Embedded software execution time estimation at different abstraction levels.[1]

P. González-Aledo, L. Díaz Suarez, P. Sanchez Espeso
Dpto. de Ingeniería Microelectrónica, Universidad de Cantabria
{pabloga, luisds, sanchez}@teisa.unican.es

Abstract— **The increasing popularity of portable devices has driven a great effort in analyzing and optimizing software execution time in embedded systems. Additionally, most of the system's functionality is implemented in software which enables high levels of flexibility and re-configurability. This software is executed on an increasingly complex platform, which is evolving toward high-performance Multiprocessor System on Chip (MPSoC).**

Current design methodologies need early estimations to guide the design process, but this growing complexity has made the process far from easy. Many techniques have been proposed to provide fast software execution estimations. Methodologies based on Instruction Set Simulators (ISS) use traces of instructions at ASM level providing accurate results with relatively low simulation time (typically 100 times faster than register transfer level simulations). However, an additional speed-up is needed in order to evaluate real embedded applications. Other techniques are based on clusters of instructions instead of single ones, providing less accurate results at the cost of faster simulations. An interesting way to extract these blocks is to characterize every element of the grammar of a high-level software language. This technique is called "Source Code Analysis" and works at source level. Low-level details are not considered in this technique so faster simulations can be performed with a little accuracy penalty.

This paper presents four approaches for time estimation at different abstraction levels and compares them in terms of accuracy and execution time. Some techniques are introduced to speed up the simulation while providing accurate results.

Keywords- software execution time abstraction level comparative

I. INTRODUCTION

Embedded systems are characterized by the fact that their functionality is divided between hardware and software components. Hardware components normally implement functionalities with strong timing constraints that require short execution times. On the other hand, software functionality can be reconfigured or fixed even after the product has been released, it's easier to develop and more flexible although it normally has higher execution times. These reasons are driving most of the functionality to software and embedded systems are evolving every day, including more and more software. However, in recent years, all versions of the International Technology Roadmap for Semiconductors (ITRS) have highlighted that the system complexities dramatically increase with this HW/SW partition. This increasing complexity and time-to-market pressure has driven many companies to adopt design methodologies that include virtual platforms. These virtual models enable the evaluation of performance and task execution time before a real hardware prototype is available. During the first steps of the design process, the main objective of these tools is to provide fast performance estimations with a relatively low error. These estimations can be used to guide the selection of the hardware elements of the platform and even the HW/SW partition. This paper is focused on early software time estimation techniques that can be integrated into a virtual platform.

Most commercial CAD frameworks include time of execution simulators, but they normally work at logic or RT level. Even though these tools have a good accuracy, the speed (number of simulated instructions per second) is not enough to evaluate real embedded software. Other tools use traces of Instruction Set Simulators (ISS) to speed up the software simulation time. Even though they increase the simulation speed, this is not enough for some applications. Other approaches execute the application source-code in the host computer (native simulation). Some additional code is included in the application in order to extract information during execution. This information is used to estimate several parameters such as target execution time and power consumption. This approach has several advantages: it provides the maximum speed-up, it can be used from the first steps of the design process and the software designer can directly use the host environment to develop the application. This paper presents a software time estimation technique based-on native simulation. The main contributions are:

- We compare a technique based on C-source-level time estimation with ASM approaches in terms of accuracy and execution time.
- We present a technique to cluster ASM instructions in order to speed up the simulation that presents some advantages over the state-of-the-art alternatives.
- We characterize these clusters of ASM instructions at different abstraction levels and compare the results obtained in terms of time estimation accuracy.

After this introduction, the next section presents an overview of the proposed software execution time simulation techniques. Section 3 details the proposed solution and section 4 presents some experimental results.

[1] *This work has been supported by the CDTI under project Artemis JU 100029 SCALOPES and the CICYT under TEC2008-04107.*

II. STATE OF THE ART

Current state-of the art techniques for software execution time estimation can be classified in six groups:

RTL modeling [1][2]: A complete description of the platform is used for execution time estimation. This leads to very accurate results, as every logic module has a detailed model that closely mimics real hardware functionality, but the method is also very slow in terms of simulation time. Microprocessor, memories, buses, peripherals and any hardware component of the platform can be modeled without losing any generality.

