop automatic program optimization techniques. For example, they can be used for performing self-tuning specialization which compares statically the estimated execution time of different specialized versions (10).

Given that our experimental results are encouraging with respect to actually being able to find more accurate upper and lower bounds to program execution times, the approach may eventually also be used for verification (or falsification) of timing constraints, as in, for example, real-time systems, which was not possible in an accurate way with previous approaches. In fact, our approach (which can be adapted to take also into account destructive assignment, as in (20)) can potentially be used to solve a common problem in current WCET static analysis, where only constant WCET bounds (i.e., non dependent on input data sizes) are inferred. These bounds are not always appropriate since the WCET of a given program often depends on several input parameters, and using an absolute bound, covering all possible situations (i.e., all possible values or sizes of input), produces only a very gross over approximation (15). Substituting the estimated costs of the bytecodes by the actual worst-case costs of the instructions and using our approach, the WCET is expressed as a cost function parameterized by the size or values of input arguments, providing tighter WCET approximations. On the other hand, WCET work has produced more accurate (but, unfortunately, non-freely available) timing models which take into account many low-level parameters (such as cache behavior, pipeline state, etc.) which we have abstracted away in our work. It is clear that a combination of both techniques might be very useful in practice.

References

[1] H. Ait-Kaci. *Warren's Abstract Machine, A Tutorial Reconstruction*. MIT Press, 1991.

[2] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In *Proc. of Static Analysis Symposium (SAS)*, LNCS. Springer-Verlag, July 2008. To appear.

[3] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of java bytecode. In R. D. Nicola, editor, *16th European Symposium on Programming, ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer, March 2007.

[4] R. Bagnara, A. Pescetti, A. Zaccagnini, E. Zaffanella, and T. Zolo. Purrs: The Parma University's Recurrence Relation Solver. `http://www.cs.unipr.it/purrs`.

[5] I. Bate, G. Bernat, G. Murphy, and P. Puschner. Low-Level Analysis of a Portable Java Byte Code WCET Analysis Framework. In *Proc. 7th International Conference on Real-Time Computing Systems and Applications*, pages 39–48, Dec. 2000.

[6] I. Bate, G. Bernat, and P. Puschner. Java virtual-machine support for portable worst-case execution-time analysis. In *5th IEEE International Symposium on Object-oriented Real-time Distributed Computing, Washington, DC, USA*, Apr. 2002.

[7] R. Benzinger. Automated higher-order complexity analysis. *Theor. Comput. Sci.*, 318(1-2), 2004.

[8] F. Bueno, P. López-García, and M. Hermenegildo. Multivariant Non-Failure Analysis via Standard Abstract Interpretation. In *7th International Symposium on Functional and Logic Programming (FLOPS 2004)*, number 2998 in LNCS, pages 100–116, Heidelberg, Germany, April 2004. Springer-Verlag.

[9] S. Buettcher. Warren's Abstract Machine - A Java Implementation. `http://www.stefan.buettcher.org/cs/wam/index.html`.

[10] S.-J. Craig and M. Leuschel. Self-tuning resource aware specialisation for Prolog. In *PPDP '05: Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 23–34, New York, NY, USA, 2005. ACM Press.

[11] 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.

[12] 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.

[13] 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.

[14] S. Diehl, P. Hartel, and P. Sestoft. Abstract machines for programming language implementation. *Future Generation Computer Systems*, 16(7):739–751, 2000.

[15] A. Ermedahl, J. Gustafsson, and B. Lisper. Experiences from Industrial WCET Analysis Case Studies. In R. Wilhelm, editor, *Proc. Fifth International Workshop on Worst-Case Execution Time (WCET) Analysis*, Palma de Mallorca, July 2005.

[16] G. Gómez and Y. A. Liu. Automatic time-bound analysis for a higher-order language. In *PEPM*. ACM Press, 2002.

[17] 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.

[18] E. Y.-S. Hu, A. J. Wellings, and G. Bernat. Deriving java virtual machine timing models for portable worst-case execution time analysis. In *On The Move to Meaningful Internet Systems 2003: OTM 2003Workshops*, volume 2889 of *LNCS*, pages 411–424. Springer, October 2003.

[19] E. Mera, P. López-García, G. Puebla, M. Carro, and M. Hermenegildo. Combining Static Analysis and Profiling for Estimating Execution Times. In *Ninth International Symposium on Practical Aspects of Declarative Languages*, number 4354 in LNCS, pages 140–154. Springer-Verlag, January 2007.

[20] J. Navas, M. Méndez-Lojo, and M. Hermenegildo. Customizable Resource Usage Analysis for Java Bytecode. Technical report, University of New Mexico, Department of Computer Science, UNM, January 2008. Submitted for publication.

[21] J. Navas, E. Mera, P. López-García, and M. Hermenegildo. User-Definable Resource Bounds Analysis for Logic Programs. In *23rd International Conference on Logic Programming (ICLP 2007)*, volume 4670 of *LNCS*, pages 348–363. Springer-Verlag, September 2007.

[22] G. Román-Díez and G. Puebla. Java bytecode timing cost models. Technical Report CLIP12/2007.0, Technical University of Madrid, School of Computer Science, UPM, December 2007.

[23] D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave, Menlo Park CA 94025, 1983.

[24] R. Wilhelm. Timing analysis and timing predictability. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *Formal Methods for Components and Objects, Third International Symposium, FMCO 2004, Leiden, The Netherlands, November 2 - 5, 2004, Revised Lectures*, volume 3657 of *Lecture Notes in Computer Science*, pages 317–323. Springer, 2004.