facultad de informática

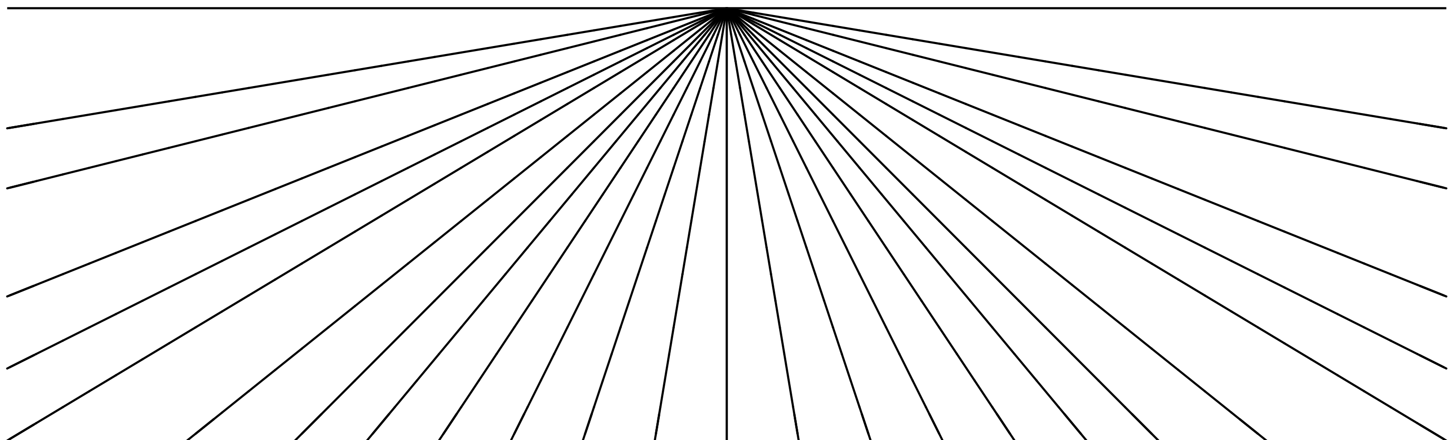
universidad politécnica de madrid

Towards Execution Time Estimation in Abstract Machine-Based (Logic) Languages

Edison Mera
Pedro Lopez-Garcia
Manuel Carro
Manuel Hermenegildo

TR Number CLIP8/2007.0

Printing date: September 1, 2007



Towards Execution Time Estimation in Abstract Machine-Based (Logic) Languages

Technical Report Number: CLIP8/2007.0

Printing date: September 1, 2007

Keywords

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

Acknowledgements

Work partially supported by Comunidad Autónoma de Madrid grant S-0505/TIC/0407 (PROMESAS), Ministry of Education and Science grant TIN2005-09207-C03-01 (MERIT/COMVERS) and EU IST FET grant IST-15905 (MOBIUS). Manuel Hermenegildo was also funded in part by the Prince of Asturias Chair in Information Science and Technology at UNM.

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 allows taking into account low-level issues without having to tailor the analysis for each architecture and platform, and instead only having to redo the calibration step. Applications of such predicted execution times include debugging/verification of time properties, granularity control in parallel/distributed computing, and resource-oriented specialization.

Resumen

Las máquinas abstractas proporcionan una cierta separación entre partas dependientes e independientes de la plataforma de ejecución en el momento de compilar y ejecutar. Muchas de las diferencias entre las distintas arquitecturas están encapsuladas en la implementación de la máquina abstracta, siendo el código de *byte* básicamente independiente del ordenador en que se ejecutará. Nos aprovechamos de este hecho para presentar un entorno que estima límites superiores e inferiores al tiempo de ejecución de programas lógicos que se ejecutan en una máquina abstracta. Nuestro enfoque incluye una etapa de *perfilamiento* independiente del programa que calcula constantes o funciones que dan límites al tiempo de ejecución de cada una de las instrucciones de la máquina abstracta. Una fase de estimación en tiempo de compilación, utilizando la información sobre el tiempo de ejecución de cada instrucción, infiere expresiones que dan límites a los tiempos de ejecución en una plataforma en concreto, tomando como entrada los tamaños de los datos. El trabajar al nivel de la máquina abstracta permite tener en cuenta consideraciones de nivel inferior sin tener que adaptar los análisis de complejidad para cada arquitectura y plataforma, teniendo que repetirse únicamente el paso de calibración. Entre las aplicaciones se incluyen la verificación / depuración de propiedades relacionadas con el tiempo, control de granularidad en computación paralela y distribuida y especialización guiada por los recursos.