Instruction-level Modeling [3][4]: This level of abstraction suppresses the hardware-specific details of the platform and focuses on microprocessor instructions. The basic idea is to characterize the time spent on each individual instruction execution, so a description of the entire instruction set in terms of execution time is needed. Characterization of pairs of instructions has even been proposed instead of single ones in order to take into consideration the effects in the internal pipeline [16]. This can require the measurement of up to 100,000 instructions pairs, which supposes a non-trivial effort. This methodology is based on traces constructed from Instruction Set Simulators (ISS) that can easily exhaust disk space. The execution time of this solution is also prohibitive for characterization in early design phases. Hardware details have been omitted at this level so hardware-specific times not bounded to software routines are not modeled. Time of peripherals can still be modeled if there is a close relationship between this time and a software parameter, but there is not always a general relationship between these terms. For example, the time required for an Ethernet controller to send a package can be modeled as it is dependent on packet size (software parameter), but it is also dependent on the network status or distance to the other node, which cannot be defined at this level. Even though the contribution of these terms is low in the total software execution time for the reasons explained in the introduction (most of the functionality is being moved to the software part), this represents a loss of accuracy.

Estimation based on real hardware counters [5][6]: Today, some microprocessors already have embedded programmable event counters to measure performance, so time estimations can be obtained from these counters. However, in many specific microprocessors, these counters are not available, or are limited. Moreover, in many cases the platform is not yet constructed so these techniques cannot be used.

Black-Box macro-modeling [7]: Based on the observation that most of the new software being developed nowadays is constructed using big portions of reused code glued with some sections of new specific functionality, these methods aim to characterize and model the sections of reused code and formalize the framework to combine them. Two aspects are important in this approach; how the basic-blocks are obtained and how they are characterized. To obtain the basic block C-source code is commonly used. Typical approximations are:

- Black-boxes are created from C-Source functions. Every function in the C-Source code has a black-box model. When the function is executed, the model is also executed and the execution time is updated.
- Black-boxes are created from ASM basic-blocks. A group of instructions that are always executed together is called a `basic-block'. Basic-blocks have proved to be a very appropriate piece of code for performance analysis.

There are also many techniques to characterize the execution time of basic blocks. When the basic block has been created from C-functions, the parameters of the function also usually model the execution time. When the basic block has been obtained from ASM sentences a fixed cost is commonly used, as there is only a single path to follow in the execution. In this work we measure how accurate this approach is.

Estimations based on source-code analysis [13][14] : This approximation uses the embedded software and a rough characterization of the platform in which it will be executed as input. This characterization usually assigns a time cost to each element of the grammar of a language (operators, loops, memory-accesses, etc.). The model can be embedded in the application program and, even though it may model a specific platform, be compiled in any other one, so it is not necessary to execute it in the target system (it can be executed in the host computer where many simulations per minute can be performed).

High-level system description: This technique can be seen as a black-box macro model where the entire platform is considered as a single black-box. For this technique, a set of hardware and software parameters are defined and the whole platform is characterized based on them. These parameters are chosen based on experience, although some techniques to automate characterization have been analyzed [10]. This approach provides very low execution time but also low accuracy. The approach is strictly constrained to particular applications and it is not easily generalized (some systems are accurately modeled with this approach while others are not, without an a-priori way of knowing to which group our system belongs).

We consider four different techniques to perform software execution estimations in this paper. They are based on source-code analysis, instruction-level modeling and black-box macro-modeling.

