Library Functions Timing Characterization for Source-Level Analysis

C. Brandolese, W. Fornaciari, F. Salice, D. Sciuto
Politecnico di Milano, Piazza L. da Vinci, 32 – 20133 Milano, Italy

Abstract

Execution time estimation of software at source-level is nowadays a crucial phase of the system design flow, especially for portable devices and real-time systems. From a source-level perspective, a call to an external library function is a black-box: only the binary code of such functions is, in fact, available. This paper proposes a methodology for library functions analysis within a source-level estimation framework.

1. Introduction

The problem of timing estimation deserves special attention for real-time applications, especially when it focuses on the analysis of software performed at a high level of abstraction. In a generic program[1], different sources contribute to the overall execution time:

Algorithms are specific to the program under analysis and are written by the user. All the source-level information is available for such portions.

Library functions appear as calls to external functions, generally collected into libraries. For these, only the function name and the number and type of the arguments is visible at source-level.

System calls appear in the form of library function calls, the function simply being a wrapper around a software interrupt routine.

At source-level, thus, both library and system calls are black-boxes and nothing can be inferred by inspecting the source code. In most cases, libraries are pre-compiled binaries and, therefore, source-level analysis is unfeasible. This work proposes a methodology to face this problem, which is crucial to design real-time applications. To the best of the authors' knowledge, no previous study has systematically approached this topic under a wide and practical perspective.

[1]The language considered in this paper is C.

2. Problem definition

The wide semantic spectrum provided by programming languages makes the function cost vary dramatically and irregularly with respect to the input data. The only information available for any software library are the object code and the function prototype found in the library's header files. Library function calls are considered as *holes* in the user source code: wherever call is present, in fact, the analysis should temporarily switch to assembly-level, estimate the cost and include the result in the overall cost of the program. The goal of the proposed methodology is to avoid such an abstraction-level switching by providing a mathematical model of the library function to be used *statically.* This requires each function to be pre-characterized by means of a *dynamic* analysis aimed at extracting the dependence of the function cost on some relevant properties of the input data. To this purpose, the function under analysis must be fed with *proper* data, actually executed on a host machine and its behavior analyzed. The critical point of such a flow is the automatic generation of input data since the necessary information to do that can only be provided by the user. The knowledge a programmer has on the functions he uses in his application can be split in three kinds of information:

Function interface. The function name, the number and the type of each parameter. This information is found in the header files.

Data semantics. The semantic meaning of the data passed to the function, i.e. the way the function interprets input data. Such knowledge cannot be inferred from the function interface or object code.

Function semantics. The algorithm implemented. This information is irrelevant to our purpose.

The only information on a function that can be automatically derived is, thus, the C type of its arguments but this is not sufficient since the same C type can have different meanings within a program. For example, the C type `char*` (referred to as *formal type* in the following) can represent a C string, an array of characters or a pointer to a single character. The user must thus describe what is the *semantics* of each parameter (referred to as *sematic type* in the following), according to its actual usage. In addition, the user must *quantitatively* describe the statistical properties of the data that the function will operate on in its application. When the semantic type and the statistical properties of each argument have been defined, the cost of the function can be automatically estimated [1]. To actually perform the estimate, a C program, the *stub*, is synthesized. Such a program executes the function under test a given number of times using as input the data generated based on the semantic and statistical information provided by the user. All the executions are traced, and the traces analyzed to compute the estimated timing. This produces a lumped model of the cost, that can be conveniently used at source level. The complete analysis flow is sketched in figure 1.