Contents

1 Introduction 1

2 Mappings Between Program Segments and Bytecodes 2

3 The Timing Model: Estimating the Execution Time of Instructions 2

4 Static Cost Analysis 3

4.1 Overview of the Approach 4

4.2 Estimating the Execution Time of Clauses and Predicates 4

5 Estimating Instruction Execution Times via Profiling 5

5.1 Profiling Instructions 6

5.2 Measuring Time Accurately 6

6 Experimental results 7

7 Conclusions and Future Work 9

1 Introduction

Cost analysis has been studied for several declarative languages [6, 14, 9, 11]. In logic programming previous work has focused on inferring upper [10, 9] or lower [11, 20] 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 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]. Bernat et al. [17, 5] have proposed a portable WCET analysis for Java. 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 [21] 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. However, these preliminary results have encouraged us to develop a new analysis which takes into account lower level factors in order to improve the accuracy of the time predictions.

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) [12] and they have 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. Given a lower-level L_B (bytecode) language, measure for each instruction in L_B its execution time (or approximate it with a function if it depends on the value of an argument) in some specific abstract machine implementation while executing on a given processor and 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` [16] 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 a_i are its arguments. For conciseness, we will use I_i to refer to such expressions. Such sequences 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 $\mathtt{C}$ be a clause $\mathtt{H} \text{ :- } \mathtt{L_1}, \ldots, \mathtt{L_m}$. Let $E(\mathtt{C})$ be a function that returns the sequence of bytecodes resulting from the compilation of clause $\mathtt{C}$:

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

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

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

Note that functions $E(\mathtt{C},\mathtt{H})$ and $E(\mathtt{C},\mathtt{L_i})$ do not necessarily return the bytecodes that one would normally associate to the clause head $\mathtt{H}$ and literal $\mathtt{L_i}$ 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, in 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` instruction needs to be associated with the previous literal to that `call`, in order not to count it every time the call is retried.
Table 1 shows how `append` is compiled to bytecodes, and identifies the result of calling the $E(\mathtt{C},\mathtt{H})$ and $E(\mathtt{C},\mathtt{L_i})$ functions for each clause head and body literal. $\mathtt{H}^1$ represents the head of the first clause, and $\mathtt{H}^2$ and $\mathtt{L}_1^2$ the head of the second (recursive) clause and the first literal in such clause body (the only body literal).

3 The Timing Model: Estimating the Execution Time of Instructions

We define a function $t(I)$, that we will refer to as the *timing model*, which takes a bytecode instruction I and returns the estimated execution time for it (as in [4]).

In many cases we can assume that the time the bytecode takes to execute is constant. However there are some special cases. Some instructions have internal recursion. In many of these cases, the timing model consists of an initial constant time t_0 plus another an additional constant time

$E(\mathtt{C},\mathtt{H}^1)$	`append([], X, X).`	
	append/3/1:	try_me_else append/3/2 allocate get_constant([],0) get_variable(0,1) get_value(0,2) deallocate proceed
$E(\mathtt{C},\mathtt{H}^2)$	`append([X|Xs], Y, [X|Zs]) :-`	
	append/3/2:	trust_me allocate get_variable(0,0) unify_list(3,1,2) unify_variable(0,3) get_variable(4,1) get_variable(5,2) unify_list(7,1,6) unify_variable(5,7) put_value(2,0) put_value(4,1) put_value(6,2)
$E(\mathtt{C},\mathtt{L}^2_1)$	`append(Xs, Y, Zs).`	
		call append/3 deallocate proceed

Table 1: Sequences of bytecodes assigned to clause heads and body literals of the clauses of predicate append by the functions $E(\mathtt{C},\mathtt{H})$ and $E(\mathtt{C},\mathtt{L_i})$.