Reference [12] uses basic-blocks for software characterization. The approach is similar to our work because it uses basic blocks to perform the estimation. This approach, however has the drawbacks of low-level estimation techniques, as they use binary code to obtain basic blocks so every machine level instruction must be analyzed to detect the block boundaries. On the other hand, [13][14] are also similar to our approach; they perform time and performance estimation based on the grammar of the C-source code, assigning a cost to every element of this grammar. However, performing the characterization at high level loses the fine-grain details of the architecture and the compiler optimizations. With our approach we overcome the problem of looking at every low-level

instruction to detect the basic blocks and still maintain a fine-grain detail of the architecture that produces highly accurate results.

Our method is constructed on top of SystemC, so we can actually simulate the hardware peripherals of the platform such as Ethernet MAC controllers, memories, buses, etc.

Additionally, [13][14] need to modify a C-compiler to extract and characterize basic blocks. Our approach can be used with a general compiler (gcc), so a fast exploration of architectures can be performed without the need to modify a compiler for each type of target architecture.

III. TIME ESTIMATION METHODOLOGIES

Next, we will introduce the four techniques we compare in this paper. We refer to them as Method 1, Method 2, Method 3 and Method 4.

- Method 1 is based on an Instruction Set Simulator with cycle accuracy. The results of this approach will be used as a gold model. Other approaches will be compared with these results. We use Skyeye [11] to simulate the system. Skyeye is an ISS simulator based in an ASM approach. Each ASM instruction is given an execution time that is dependent on the pipeline state and the value of the registers. Data and instruction caches are considered.
- Method 2 is based on native simulation. The application source code is annotated with an execution time per basic-block. Every basic block is characterized at ASM-level. A cost is assigned to each instruction based on its opcode but we do not consider the internal state of the MPSoC (as in method 1).
- Method 3 is similar to method 1 but we define an average execution time per ASM instruction, assigning the same value for all the instructions. In this case, the execution time of a basic-block is proportional to the number of ASM instructions.
- Method 4 is a technique based on C-Source Code analysis. In this case the elements necessary to estimate software consumption are extracted from C-Source code using the operators technique explained in section III.B. The considered grammar is a high-level view of the functionality of the code and does not take into consideration specific implementation details. Particularly, we do not have details for internal pipe status, exact number of instructions per element in the grammar, cache sizes and policies or register values.

A. *Basic-Block methodology (Method 2 and 3)*

This section is focused on ASM estimation techniques. First, we present a general overview of the method, a way to extract basic blocks that operates directly on the C-Source code and a way to characterize them that is simple and accurate enough for early characterization. Figure 1 shows the main steps of the process.

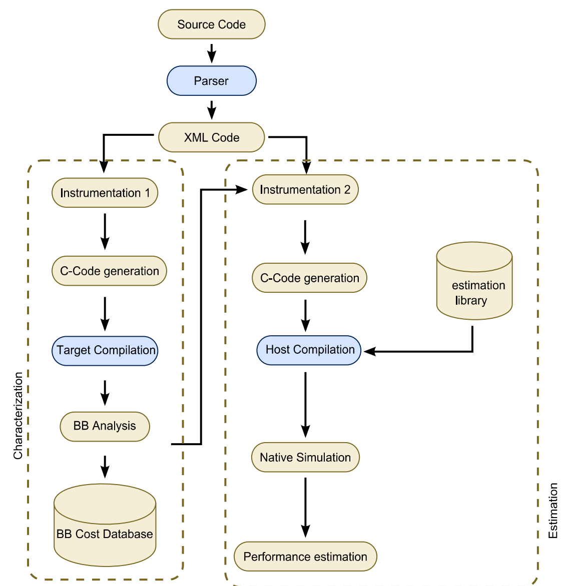


Figure 1: Time estimation methodology

The application C/C++ source-code is analyzed by a parser to obtain a language-independent representation (XML model). In contrast to other basic-block-based approaches for performance and time estimation [7], the annotation is done directly in the C-source code, so the model execution is fast even for big and complex systems.

