Evaluating Performance and Power of Object-Oriented Vs. Procedural Programming in Embedded Processors

Alexander Chatzigeorgiou and George Stephanides

Department of Applied Informatics,
University of Macedonia,
156 Egnatia Str., 54006 Thessaloniki, Greece
alec@ieee.org, steph@uom.gr

Abstract. The development of high-performance and low power portable devices relies on both the underlying hardware architecture and technology as well as on the application software that executes on embedded processor cores. It has been extensively pointed out that the increasing complexity and decreasing time-to-market of embedded software can only be confronted by the use of modular and reusable code, which forces software designers to use objected oriented programming languages such as C++. However, the object-oriented approach is known to introduce a significant performance penalty compared to classical procedural programming. In this paper, the object oriented programming style is evaluated in terms of both performance and power for embedded applications. A set of benchmark kernels is compiled and executed on an embedded processor simulator, while the results are fed to instruction level and memory power models to estimate the power consumption of each system component for both programming styles.

1 Introduction

The increasing demand for high-performance portable systems based on embedded processors has raised the interest of many research efforts with focus on low power design. Low power consumption is of primary importance for portable devices since it determines their battery lifetime and weight as well as the maximum possible integration scale because of the related cooling and reliability issues [1]. The challenge to meet these design constraints is further complicated by the tradeoff between performance and power: Increased performance, for example in terms of higher clock frequency, usually comes at the cost of increased power dissipation.

To reduce the system power consumption, techniques at both the hardware and the software domain have been developed. The overall target of the most recent research that is summarized in [2] is to reduce the dynamic power dissipation, which is due to charging/discharging of the circuit capacitances [1]. Hardware techniques attempt to minimize power by optimizing design parameters such as the supply voltage, the number of logic gates, the size of transistors and the operating frequency. Such decisions usually affect performance negatively. On the other hand, software

J. Blieberger and A. Strohmeier (Eds.): Ada-Europe 2002, LNCS 2361, pp. 65-75, 2002.

techniques primarily target at performing a given task using fewer instructions resulting in a reduction of the circuit switching activity. In this case, an improvement is achieved for both performance and power. Moreover, software methodologies normally address higher levels of the system design hierarchy, where the impact of design decisions is higher and the resulting energy savings significantly larger.

The increasing complexity and decreasing time-to-market of embedded software, forces the adoption of modular and reusable code, using for example object oriented techniques and languages such as C++ [3], [4]. The shift of the system functionality to the software domain enables greater flexibility in maintaining and updating an existing application. Object Oriented Programming (OOP), through features such as data abstraction and encapsulation of data and functions, is widely accepted as a methodology to improve modularity and reusability [5], [6]. Equally important, is the integration of hardware description languages and OOP programming languages into a common modeling platform. A promising example of this case is the enhancement of C++ with classes to describe hardware structures in SystemC [7].

In spite of its advantages, the acceptance of OOP in the embedded world has been very slow, since embedded software designers are reluctant to employ these techniques due to the additional performance overhead, in an environment with relatively limited computational power and memory resources. The introduced penalty on the system performance, in terms of execution time and memory overhead, has been demonstrated in the literature [8], [9], [10], [11]. This inherent drawback of object-oriented languages has forced the software community to develop sophisticated compilers, which attempt to optimize the performance of OOP [12], [13], [14]. An open standard defining a subset of C++ suitable for embedded applications has also been initiated [15].

Considering the intense need for low power, the purpose of this work is to investigate the effect of object oriented techniques compared to traditional procedural programming style, in an embedded environment, on both performance and power. Power exploration is not restricted to the processor but also considers the energy consumption of the instruction and data memories, whose power dissipation is a significant component of the total power in an embedded system. Since this is the first study of the power implications of object oriented programming, the aim here is not to evaluate existing compiler techniques in improving the performance of OOP but rather to show that OOP, if not applied properly, affects significantly not only the system performance but also its power consumption.

The target architecture that has been used for comparing object oriented programming style versus procedural programming is the ARM7 TDMI embedded processor core which is widely used in embedded applications due to its promising MIPS/mW performance [16]. Moreover it offers the advantage of an open architecture to the designer [17]. In order to evaluate both programming styles in terms of performance and power, the OOPACK benchmark kernels will be used a test vehicle [18].