$t_{iteration}$ to cater for the cost of each iteration, i.e., we generally use a simple linear model: $t_0 + nt_{iteration}$. Consider for example the `unify_void` n instruction. Its execution time is a linear function on n, where n is the number of new unbound cells pushed on the heap [1]. 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 profiling –see Section 5).

Finally, there are some additional variable factors (such as, e.g., length of dereferencing chains) which may affect execution times (in addition to other lower-level factors, such as cache behavior, etc.). These factors are 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 will not be taken into account at this moment is garbage collection. We assume for now that garbage collection is turned off for upper bound estimation. For lower bounds garbage collection time can also be assumed to be turned off or, if left on, then it would simply make the bounds obtained end up being more conservative.

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` [15] by extending previous implementations of reduction-counting cost analyses.

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 $\texttt{Time}_p(n)$, where $\texttt{Time}_p(n)$ denotes the cost (in time units) of computing p for an input of size n. 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., [15] 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 *argument 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. [3], Mathematica, Matlab, etc.). 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 [10, 9] (for estimation of upper bounds on resolution steps) and further extended in [11] (for lower bounds). More recently, in [21] 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 [11] for details on lower-bounds cost analysis.

Consider as before a clause `C` defined as $\texttt{H :- L}_1, \ldots, \texttt{L}_m$. 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 `C` and that each $\overline{n}_i$, $i = 1 \ldots m$, is a vector such that each element corresponds

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

to the size of an input argument to literal $\mathtt{L}_i$. Assume also that $\tau(\mathtt{H})$ is the execution time needed to resolve the head $\mathtt{H}$ of the clause with the literal being solved, $\mathtt{Sols}_{\mathtt{L}_j}$ is the number of solutions literal $\mathtt{L}_j$ can generate, and $\beta(\mathtt{L_i})$ the time needed to prepare the call to literal $\mathtt{L_i}$ in the body of the clause (note that this execution time includes the time needed to create terms and only depends on the bytecodes assigned $E(\mathtt{C},\mathtt{L_i})$ but does not depend on $\overline{n}_i$). 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:

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

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

$$\beta(\mathtt{L_i}) = \sum_{I \in E(\mathtt{C},\mathtt{L_i})} t(I), i = 1, \cdots, m$$

with $E(\mathtt{C},\mathtt{L_i})$ and $t(I)$ defined as in Sections 2 and 3 respectively. Also:

$$\tau(\mathtt{H}) = \sum_{I \in E(\mathtt{C},\mathtt{H})} t(I)$$

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 generated by a predicate that will be demanded 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 that all clauses are executed (thus the execution time of the predicate is assumed to be the sum of the execution times of its defining clauses). If we take mutual exclusion among clauses into account, we can obtain a more precise estimate of the execution time of a predicate: the execution time for deterministic predicates can be approximated by the maximum of the execution times of mutually exclusive groups of clauses.

Given a predicate defined by r clauses $\mathtt{C}_1, \ldots, \mathtt{C}_r$, we can improve the precision of our analysis by noting that clause $\mathtt{C}_i$ will be tried only if clauses $\mathtt{C}_1, \ldots, \mathtt{C}_{i-1}$ fail to yield a solution. For an input of size $\overline{n}$, let $\delta_i(\overline{n})$ denote the execution time necessary to determine that clauses $\mathtt{C}_1, \ldots, \mathtt{C}_{i-1}$ will not yield a solution and that $\mathtt{C}_i$ must be tried: the function δ_i obviously has to take into account the type and cost of the indexing scheme being used in the underlying implementation. The precision of the analysis is improved by adding $\delta_i(\overline{n})$ to the cost of clause $\mathtt{C}_i$.

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 used assertion language is out of the scope of this paper, and we refer the reader to [22] for details.

5 Estimating Instruction Execution Times via Profiling

As mentioned before, data regarding the expected execution time of each instruction in the abstract machine is obtained via profiling. This process requires, on one hand, a means to actually isolate the execution of each instruction in a realistic environment, and, on the other hand, the ability to measure actual execution times as accurately as possible.