The annotation process has two steps. During the first step (characterization), some assembler directives are inserted into the code (Figure 2). The goal is to directly identify the basic-block code even with compiler optimization. We have taken advantage of the GNU/gcc facilities to mix assembler instructions with source code with the 'asm' sentence. To prevent the compiler optimizations from removing these marks they must be declared 'volatile'. Our methodology is based on the fact that boundaries between basic blocks are preserved between all the representations and optimizations making up the compilation process. The C-annotated code is then compiled with a target compiler and the resultant code is analyzed to detect the marks introduced. The output of this process is a database that characterizes each basic block, storing its number of instructions.

Original Code	Annotated Code

if (a[0] == 0)
{
  a[0] = 1;
 }
	asm("b _uc_mark_2_am3__");
	if (a[0] == 0) {
	asm("b _uc_mark_3_rm__");
	  a[0] = 1;
	 asm("b _uc_mark_4__");
	}
	{asm("b _uc_mark_5_ai1__");

Figure 2: Example of code annotation

During the second step (native simulation, right-side of Figure 1), the application code is annotated with a function per basic block that provides performance estimation. These functions use two additional parameters that are generated during host execution. These parameters compute the number of cache misses in a basic block:

- ICmiss : number of instruction cache misses
- DCmiss: number of data cache misses

The description of the techniques that generate these parameters is beyond the scope of the paper [15]. With this annotation and these parameters, the execution time of a block can be estimated as:

$$T = C \cdot \overline{T_m} + \overline{T_{imiss}} \cdot ICmisses + \overline{T_{dmiss}} \cdot DCmisses \quad (1)$$

Where C is the number of instructions of the basic-block, $\overline{T_m}$ the mean time per instruction, $\overline{T_{imiss}}$ the mean time spent per miss in the instruction cache and $\overline{T_{dmiss}}$ the mean time spent per miss in the data cache.

The time needed to execute an ASM sentence is usually considered constant and the same for every instruction in the Instruction set architecture (ISA) of the platform. However, there are differences in time costs between different instructions inside the ISA and also between the same instructions executed at different times.

The first approach to characterize an instruction in terms of execution time is by a constant value that is added to estimation time each time the instruction is executed. This cost of an instruction can be considered to be the cost associated with the basic processing needed to finish the instruction. However, this base cost is also affected by other inter-instruction effects that can occur in real programs. Examples of these alterations are prefetch buffer and write buffer stalls, other pipeline stalls and cache misses. Base cost per instruction does not take into consideration the impact of these effects, so separate costs need to be assigned to them.

We can deduce from the last explanation that the base cost of instructions is always fixed, and characteristic of the compilation result. However, extra cost depends on the input data sets of the system so this information is not available at compilation time, as the input data is unknown in this step. As the annotation process is done in the compilation step and the cost in terms of instructions is not known at this time, assigning a mean cost in this step implies a loss of accuracy.

B. *Operator-based modeling (Method 4)*

As explained before, these methods perform the estimation based on a high-level language, typically C. The input to this method is a software code and a rough characterization of the platform in terms of hardware and software. Here we use a characterization based on elements of the C/C++ grammar, assigning a constant cost per element of the grammar. Time estimation is performed assigning a cost to each C++ operator, and overloading the application software operators to keep track of the total time. We consider N the number of C statements corresponding to some functionality and T the time needed to execute that functionality in the target processor. We can express T as:

$$T = \sum_{n=1}^{N} T_n \quad (2)$$

Where T_n is the time spent executing the functionality of the nth statement of the source code.

Going down in the level of abstraction that represents the functionality, we can decompose the functionality of each statement of the high-level C language in a set of machine level instructions and rewrite T_n as the sum of all them.

$$T_n = \sum_{m=0}^{Mn} T_{mn} \quad \Rightarrow \quad T = \sum_{n=1}^{N} \sum_{m=0}^{Mn} T_{mn} \quad (3)$$

Where T_{mn} is the time that is spent executing machine-level instruction ‘m’ when it is associated with C-statement ‘n’.