The paper is organized as follows: Section II provides an overview of the sources of power consumption in an embedded system. Section III describes briefly the OOPACK benchmarks, while in Section IV the process that has been followed for the comparisons will be presented and the experimental results will be discussed. Finally, we conclude in Section V.

2 Sources of Power Consumption

The sources of power consumption in an embedded system, with varying importance according to the architecture and target application can be categorized as follows:

1. Processor power consumption, which is due to the operation of the processor circuitry during the execution of program instructions. This operation translates to switching activity at the nodes of the digital circuit, which in turn corresponds to charging/discharging the node capacitances, resulting in dynamic power dissipation [1]. To quantify this power component appropriate instruction-level power models have been developed. These models are based on the hypothesis that [19], it is possible by measuring the current drawn by a processor as it repeatedly executes certain instructions, to obtain most of the information required to evaluate the power cost of a program for that processor. This claim has been refined to state that the total energy cost cannot be calculated by the summation of the energy costs of the individual instructions [19], [20], [21]. It has been proved that the change in circuit state between consecutive instructions has to be taken into account in order to establish accurate instruction level power models.

The two basic components of an instruction power model therefore are:

a. *Base Energy Costs*: These are the costs that are associated with the basic processing required to execute an instruction. This cost is evaluated by measuring the average current drawn in a loop with several instances of this instruction. Some indicative base costs for several instruction types and addressing modes for the ARM7 processor core are shown in Table 1.

Type	Instruction	Addressing Mode	Base Cost (mA)
Arithmetic	ADD	LSL Immediate	9.92
	SUB	Immediate	6.67
	CMP	Immediate	6.65
	MOV	Immediate	8.07
Load/Store	LDR	Offset Immediate	10.76
	STR	Offset Immediate	8.55
Branch	B		8.73

Table 1. Base Costs for the ARM7 processor

b. *Overhead Costs*: These costs are due to the switching activity in the processor circuitry and the implied energy consumption overhead resulting from the execution of adjacent instructions. To measure the average current drawn in this case, sequences of alternating instructions are constructed. Some indicative overhead costs between pairs of instructions are shown in the matrix of Table 2, for the addressing modes of Table 1. Overhead costs between instructions of the same kind are significantly smaller.

Therefore, the total energy consumed by a program executing on a processor can be obtained as the sum of the total base costs and the total overhead costs. Thus, energy is given by

$$E_p = \sum_{i=1}^{n} I_{base_i} \times V \times N_i \times t + \sum_{i=2}^{n} I_{ovhd_{i,i-1}} \times V \times t \quad (1)$$

where I_{base_i} is the average current drawn by instruction #*i*, $I_{ovhd_{i,i-1}}$ the overhead cost for the sequence of instructions *i* and *i*-1, N_i the required number of clock cycles for instruction #*i*, *V* the supply voltage and *t* the clock period. For the results that will be shown next a supply voltage of 5 V and a clock speed of 20 MHz has been assumed.

	ADD	CMP	STR
SUB	1.24	0.13	2.42
MOV	1.35	1.10	2.64
LDR	3.29	2.77	0.80
B	1.25	1.03	2.00

Table 2. Overhead Costs (mA) for pairs of different instructions

2. Memory power consumption, which is associated with the energy cost for accessing instructions or data in the corresponding memories. Energy cost per access depends on the memory size and consequently power consumption for large off-chip memories is significantly larger than the power consumption of smaller on-chip memory layers. This component of the total power consumption is related also to the application: The instruction memory energy consumption depends on the code size, which determines the size of the memory and on the number of executed instructions that correspond to instruction fetches from the memory. The energy consumption of the data memory depends on the amount of data that are being processed by the application and on whether the application is data-intensive, that is whether data are often being accessed. For a typical power model the power consumed due to accesses to a memory layer *i*, is directly proportional to the number of accesses, f_i, and depends on the size, S_i, and the number of ports, Nr_ports_i, of the memory, the power supply and the technology. For a given technology and power supply the consumed energy can be expressed as:

$$E_i = f_i \cdot F\left(S_i, Nr\_ports_i\right) \tag{2}$$

The relation between memory power and memory size is between linear and logarithmic.

An example of a power optimizing approach for data-dominated applications such as multimedia algorithms, is the Data Transfer and Storage Exploration (DTSE) methodology [22] which aims at moving data accesses from background memories to smaller foreground memory blocks, which are less power costly, resulting in significant power savings.

