Source–Level Execution Time Estimation of C Programs

C. Brandolese, W. Fornaciari, F. Salice, D. Sciuto

Politecnico di Milano
Piazza L. da Vinci, 32
20133 Milano, Italy

{brandole,fornacia,salice,sciuto}@elet.polimi.it

ABSTRACT

In this paper a comprehensive methodology for software execution time estimation is presented. The methodology is supported by rigorous mathematical models of C statements in terms of elementary operations. The deterministic contribution is combined with a statistical term accounting for all those aspects that cannot be quantified exactly. The methodology has been validated by realizing a complete prototype toolset, used to carry out the experiments.

1. INTRODUCTION

The recent design trend of embedded applications requires the analysis and optimization of performance and power in all the components of the system, especially during the first stages of the design, when alternative solutions are compared. The current pervasiveness of microprocessor–based architectures, is enforcing the importance of a fast analysis of the software, carried out through the entire development/compilation flow, namely from the source level down to the assembly. To this purpose, in literature several authors explored the problem to determine bounds on the execution time of a process running on a microprocessor (e.g. [4], [7], [6], [5], [1], [9], [8]); unfortunately, this task is becoming more and more complicated with the current CPU features. Many investigations attacked the problem at a coarse grain, focusing mainly on the WCET (Worst Case Execution Time), because the target application is often real–time constrained. In general, the problem has two aspects:

- *program paths* analysis, to determine which sequences of instructions will be executed and how they influence the WCET;
- *microarchitectural* analysis, i.e. the modeling of the hardware system executing the program.

Efficient estimation of WCET is, however, a hard task: to be *decidible* it requires absence of recursive functions, absence of function pointers and bounded loops [4] and, in addition, the scope of the path information strongly impacts the estimation accuracy. To cope with the latter problem, [7] includes such information by adding loop bounds and maximum execution bounds for some statements, while [6] uses path information specifying interactions among different program statements through regular expressions and annotations. Feasibility analysis of program paths using a set of linear equations has been considered in [5]. In any case, the analysis is computationally very hard and thus is not suitable during design space exploration, when frequently the proper hw/sw architecture has to be chosen quickly comparing alternatives. Microarchitectural-related issues, modeling the timing analysis of the assembly instructions, are another source of possible estimation errors. Some investigations in this field are reported in [1] and cycle-accurate performance models have also been presented in [9], while the problem of annotating C code guessing the compiler behavior has been addressed in [8]. Our goal is to provide a *framework*, general enough to gather a broad range of specification formalisms, as well as to take into account the peculiarities of the actual processor on which the software will be finally mapped. In particular, we focused on the problem of efficiently estimating the average timing features of the code, making at the same time the formulation open to easily incorporate a statistic characterization of the parameters, if necessary. It constitutes the base for hw/sw partitioning or simply to analyze the impact of different microprocessors on the performance. Work is in progress to extend it towards the analysis of the software power at the source level. The paper is organized as follows. Next section presents the basic abstractions have been introduced to categorize the code statements, using C as reference language. The general model describing the software timing characteristics is addressed in Section 3, where the notations for profiling and timing are also introduced. Interpretation of such a model and a statistical characterization of the timing estimates are presented in Section 4 and Section 5, respectively. Finally, Section 6 describes the experimental environment used to derive the parameters and to validate the methodology.

2. LANGUAGE MODEL

Our initial effort has been devoted to the definition of a modeling framework to capture in a general manner the structure of a program. To model a high–level language in a constructive and hierarchical way it is necessary to define what an 'elementary component' is. In the following, elementary components will be referred to as *atoms*.