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

Figure 1: Instrumenting a simple WAM emulation loop

5.1 Profiling Instructions

More concretely, profiling aims at calculating $t(I)$ for each bytecode instruction I. One way to do this, which is the approach we have taken, is by instrumenting the source code of the WAM implementation so that time measures are taken and recorded at appropriate times. In practice a number of issues have to be carefully taken into account in order to achieve a reasonable degree of reliability in such measurements [17]. These include the choice of the point 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 are measuring very small pieces of code.

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

5.2 Measuring Time Accurately

Being able to make precise timing measurements is clearly an *a priori* condition in order to achieve reliable estimation of execution times. Unfortunately, portable, standardized time measurement calls provide very limited accuracy in most operating systems. Additionally, the overhead of calling O.S. routines is often very high when compared to the times being measured in our context and thus too much noise is introduced. Therefore, in practice, time has to be measured often in an architecture- and operating system-dependent fashion.

As an example, modern Intel (and compatible) processors include a `rdtsc` instruction [19] which returns the number of CPU cycles since the last processor reset. The (exact) frequency of the processor can then be used to work out a relatively precise estimation of the time taken by each instruction. It is in practice important to note that recent architectures can change dynamically the clock speed to save energy, so the execution times for the bytecodes should be taken in a situation similar to that in which we expect final programs to run —most probably, at full speed. Similarly, multicore processors can give erroneous results regarding the number of executed instructions, so that the estimation should be run making sure that the estimating program is *locked* in one core.

The same approach can be used in other platforms where similar instructions exist. As an example, PowerPC platforms have a "time-base counter" whose frequency is one-fourth of that of the bus clock [18]. Solaris provides a very precise `gethrtime()` routine specifically designed to take frequent timings.

In platforms where high resolution timing is difficult or impossible to achieve (e.g., mobile or embedded devices whose processors do not offer access to a CPU cycle counter or whose O.S. does not provide any high-resolution timing routine), workarounds have to be used. One possibility is to

use synthetic benchmarks which on purpose repeatedly execute the instructions under estimation during a large enough time, and later divide the total execution time by the number of times the instructions have been executed. In general this requires finding an approximate solution to a system of equations, and can be less precise than the direct measurement approach, but it does provide a solution for such processors.

Finally, issues related to non-trivial instruction scheduling at the processor level, such as out-of-order and speculative execution, need to be dealt with at the processor level as well. The main problem with out-of-order execution is that an instruction like `rdtsc`, with no data dependencies, may be executed *before* the previous (in program order) instructions have finished, thereby giving wrong CPU cycle counts. In the case of Intel systems, this can be worked around (not without other complications) by using instructions which force all previous instructions to finish, such as `cpuid`, thereby making timing estimations much more accurate.

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 [7], but has been ported from Java to C/C++. 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. The leftmost column of Table 2 summarizes the instructions of our WAM implementation that are actually being used in the examples tested. 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 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.

In the examples which follow times are given in terms of CPU cycles. The experiments where made in a computer with Intel Core Duo 1.66GHz processor, 2GB of RAM, and Ubuntu Linux 7.04. To reduce noise in the data due to spurious results, the highest 1% observed times have been discarded, and at least 1.000 measurements were taken for each instruction. Note that discarding the lowest observed times is not necessary, since we are simply trying to disregard executions where a large delay occurred. The tests were performed with the machine 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.

Table 2 shows the timing model for this WAM and architecture. As a first approximation, we assume that the time of all WAM instructions is bounded by constants, and we have taken the minimum and maximum observed times as such bounds. The `is` instruction considers only basic operations over two numbers. The execution time of the `unify_*` family of instructions is not bounded by any a-priori known constant, but here we are in practice only using unifications between a variable and a ground term. Note however, that, as mentioned before, in our implementation it is possible to use functions instead of constants as timing model for a given bytecode, so an improvement to handle such bytecodes is to introduce tighter functions depending of the bytecode arguments as input into the timing model. The errors and the differences that appear in the lower and upper bounds in Table 2 are in general due to the fact that the cost of a specific bytecode depends on its arguments and the current status of the abstract machine. However, note that, even for instructions whose execution time is constant, and even if we use CPU cycles precisely to measure such times, the measurements will still vary slightly because they depend for example on the cache status, processor pipeline status, or system tasks that cannot be controlled