3. Interconnect power consumption, which is due to the switching of the large parasitic capacitances of the interconnect lines connecting the processor to the instruction and data memories. This source of power consumption will not be explored in this study, however, since it depends on the number of data being transferred on the interconnect, it can be considered that a larger number of accesses to the instruction and data memory will result in higher interconnect energy dissipation. Several power reduction techniques have been developed for addressing this source of power consumption of which most important are appropriate data encoding schemes that minimize the average switching activity on the interconnect busses [2].

3 OOPACK Benchmarks

OOPACK is a small suite of kernels [18] that compares the relative performance of object oriented programming in C++ versus plain C-style code compiled in C++. All of the tests are written so that a compiler can, in principle, transform the OOP code into the C-style code. Although the style of object-oriented programming tested is fairly narrow, employing small objects to represent abstract data types, the range of applications to which they are used justifies the performance and power exploration. The four kernels for OOPACK are :

- **Max:** measures how well a compiler inlines a simple conditional.
- **Matrix:** measures how well a compiler propagates constants and hoists simple invariants
- **Iterator:** measures how well a compiler inlines short-lived small objects
- **Complex:** measures how well a compiler eliminates temporaries

The above benchmarks have some desirable characteristics as outlined in [14]: They allow measurements of individual optimizations implemented in the compiler, performance is tested for commonly used language features and are representative of widely used applications (for example matrix multiplication is common in embedded DSP applications).

The *Max* benchmark (OOPACK_1) uses a function in both C and OOP style to compute the maximum over a vector. The C-style version performs the comparison operation between two elements explicitly, while the OOP version performs the comparison by calling an inline function. This benchmark aims to investigate whether inline functions within conditional statements are compiled efficiently.

The *Matrix* benchmark (OOPACK_2) multiplies two matrices containing real numbers to evaluate the efficiency of performing two classical optimizations on the indexing calculations: invariant hoisting and strength-reduction. C-style code performs the multiplication in the following manner :

```
for( i=0; i<L; i++ )
  for( j=0; j<L; j++ )
  {
    sum = 0;
    for( k=0; k<L; k++ )
      sum += C[L*i+k]*D[L*k+j];
    E[L*i+j] = sum;
  }
```

where, for example, the term L*i is constant for each iteration of k and should be computed as an invariant outside the k loop. Modern C compilers are good enough at this sort of optimization for scalars and programmers do not have to bother doing the optimization by hand. However, in OOP style, invariants and strength reduction often concern members of objects. Optimizers that do not peer into objects miss the opportunities. In the above example, the OOP version performs the multiplication

employing member functions and overloading to access an element, given the row and the column.

The *Iterator* benchmark (OOPACK_3) computes a dot-product using a common single index in the C-style version and using iterators for the OOP-version. Iterators are a common abstraction in object-oriented programming, enabling the management of a collection class without the client program caring about the underlying structure of the collection. Although iterators are usually called "light-weight" objects, they may incur a high cost if compiled inefficiently. In this benchmark all methods of the iterator are inline and in principle correspond exactly to the C-style code. It has to be noted that the OOP-style code uses two iterators, and good common-subexpression elimination should be expected to reduce the two iterators to a single index variable.

Complex numbers are a common abstraction in scientific programming. The purpose of the *Complex* benchmark (OOPACK_4) is to measure the efficiency of C++ in handling complex arithmetic by multiplying the elements of two arrays containing complex numbers (defined with a class). In C-style the calculation is performed by explicitly writing out the real and imaginary parts while in OOP-style complex addition and multiplication is done using overloaded operations. The complex arithmetic is all inlined in the OOP-style, so in principle the code should run as fast as the version using explicit real and imaginary parts.

4 Results and Discussion

The process that has been set up in order to evaluate each kernel in terms of performance and power is shown in Fig. 1. Each OOPACK code was compiled using the C++ compiler of the ARM Software Development Toolkit v2.50 [17], which provided both the code size and the minimum RAM requirements for the data of each kernel. Next, the execution of the code using the ARM Debugger provided the number of executed assembly instructions as well as the total number of cycles. The ARM Debugger was set to produce a trace file logging instructions and memory accesses. It should be noted that the ARM C++ compiler implements most basic optimizations such as common subexpression elimination, loop invariant motion, live range splitting, constant folding, tail-calling and branch elimination [17].