This time, T_{mn}, depends of the base costs and extra costs as explained before. To model the basic block at a high level of abstraction we consider only the base cost, that we call here “mean time per instruction” $\overline{T}$. We can rewrite the execution time as:

$$T = \sum_{n=1}^{N} I_n \cdot \overline{T} \quad (4)$$

In Equation 4, we have obtained the approximate execution time of a task composed of *N* elements of the C-grammar, each of them being decomposed by the compiler in ASM instructions.

The time estimation methodology for operator-based modeling is shown in Figure 3.

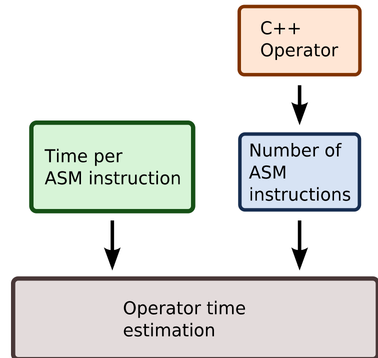


Figure 3: Operator cost estimation

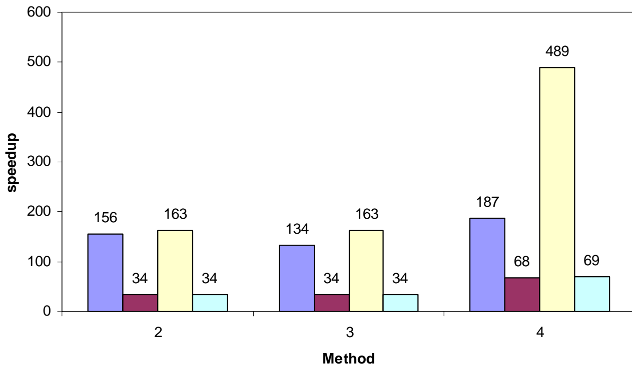
Compiling characteristic code in which single instances of the C-grammar are used, the number of instructions that are mapped to these high-level elements can be obtained. Once the elements of the grammar have been characterized, they can be directly reused for all designs developed in the same HW platform.

	Method 1		Method 2		Method 3		Method 4	
	Simulation Time (ms)	Estimated time (ns)	Simulation Time (ms)	Estimated time (ns)	Simulation Time (ms)	Estimated time (ns)	Simulation Time (ms)	Estimated time (ns)
Bubble	938	4163	6	4152	7	4122	5	3945
Factorial	206	1961	6	1954	6	1993	3	1558
Queens	489	16909	3	16787	3	16505	1	16479
Hanoi	209	3517	6	3491	6	3305	3	3535

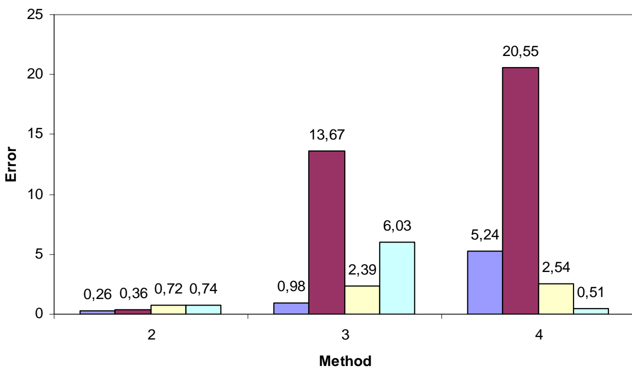
Table 1: Simulation results

	Method 1 (reference)		Method 2		Method 3		Method 4	
	Speed-up	error	Speed-up	Error (%)	Speed-up	Error (%)	Speed-up	Error (%)
Bubble	1	0	156	**0.26**	134	0.98	**187**	5.24
Factorial	1	0	34	**0.36**	34	1.63	**68**	20.55
Queens	1	0	163	**0.72**	163	2.39	**489**	2.54
Hanoi	1	0	34	**0.74**	34	6.03	**69**	0.51