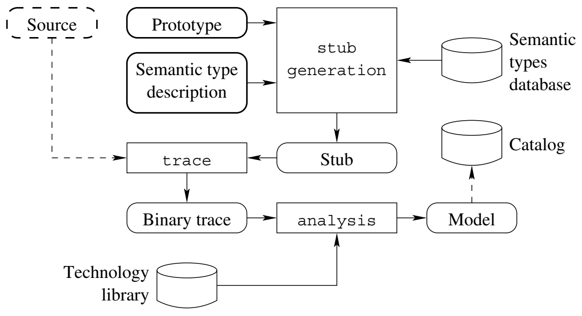

Figure 1. Function cost modelling flow

3. Experimental results

The proposed methodology has been implemented and applied to the Standard C Library [4]. The characterization has been organized in four steps: 1) function grouping based on the type of arguments; 2) semantic type definitions; 3) Dynamic analysis and collection of statistical data; 4) Post-analysis. The flow has been run on a Intel Pentium III workstation at 933 MHz with 512 MB of main memory, under Linux Mandrake 8.0. The full characterization process for 65 functions required 3,5 hours and approximately 168 MB of disk space. The resulting density functions have then been classified into four groups according to their shape, ranging from very simple to apparently chaotic behaviors (see Figure 2). Functions belonging to the first three groups have then been post-processed in order to extract, whenever possible, a relation between the data density function (input) and the execution time density (output). Out of the 65 functions considered, 59 showed a sufficiently regular behavior to be modelled in closed form (Dirac's δ, Uniform, Exponential and Gaussian). The function models have then been used within a source-level estimation flow in order to provide timing figures to the library function calls that would have been otherwise "holes" since no source code

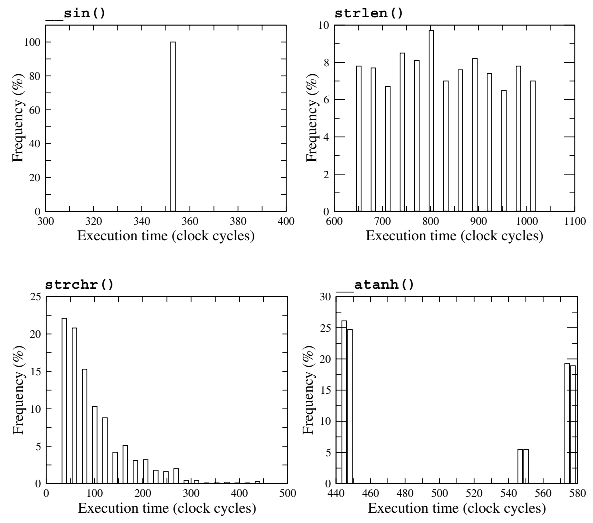

Figure 2. Sample function timing densities

for them is available. The source-level model and estimation flow used to validate the results obtained for function characterization is the PEOPLE toolset [3, 2]. Table 1 reports the timing results obtained with the source-level estimation flow, both with and without the additional cost of the function calls.

		w/o Library		w/ Library	
Source	**Actual**	**Est.**	**Err.**	**Est.**	**Err.**
bsort	90.726.800	67.110.733	−26.0%	86.515.586	−4.6%
newton	4.093.049	1.202.695	−70.6%	3.714.181	−9.3%
poly	863.228	180.323	−79.1%	847.879	−1.8%
linsrch	940.101	660.121	−29.8%	1.039.492	10.6%
Overall	—	—	**51.4%**	—	**6.6%**

Table 1. Suorce-level estimation results

4. Conclusions

The paper presented a methodology for timing estimation of library functions within the more abstract context of source-level analysis of programs. Preliminary results are also presented to confirm the viability of the the proposed approach in the context of timing analysis of real-time software and the accuracy provided by the toolset and the underlying models.

References

[1] L. Ceresoli. *Time and Power Characterization of Software Libraries.* Report 02.059, Cefriel, Milan, 2002.
[2] M. Dadomo. *Estimation of the Energy/Timing Characteristics of Source-Level C Code.* Report 02.002, Cefriel, Milan, 2002.
[3] PEOPLE. *Power Estimation for Fast Exploration of Embedded Systems.* Esprit-ESD project N. 26769.
[4] P. Plauger. *The Standard C Library.* Prentice Hall, Englewood Cliffs, NJ, USA, 1992.