The most critical point in defining the atoms of a language is the choice of the granularity at which the language is analyzed. To this purpose it is helpful to describe the language, in this case C, using the formalism of grammars.
The starting point is the definition of a set of terminal symbols: these symbols are the basic building blocks of the atoms of the language and, as a consequence, determine the granularity of the analysis. It is important to note that some of the terminal symbols adopted would be non–terminals in the complete grammar of the language[1] and would be not disjoint but rather related by a production.
The essential terminals are, as in usual grammars, operators, keywords and special symbols such as `=`, `(`, `)`, `{`, `}`, `for`, `return` and others. In addition, to the purpose of the definition of atoms, it is useful introducing the following terminals: *var* (variables), *expr* (arithmetic expressions) and *cond*.
Using this set of terminals a few productions can already be built. Consider, as an example the productions of figure 1.

```
st-break    :=  break ;
st-return   :=  return expr ;
st-assign   :=  var = expr ;
             |  var [ expr ] = expr ;
```

Figure 1: Productions for some simple statements

The partial grammar of figure 1 clarifies how the terminals combines to give more complex portions of the language. In a similar manner, it is possible to define more complex statements, such as `while` or `if`. Their grammar is reported in figure 2.

```
st-while  :=  while ( cond ) stmt
st-if     :=  if ( cond ) stmt
           |  if ( cond ) stmt else stmt
```

Figure 2: Productions for two complex statements

The non–terminal symbol *stmt* is a generic statement defined by the partial grammar of figure 3.

```
stmt          :=  simple-stmt
               |  block-stmt
simple-stmt   :=  st-assign
               |  ...
block-stmt    :=  { stmt-list }
stmt-list     :=  stmt
               |  stmt-list stmt
```

Figure 3: Definition of the generic statement

Proceeding in this way it is possible to define all the statements and constructs of the C language. The outcome is an incomplete grammar Γ that lacks the production for all the symbols that have been assumed as terminals but allows defining in a formal way the concept of atom.

[1]As an example consider the atoms *var* and *expr* defined below.

DEFINITION 1. *The terminal symbols on the right–hand side of a production of the grammar Γ constitute an atom whose name is defined in the left–hand side.*

According to this definition and referring, as an example, to the production for the `while` statement, the atom whose name is *while* is constituted by the keyword `while` and the couple of parentheses enclosing the conditional expression *cond*. The body of the `while` construct is a generic statement but it is not part of the *while* atom itself. Based on the concept of atom, the following definitions and notations can be introduced.

3. GENERAL MODEL

3.1 Source code notation

According to the grammar Γ and to definition (1), a generic source code C_s can be described as a list of L_s couples describing the atoms and the hierarchical relations between atoms. The symbols summarized in table 1 and described in the following are used to formally define this concept.

Table 1: Notation for source code

Symbol	Meaning
s	source code index
i	atom index
j	data set index
C_s	s-th source code
$A_{s,i}$	i-th atom of the s-th source code
L_s	number of atoms of the s-th source code
$D_{s,j}$	j-th data set for the s-th source code

The symbol $D_{s,j}$, far from giving details on the type, size and value of data, is used to refer to a specific run of the source code. Data is made dependent both from the spanning index j and the source code index s since a set of data must be compatible with the specific source code.

3.2 Profiling notation

When a source code C_s is run with data $D_{s,j}$ each atom, observed as a whole at source level, is executed a certain number of times. The symbols summarized in table 2 are used to formally express profiling information.

Table 2: Notation for source–level profiling

Symbol	Meaning
$n()$	returns the number of executions
$N_{s,i,j}$	number of executions of the atom $A_{s,i}$ with data $D_{s,j}$
$N_{s,j}$	number of execution of all atoms of C_s with data $D_{s,j}$

Since the function $n()$ returns a dynamic measure, it must depend on a set of data. According to the definition and notations previously introduced, the following relations hold:

$$N_{s,i,j} = n(A_{s,i}, D_{s,j}) \tag{1}$$

$$N_{s,j} = \sum_{i=1}^{L_s} n(A_{s,i}, D_{s,j}) = \sum_{i=1}^{L_s} N_{s,i,j} \tag{2}$$

The count returned by the function $n()$ has no explicit relation with the actual execution time or with the number of clock cycles.

3.3 Timing notation

In order to account for a real measure of the execution time, the functions and the symbols of table 3 are introduced.