Table 2: Speed-up and error in the methods considered



Graph 1: Speed-up of the different methods considered



Graph 2: Error of the different methods considered

IV. Estimation results

In order to compare the estimation methodologies several test benches have been executed in an ARM920T-based platform. The average execution time per instruction has been obtained from reference [17]. Average time per cache-miss has been measured in a real hardware platform. Four test benches have been used: a bubble-sort, a recursive implementation of the Hanoi game, a recursive factorial and a solution of the queens problem (how to put 8 queens on a chess board in such a way that no one can be taken by another). Table 1, Graph 1 and Graph 2 present the results. Table 1 shows an estimation of the execution times in the target platform (estimated time). The table also shows the simulation time in the host platform (2.61 GHz Dual-core PC computer). Method 1 is the paper reference (a cycle-accurate ISS Simulator). Method 2, 3 and 4 are the native-simulation techniques that are presented in this paper. The first conclusion that can be extracted from the data is that native simulation is 2-3 orders of magnitude faster than ISS approaches, even when the error is less than 1%. This information is included in table 2 in which speed-up and accuracy error are presented. Comparing the results of different approaches, it can be observed that Method 2 provides a speed-up of 2-3 orders of magnitude with an accuracy error less than 1%. Method 4 provides a speed-up of two times compared to method 2 but with an accuracy penalty of 20%.

Method 2 and 3 have similar speed-ups but the accuracy error of method 3 (2.39%) is higher than method 2 (0.74%). The speed-up is the same for these methods as the characterization of the basic blocks is performed in the same way in both cases. The reason to use an average value per instruction (method 3) instead of a full ISA characterization (method 2), even considering that worse results are obtained with the same speed-up is that the average time per instruction is usually given in the processor datasheet, while an entire ISA characterization is far from easy to obtain. Operator cost (method 4) gives important speed-ups, but with more error, especially in the case of the factorial test case where the recursive implementations of the factorial function cause a lot of problems due to the stack-handling. This technique can be

used for the first steps of platform design, where precise values are not required but fast simulation is a must. We base the operator methodology on the idea that each element of the C-grammar is converted by the compiler into an ASM template which is then filled with specific details. However, real compilers (especially when optimizations are introduced), perform some operations at ASM level after this template-filling to increase performance. This explains the errors of this method.

V. CONCLUSIONS

New design methodologies require higher levels of abstraction due to their increasing complexity. As a consequence, new estimation techniques are required.

This paper presents three techniques for software execution time estimation that work at C-Source code level. These techniques are based on native simulation; a set of timing annotations is introduced in the source code so performance estimations are obtained during host execution. We compare these techniques with a cycle-accurate ISS one in terms of execution time and estimation error.

We show that software execution time can be characterized by associating a constant time cost to each element of the high-level description grammar. These costs are derived from the number of machine instructions that are needed to execute the corresponding statement. Once the costs have been obtained, they can be directly reused for all designs developed in the same HW platform. This technique can be used in early design steps where fast simulation is more important than accurate results.

To achieve more accuracy, we present a methodology that extracts basic blocks from C-Source code and explain two ways to characterize these basic blocks. The former needs a complete characterization of the platform in terms of base cost of each instruction in the ISA. The latter considers an average value for every instruction. The results have shown that considering just the number of ASM instructions and number of cache misses in a block provides enough accuracy and speed-up. Comparing these results with the ISS approach we see that very accurate results (error less than 1%) can be obtained, with a speed-up of 2-3 orders of magnitude.

REFERENCES

[1] Gurumurthi, S., Sivasubramaniam, A., Irwin, M. J., Vijaykrishnan, N., Kandemir, M., Li, T., and John, L. K. 2002. Using Complete Machine Simulation for Software Power Estimation: The SoftWatt Approach. In Proceedings of the 8th international Symposium on High-Performance Computer Architecture (February 02 - 06, 2002). HPCA. IEEE Computer Society, Washington, DC, 141.

[2] Brooks, D., Tiwari, V., and Martonosi, M. 2000. Wattch: a framework for architectural-level power analysis and optimizations. In Proceedings of the 27th Annual international Symposium on Computer Architecture (Vancouver, British Columbia, Canada). ISCA '00. ACM, New York, NY, 83-94.