Instruction	Lower	Upper	Mean	Error
allocate	550	900	600	5.5%
call*	420	560	455	4.9%
cut	600	830	629	2.4%
deallocate	360	430	385	2.0%
get_constant	1260	5400	2552	44.7%
get_level	1930	7750	2422	22.1%
get_struct	3730	14990	6306	42.4%
get_value	1080	1180	1110	1.6%
get_variable	1200	7790	2489	20.1%
is*	2660	7980	2862	19.1%
proceed	340	390	366	2.4%
put_constant	2020	10970	4064	33.1%
put_value	880	1310	958	10.0%
retry_me_else	5600	11660	6142	18.4%
trust_me	360	430	390	4.3%
try_me_else	2470	11520	4268	46.3%
unequal	1090	1330	1150	2.4%
unify_list*	1530	10020	3347	47.5%
unify_variable*	1010	2860	1501	39.3%

Table 2: A timing model expressed in terms of Upper and Lower Bounds for the WAM instructions with a confidence level of 99%, in CPU cycles.

by the user.

Table 3 shows the results of applying our technique to a series of programs for which exact cost functions could be automatically derived. The timing model we used was the (simplistic) one in Table 2. The maximum, minimum, and average execution times given by this model (measured in CPU cycles, and labeled as U, L, and M, respectively), are used to generate another cost function that returns CPU cycles as result (column labeled *Cost Function*). The execution time predicted by this cost function appears in the following column, together with the error with respect to the experimentally measured execution time. The reported execution time does not take into account the overhead of profiling.

In general, the combination of an exact cost function and an average of the running time of the bytecode instructions gives results which are quite close to the actual execution time (always with an error less than 10%). These results are more than three times better on the same platform than those obtained for similar programs using higher-level models [21]. With the abstract machine-based model, for this type of programs we believe 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. Also, lower bounds are indeed always smaller and upper bounds larger than the actual execution times (which was not the case in [21]), with variations within 160%.

Table 4 diplays the results for a series of programs in which the automatically obtained lower and upper cost function approximations (labeled as L and U, respectively, in the second column) differ. The predictions in Table 4 are understandably much less accurate than those in Table 4. In any case, lower bounds (L-L combinations) are still always smaller and upper bounds (U-U combinations) larger than the actual execution times. In most cases the best approximation is given by the combination of "upper approximation of cost execution" with "mean of bytecode instruction execution time," even if this combination still sometimes produces inaccurate results for this class of programs. This is, in any case, quite understandable since, to start with, no exact cost function was deduced for them.

Program	Prof. App.	Static Estimation			Exec. Time
		Cost Function	Time	Error	
palindro(+A,-)	M	$30x \cdot 2^{x-1} + 22 \cdot 2^x - 12$	79399	-4%	82326
x=length(A)=9	L	$23x \cdot 2^{x-1} + 25 \cdot 2^x - 14$	47876	-42%	
	U	$71x \cdot 2^{x-1} + 56 \cdot 2^x - 31$	198718	141%	
evalpol(+A,+X,-)	M	$30.5x + 13.2$	3066	-2%	3286
x=length(A)=100	L	$20.4x + 7.44$	2048	-34%	
	U	$78.8x + 32.4$	7913	152%	
nrev(+L,-)	M	$14.8x^2 + 28.4x + 9.7$	104511	-2%	106315
x=length(L)=83	L	$8.8x^2 + 17.5x + 5.3$	62186	-42%	
	U	$38x^2 + 68.8x + 23.1$	267615	152%	
powset(+A,-)	M	$18 \cdot 2^{x+1} + 40x - 13$	75071	0.3%	74851
x=length(A)=11	L	$11 \cdot 2^{x+1} + 25x - 8$	43771	-42%	
	U	$48 \cdot 2^{x+1} + 95x - 33$	197807	164%	
append(+A,+,-)	M	$29.7x + 11.8$	4790	5%	4259
x=length(A)=150	L	$17.6x + 7.3$	2652	-38%	
	U	$76.0x + 28.3$	11433	169%	
hanoi(+N,+,+,+,-)	M	$2^x(30x + 18) - 56$	65330	-9%	72181
x=N=8	L	$2^x(18x + 13) - 39$	39380	-45%	
	U	$2^x(76x + 39) - 126$	165673	130%	
fib(+N,-)	M	$47 \cdot 1.6^x + 12(-0.6)^x - 45$	102813	-10%	113920
x=N=16	L	$33 \cdot 1.6^x + 8(-0.6)^x - 32$	72519	-36%	
	U	$110 \cdot 1.6^x + 31(-0.6)^x - 108$	243085	113%	