Table 3: Notation for timing

Symbol	Meaning
$t()$	returns the actual execution time
$\overline{t}()$	returns the reference execution time
$\hat{t}()$	returns the estimated execution time
$T_{s,j}$	actual execution time of C_s with data $D_{s,j}$
$\overline{T}_{s,j}$	reference execution time C_s with data $D_{s,j}$
$\hat{T}_{s,j}$	estimated execution time of C_s with data $D_{s,j}$

The measure of unit of time, in this context, needs to be independent of the operating frequency of a microprocessor and thus the concept of clock–cycles per instruction (CPI) is used.

In the construction of the mathematical form the upper–case, shorthand forms will be used more often. Their meaning is clarified by the three following simple relations:

$$T_{s,j} = t(C_s, D_{s,j}) \tag{3}$$

$$\overline{T}_{s,j} = \overline{t}(C_s, D_{s,j}) \tag{4}$$

$$\hat{T}_{s,j} = \hat{t}(C_s, D_{s,j}) \tag{5}$$

The total actual execution time $T_{s,j}$ of the source code C_s with data $D_{s,j}$ can be expressed as the sum of the execution time of each atom $A_{s,i}$, counted $N_{s,i,j}$ times, that is:

$$\begin{aligned} T_{s,j} &= \sum_{i=1}^{L_s} t(A_{s,i}, D_{s,j}) \cdot n(A_{s,i}, D_{s,j}) = \\ &= \sum_{i=1}^{L_s} t(A_{s,i}, D_{s,j}) \cdot N_{s,i,j} \end{aligned} \tag{6}$$

The reference and estimated times can be also expressed with similar relations, but their discussion is deferred to the next section.

3.4 Mathematical formulation

In this section a general mathematical model that can be used to describe the timing of a software, starting from its high–level source description is presented. The hypotheses on which the model is based are explained and partly justified on the basis of some preliminary experimental results.

As indicated in the concluding part of the previous section, the execution time of a source code C_s with input data $D_{s,j}$ can be expressed as:

$$T_{s,j} = \sum_{i=1}^{L_s} N_{s,i,j} \cdot t(A_{s,i}, D_{s,j}) \tag{7}$$

where the dependence of the real execution time of an atom is explicitly indicated by the second argument of the function $t()$. This equation defines what is called *actual time* throughout the rest of the paper.

To afford the complexity of the problem it is useful to consider the ideal conditions defined by these assumptions:

- the target architecture has an unlimited number registers and all the variables of the code are allocated to a fixed register;
- the initial value of the variables is pre–loaded in the corresponding register and thus no explicit initialization is required;
- inter–atom compiler optimizations are neglected;
- intra–atom compiler optimizations are neglected.

The first two bullets of this list lead to an underestimate of the real execution time while the last two tend to produce an overestimate.

Let $\overline{t}()$ denote the function returning the execution time in these ideal conditions. The time in ideal conditions is referred to as *reference time*.

The aim of the model is to determine a function $\hat{t}()$ returning an estimate of the actual execution time such that the estimation error is minimum. The relation that express the total estimated timing of a source code is similar to the one for the real timing:

$$\hat{T}_{s,j} = \sum_{i=1}^{L_s} N_{s,i,j} \cdot \hat{t}(A_{s,i}, C_s) \tag{8}$$

It is worth noting that the function $\hat{t}()$ does not depend on the actual data fed as input to the source code: this dependence is completely accounted for in the execution count coefficient $N_{s,i,j}$. This is essential in order to allow an a–priori characterization of the atoms. On the other hand, an explicit dependence on the source code C_s, considered as a whole, is present: this point is clarified later on. The error to be minimized over a number of source codes and input data sets is thus:

$$\epsilon_{s,j}^2 = \left(T_{s,j} - \hat{T}_{s,j}\right)^2 \tag{9}$$

The basic idea behind the model is that the estimated timing of each atom can be expressed as the sum of two contribution: the reference timing, that accounts for all the *deterministic* aspects in ideal conditions, and a *statistical* deviation that depends on complex factors such as the structure of the source code, the characteristics of the compiler, the actual architecture etc. This idea is formally expressed by the following relation:

$$\hat{t}(A_{s,i}, C_s) = \overline{t}(A_{s,j}) + \delta(C_s, A_{s,i}) \tag{10}$$

where the form of the function $\delta()$ can be arbitrarily defined.