[3] Filho, S. J., Aguiar, A., Marcon, C. A., and Hessel, F. P. 2008. High-Level Estimation of Execution Time and Energy Consumption for Fast Homogeneous MPSoCs Prototyping. In *Proceedings of the 2008 the 19th IEEE/IFIP international Symposium on Rapid System Prototyping* (June 02 - 05, 2008). RSP. IEEE Computer Society, Washington, DC, 27-33.

[4] Lajolo, M., Lazarescu, M., and Sangiovanni-Vincentelli, A. 1999. A compilation-based software estimation scheme for hardware/software co-simulation. In Proceedings of the Seventh international Workshop on Hardware/Software Codesign (Rome, Italy). CODES '99. ACM, New York, NY, 85-89.

[5] Martin P. Lawitzky, David C. Snowdon and Stefan M. Petters "Integrating real time and power management in a real system" Proceedings of the 4th Workshop on Operating System Platforms for Embedded Real-Time Applications, Prague, Czech Republic, July, 2008

[6] Li, T. and John, L. K. 2003. Run-time modeling and estimation of operating system power consumption. In Proceedings of the 2003 ACM SIGMETRICS international Conference on Measurement and Modeling of Computer Systems (San Diego, CA, USA, June 11 - 14, 2003). SIGMETRICS '03. ACM, New York, NY, 160-171

[7] Tan, T. K., Raghunathan, A. K., Lakishminarayana, G., and Jha, N. K. 2001. High-level software energy macro-modeling. In Proceedings of the 38th Annual Design Automation Conference (Las Vegas, Nevada, United States). DAC '01. ACM, New York, NY, 605-610

[8] C.-H. Hsu and U. Kremer. The design, implementation and evaluation of a compiler algorithm for CPU energy reduction. SIGPLAN Not., 38(5);38-48 2003

[9] F.Xie, M. Martonosi, and S. Malik. Compile-time dynamic voltage scaling settings: Opportunities and limits. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 49-62, New York, NY, USA, 2003. ACM Press

[10] Nandi, A. and Marculescu, R. 2001. System-level power/performance analysis for embedded systems design. In *Proceedings of the 38th Annual Design Automation Conference* (Las Vegas, Nevada, United States). DAC '01. ACM, New York, NY, 599-604.

[11] S. Kang, H.Wang, Y.Chen, X.Wang,Y.Dai. "Skyeye: An Instruction Simulator with Energy Awareness", Lecture Notes in Computer Science, Embedded Software and Systems, Volume 3605/2005, pp 456-461

[12] Tiwari, V., Malik, S., and Wolfe, A. 2002. Power analysis of embedded software: a first step towards software power minimization. In Readings in Hardware/Software Co-Design, G. De Micheli, R. Ernst, and W. Wolf, Eds. The Morgan Kaufmann Systems On Silicon Series. Kluwer Academic Publishers, Norwell, MA, 222-230.

[13] C.Brandolese, W. Fornaciari and F. Salice, "Timing and Energy Estimation of C Programs", ACM Transactions on Embedded System Computing.

[14] Brandolese, C., Fornaciari, W., Salice, F., and Sciuto, D. 2001. Source-level execution time estimation of C programs. In Proceedings of the Ninth international Symposium on Hardware/Software Codesign (Copenhagen, Denmark). CODES '01. ACM, New York, NY, 98-103.

[15] J. Castillo, H. Posadas, E. Villar, M. Martínez (DS2) "Fast Instruction Cache Modeling for Approximate Timed HW/SW Co-Simulation " 20th Great Lakes Symposium on VLSI (GLSVLSI'10), Providence, USA. 2010-05

[16] Tiwari, V., Malik, S., Wolfe, A., and Lee, M. T. 1996. Instruction level power analysis and optimization of software. J. VLSI Signal Process. Syst. 13, 2-3 (Aug. 1996), 223-238.

[17] S. Furber, "ARM, System-On-Chip Architecture. 2nd ed", Adison-Wesley., 2000