Table 3: Observed and estimated execution time (with exact cost functions), measured in thousands of CPU cycles. L = Lower, M = Mean, U = Upper.

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 obtaining more accurate estimates than with previous approaches, including correct upper and lower bounds on execution time.

Although the framework has been presented in the context of logic programs, the technique can easily be applied to other languages, which generally results in a simplification, since backtracking does not need to be taken into account. For example, analyses have been recently developed for Java bytecode [2] which infer the number of execution steps using similar techniques to those used in logic programming [10, 9, 11]. 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.

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 [8].

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 can also be

Program	An. App.	Prof. App.	Static Estimation			Exec. Time
			Cost Function	Time	Error	
list_inters(+L,+D,-)	L	M	$41x + 13$	2667	-31%	3894
x=length(L)=65	L	L	$28x + 7$	1800	-53%	
y=length(D)=65	U	M	$29xy + 103x + 26$	127673	-13%	146852
	U	U	$77xy + 264x + 65$	344171	134%	
list_diff(+L,+D,-)	L	M	$36x + 13$	2352	-35%	3597
x=length(L)=65	L	L	$25x + 7$	1635	-55%	
y=length(D)=65	U	M	$29xy + 95x + 26$	127200	-13%	145204
	U	U	$77xy + 243x + 65$	342771	136%	
derive(+E,+,-)	L	M	$44.41x$	3331	-68%	10031
x=term_size(E)=75	L	L	$29.49x$	2212	-79%	
	U	M	$177x + 171$	13435	34%	10031
	U	U	$402x + 386$	30565	205%	
substitute(+A,+B,-)	L	M	$44.41x$	2976	-70%	9929
x=term_size(A)=67	L	L	$29.49x$	1976	-80%	
y=length(B)=80	U	M	$69(x + 1)y + 200x + 96$	386872	178%	138871
	U	U	$174(x + 1)y + 475x+231$	979350	605%	
flatten(+A,-B)	U	M	$14.8(x - 1)^2 + 92x + 29$	224652	566%	33718
x=term_size(A)=121	U	U	$38.6(x - 1)^2 + 227x+77$	583417	1630%	

Table 4: Observed and estimated execution time (with approximate cost functions), measured in thousands of CPU cycles. L = Lower, M = Mean, U = Upper.

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, the approach can 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 [13]. Using our tool, the WCET is expressed as a cost function parametrized by the size or values of input arguments, providing tighter WCET approximations.

Bibliography

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

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

[3] 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/`.

[4] 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.

[5] 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.

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

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

[8] S.J. Craig and M. Leuschel. Self-tuning resource aware specialisation for Prolog. In *Proc. of PPDP'05*, pages 23–34. ACM Press, 2005.

[9] 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.

[10] 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.

[11] 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.

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

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

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

[15] 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.

[16] 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.

[17] E. Yu-Shing 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.

[18] IBM. *PowerPC Virtual Environment Architecture. Book II.* IBM, 2005.
`http://www.ibm.com/developerworks/eserver/library/es-archguide-v2.html`.

[19] Intel. *Intel's Application Notes. Using the RDTSC Instruction for Performance Monitoring.* Intel, 1997.
`http://cedar.intel.com/software/idap/media/pdf/rdtscmp1.pdf`.

[20] 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.

[21] 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.

[22] 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)*, LNCS. Springer-Verlag, September 2007.

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

[24] Reinhard Wilhelm. Timing analysis and timing predictability. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem 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.