An analysis of some preliminary timing measurement results, suggests that δ should depend on the specific atom $A_{s,i}$ as well as on the source code considered as a whole. For the sake of generality, it might be useful and interesting to consider a dependence not only on the atom $A_{s,j}$ but rather on a range of adjacent atoms, thus:

$$\delta = \delta(A_{s,i-k_1}, \ldots, A_{s,i}, \ldots A_{s,i+k_2}, C_s) \tag{11}$$

where k_1 and k_2 determine the extension of the range around $A_{s,i}$. The dependence on the source code and the range of atoms defined thus far is only symbolic, since atoms and source code are neither numbers nor functions, and does not specify the explicit mathematical form. To refine the expression of the function it is necessary to define some measures to be performed statically on an arbitrary range of atoms and on the entire source code. As examples consider measurements such as the number of consecutive sequential statements, the maximum nesting level of the whole source, the number of variable used and so on. For the sake of conciseness, let $\mathbf{q}()$ and $\mathbf{Q}()$ be two vector functions operating on a range of atoms and on the tree representing the entire source code, respectively. Formally:

$$\mathbf{q} = \mathbf{q}(A_{s,i-k_1}, \ldots, A_{s,i}, \ldots A_{s,i+k_2}) \tag{12}$$

$$\mathbf{Q} = \mathbf{Q}(C_s) \tag{13}$$

According to the definitions and hypotheses discussed thus far, and combining equations (8), (10), (11), (12) and (13) the estimated time can be expressed as:

$$\hat{T}_{s,j} = \sum_{i=1}^{L_s} N_{s,i,j} \cdot \left[\overline{t}(A_{s,i}) + \delta(\mathbf{q}, \mathbf{Q})\right] \tag{14}$$

This last equation expresses a very general and flexible model but involves a number of scalar, vectorial and functional unknowns that render it almost mathematically untreatable.

The next paragraph proposes some simplifications that lead to an affordable mathematical problem.

4. SIMPLIFIED MODEL

At this point it is useful to summarize what, in the model, is to be considered known and what is unknown. The following components are known:

- The reference time of each atom $\overline{t}(A_{s,j})$. The value can be derived using the analytical models of the atoms, combined with the timings of elementary operations. Both the atom mathematical models and the set of operation that are considered elementary are defined by refinement of those presented in [2].
- The execution count of atoms $N_{s,i,j}$. These values can be derived from source–level profiling of the code. A preliminary solution to this problem has been implemented by instrumenting the original source code, compiling and running it on a generic platform.

On the other hand, the components listed below are still undefined:

- The mathematical form of the function δ with respect to the vector functions $\mathbf{q}$ and $\mathbf{Q}$.
- The meaning of the vector functions $\mathbf{q}$ and $\mathbf{q}$. They should be applied to lists of atoms, and return numeric values corresponding to some sort of *measure* on the source code. Though it is possible to suggest a number of such measurement functions, it is all but straightforward to determine whether the adopted functions are meaningful for the problem or not.

The problem concerning the form of the function δ can be faced assuming a multilinear dependence on the two vector functions: this leads to a significant mathematical simplification with an acceptable loss of generality. This assumption leads to the expression:

$$\delta = \mathbf{a} \times \mathbf{q} + \mathbf{b} \times \mathbf{Q} + c \tag{15}$$

that, expanding the vectors and denoting with n_q and n_Q the number of elements of the two vectors $\mathbf{q}$ and $\mathbf{Q}$, respectively, becomes:

$$\delta = [\, a_1 \ \cdots \ a_{n_q} \,] \times \mathbf{q} + [\, b_1 \ \cdots \ b_{n_q} \,] \times \mathbf{Q} + c \tag{16}$$

In this equation $a_1, \ldots, b_1, \ldots$ and c are the parameters of the model and their values have to be determined statistically in order to minimize the square error, while $q_1, \ldots$ and $Q_1, \ldots$ are the results of the measures performed on ranges of atoms and the whole source code, respectively.

5. STATISTICAL CHARACTERIZATION

Consider now a source code C_s and a compatible set of data $D_{s,j}$. According to equations (15) and (14) and noting that the reference times $\overline{t}(A_{s,i})$ and the vectors $\mathbf{q}$ and $\mathbf{Q}$ are known, the estimated time can be expressed as:

$$\hat{T}_{s,j} = \sum_{i=1}^{L_s} N_{s,i,j} \cdot \left[\overline{t}(A_{s,i}) + \mathbf{a} \times \mathbf{q} + \mathbf{b} \times \mathbf{Q} + c\right] \tag{17}$$

Distributing the summation an noting that $\mathbf{a}$ and $\mathbf{b}$ are intended to be independent of the atom, this relation can be rewritten as:

$$\hat{T}_{s,j} = \overline{T}_{s,j} + \mathbf{a} \times \mathbf{q}_{tot,j} + \mathbf{b} \times \mathbf{Q}_{tot,j} + c \cdot N_{s,j} \tag{18}$$

where:

$$\overline{T}_{s,j} = \sum_{i=1}^{L_s} N_{s,i,j} \cdot \overline{t}(A_{s,i}) \tag{19}$$

$$\mathbf{q}_{tot,j} = \sum_{i=1}^{L_s} N_{s,i,j} \cdot \mathbf{q} \tag{20}$$

$$\mathbf{Q}_{tot,j} = \sum_{i=1}^{L_s} N_{s,i,j} \cdot \mathbf{Q} \tag{21}$$

$$N_{s,j} = \sum_{i=1}^{L_s} N_{s,i,j} \tag{22}$$

Let now C_s be fixed and let the data set $D_{s,j}$ vary with the index $j = [1, \ldots, n_d]$. For each data set, the actual timing is given by $T_{s,j}$ while the estimated timing can be derived by using the relations above. With these data it is possible to build the linear system:

$$\mathbf{T}_s = \overline{\mathbf{T}}_s + \mathbf{A} \times \mathbf{X} \tag{23}$$

where the matrix $\mathbf{A}$ is:

$$\mathbf{A} = \begin{bmatrix} \mathbf{q}_{tot,1}^T & \mathbf{Q}_{tot,1}^T & N_{s,1} \\ \vdots & \vdots & \vdots \\ \mathbf{q}_{tot,n_d}^T & \mathbf{Q}_{tot,n_d}^T & N_{s,n_d} \end{bmatrix} \tag{24}$$

and the column vector $\mathbf{X}$ has the form:

$$\mathbf{X} = \begin{bmatrix} \mathbf{a}^T \\ \mathbf{b}^T \\ c \end{bmatrix} \tag{25}$$

Equation (23) can be rewritten as:

$$\mathbf{Y} = \mathbf{A} \times \mathbf{X} \tag{26}$$

where the vector $\mathbf{Y}$ has been defined as:

$$\mathbf{Y} = \mathbf{T}_s - \overline{\mathbf{T}}_s \tag{27}$$

In the linear form of equation (26), $\mathbf{Y}$ is a $n_d \times 1$ column vector, $\mathbf{A}$ is a $n_d \times (n_q + n_Q + 1)$ matrix and $\mathbf{X}$ is a $(n_q + n_Q + 1) \times 1$ column vector. Since the matrix $\mathbf{A}$ is not square and the experimental setup is such that $n_d \gg (n_q + n_Q + 1)$, the linear system can only be solved in a statistical sense. A simple and well known statistical estimator is the least square method that leads to the solution:

$$\mathbf{X} = \left(\mathbf{A}^T \times \mathbf{A}\right)^{-1} \times \mathbf{A}^T \times \mathbf{Y} \tag{28}$$

Equation (28) allows deriving the parameters of the simplified model for any number of metric functions $\mathbf{q}$ and $\mathbf{Q}$.

6. RESULTS

Performing experiments in order to derive actual data to be used to obtain the model parameters is a lengthy operation since it requires executing the code under analysis in *single–step mode*[2] and collecting large amounts of data. Furthermore, the original source code must be suitably instrumented and elaborated in a non–trivial manner. These preparation phases, unfortunately, can be automated only partly and thus a significant human intervention is required.

At the time this thesis is being written, only a few experimental data are available. These data has been used to derive the preliminary results reported in the following.

6.1 Experimental setup

The experiments performed to derive the data reported in the next paragraph have been set up as described below.

The code to be analyzed is stored in a separate file named after the function name: `<function>.c`. This source file is then processed manually to obtain the two modified versions `<function>.prof.c` and `<function>.spix.c` to be used to generate two executables: `main.prof` and `main.spix`. The former is used for assembly–level timing measure while the latter is used for source–level estimation.

By running the executable `main.prof` a file containing the atoms execution count and a file containing the number of clock cycles taken by each atom are generated. Combining these results, the overall estimated time can be easily obtained.

The executable `main.spix` is run to generate a binary data file that is then interpreted by the utilities of the Spix toolset. The outcome is the assembly code of the executable, annotated with the execution count of each assembly instruction. The timing of each instruction is then derived parsing the assembly code and looking up the clock cycles counts in a library file. Combining the counts with the instructions timings, the overall exact[3] timing can be derived.

[2]The single–step mode causes the execution of a child process, under the control of its parent, one assembly instruction at a time. In most machines, this is done via signals (software interrupts), introducing a considerable overhead that leads to run times 100-300 times longer than the original code.

The model used to derive these preliminary results assumes for the function δ the simplest linear form: $\delta = c$. The dependence on ranges of atoms and on the complete source code is dropped. As a consequence of this simplification, the values of the constant c are expected to depend on the source code being considered. A more complex model, that is going to be verified, might account for the different source codes including some measures on the entire code, i.e. some elements of the vector $\mathbf{Q}$.

6.2 Validation

The methodology has been validated considering a benchmark set composed of a dozen of programs written in C (RLE Encoding, 16–bit CRC, sorting algorithms, GCD, recursive functions like Fibonacci, ...). The model has been applied to the data obtained from 50 runs of each function of the benchmark set with different, randomly generated data. The resulting value for the parameter is $c = -0.0441$. This value has been used to estimate the execution time of the quick sort algorithm[4] whose code is reported in figure 4.

```
void qsort(int ilo, int ihi, int a[] ) {
    int pivot, ulo, uhi, ieq, temp;

    if( ilo >= ihi )
        return;

    pivot = a[(ilo + ihi)/2];
    ieq   = ilo;
    ulo   = ilo;
    uhi   = ihi;

    while( ulo <= uhi )
        if( a[uhi] > pivot )
            uhi--;
        else {
            temp   = a[ulo];
            a[ulo] = a[uhi];
            a[uhi] = temp;
            if( a[ulo] < pivot ) {
                temp   = a[ieq];
                a[ieq] = a[ulo];
                a[ulo] = temp;
                ieq++;
            }
            ulo++;
        }
    }

    qsort(ilo, ieq - 1, a);
    qsort(uhi + 1, ihi, a);
}
```

Figure 4: The source code of the function `qsort`

The obtained results show an average error of 3–4 %. The estimates and relative errors are plotted in figures 5 and 6.

[3]This procedure produces the timing in ideal conditions i.e. when no stalls are introduced and no cache misses occur.

[4]The algorithm was not part of the set of testbenches used to derive the constant c.

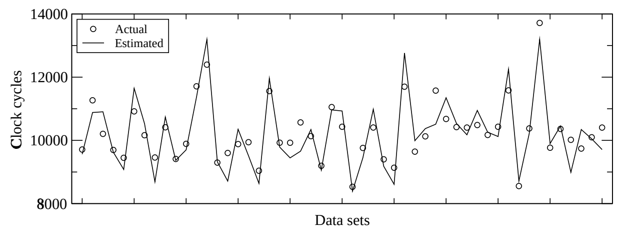

Figure 5: Actual and estimated timing

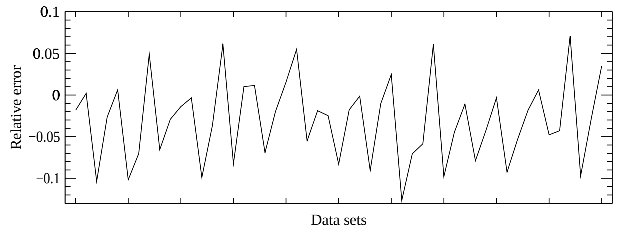

Figure 6: Relative errors

The distribution of the relative errors, shown in figure 7, is approximately Gaussian with average $\mu_\epsilon = -0.030$ and standard deviation $\sigma_\epsilon^2 = 0.048$. This suggests that the least square estimator is correct and unpolarized. An analytical proof of this properties is beyond the scope of this paper.

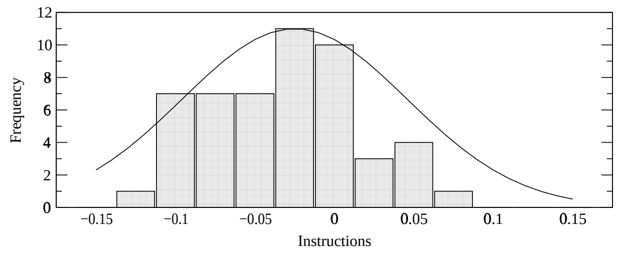

Figure 7: Distribution of relative errors

7. CONCLUSIONS

This paper presented an approach to characterize the timing features of a program at the source-level. The proposed strategy is based on a formal hierarchical analysis of the code structure and gathers a mathematical model of the timing issues with profiling/statistical information. The entire analysis flow has been implemented within a prototype toolset that we are currently using for validating and tuning the model parameters. In particular, the obtained estimates on timing fits very well actual values, for a broad range of benchmarks with a Gaussian distribution of the absolute relative errors, whose magnitude is around 3-4%.
Work is in progress to:

- extend the analysis strategies in order to include energy estimation at the source–level, based on the low-level processor characterization described in [3];
- include this fast and flexible timing estimation strategy within the TOSCA hw/sw codesign framework.

8. REFERENCES

[1] J. R. Bammi, W. Kruijtzer, and L. Lavagno. Software performance estimation strategies in a system-level design tool. In *Proceedings of the Hardware Software Codesign Workshop*, pages 82–86, December 2000.

[2] C. Brandolese, W. Fornaciari, L. Pomante, F. Salice, and D. Sciuto. A multi–level strategy for software power estimation. In *Proceedings of the 13th Int. Symposium on System Synthesis*, pages 187–192, September 2000.

[3] C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto. An instruction–level functionality–based energy estimation model for 32-bits microprocessors. In *Proceedings of the 37th Design Automation Conference*, pages 346–351, June 2000.

[4] E. Klingerman and A. D. S. 2ex. Real-time euclid: A language for reliable real-time systems. *IEEE Transactions on Software Engineering*, 12(9):795–825, November 1993.

[5] S. Malik, M. Martonosi, and Y. T. S. Li. Static timing analysis of embedded software. In *Proceedings of the Design Automation Conference*, pages 147–152, June 1997.

[6] C. Y. Park. Predicting deterministing execution time of real-time programs. In *PhD Thesis*, Seattle, August 1992. University of Whashington.

[7] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *The Journal of Real-Time Systems*, 1(2):160–176, September 1989.

[8] K. Suzuki and A. Sangiovanni-Vincentelli. Efficient software performance estimation methods for hardware/software codesign. In *Proceedings of the Design Automation Conference*, June 1996.

[9] V. Zivojnovic and H. Meyr. Compiled hw/sw co-simulation. In *Proceedings of the Design Automation Conference*, June 1996.