The trace file is then parsed serially by a separate profiler that has been developed in *C* language, in order to collect information concerning the executed instructions and to obtain the number of data memory accesses. The parser has built-in look-up tables containing physical measurements [23] of the base and overhead energy costs in mA, for all types of instructions and instruction pairs. In this way it is possible by counting all instruction occurrences and assigning to them a base and an overhead energy cost according to the instruction type and addressing mode, to obtain the total energy cost for the processor.

Finally, the number of executed instructions and the code size are used as input to a memory power model in order to calculate the energy consumption of the instruction memory. In the same way, the number of data memory accesses and the minimum RAM size are used to compute the energy consumption of the data memory.

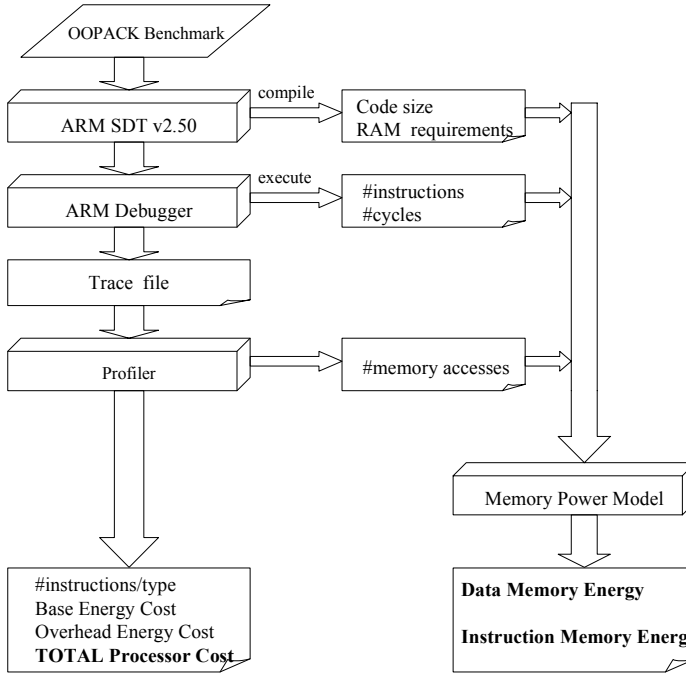

Fig. 1: Experiment set up for evaluating performance and power

Experimental results concerning the code size of each kernel, the number of executed instructions and cycles are given in Table 3 for all OOPACK kernels. As it can be observed, the OOP programming style has a larger impact on the resulting code size than on the number of executed instructions. This is reasonable, since the use of objects increases significantly the code size through the definition of classes, however runtime is not drastically increased mainly due to the use of inline methods. Code size refers only to the kernel size, excluding library functions, since the aim is to illustrate the effect of OOP on the programming style and its consequences. Whether the performance penalty, which can be up to 18%, is considered significant or not, depends on the application. In any case, the results are in agreement with previous studies and clearly demonstrate the so-called abstraction penalty [11] when writing object-oriented code.

Results concerning the required data memory size and the number of data memory accesses are given in Table 4. The RAM (this could be any type of random access memory) size is the same for both programming styles since the read-write data are not altered. For example, in the OOPACK3 kernel, the data memory size corresponds to two tables of *double* with 1000 elements plus a global *double* variable, for both C-style and OOP-style. The number of memory accesses refers only to the benchmark kernel and consequently it reflects the increased data transfers when abstract data types are used, probably due to inefficient use of registers. This is consistent with the

observation in [10] that one of the most striking differences between C and C++, is that C++ programs issue more loads and stores than C programs.

Benchmark	code size (bytes)	Instructions	Cycles
OOPACK1_c	180	50536	77118
OOPACK1_oop	212	56032	91605
OOP Penalty	*17.78 %*	*10.88 %*	*18.79 %*
OOPACK2_c	308	5402229	8303851
OOPACK2_oop	424	5625529	9051974
OOP Penalty	*37.66 %*	*4.13 %*	*9.00 %*
OOPACK3_c	260	433042	635096
OOPACK3_oop	356	450049	677103
OOP Penalty	*36.92 %*	*3.93 %*	*6.61 %*
OOPACK4_c	620	1041241	1606642
OOPACK4_oop	804	1084256	1710665
OOP Penalty	*29.68 %*	*4.13 %*	*6.47 %*

Table 3: Performance comparison between C_style and OOP_style for all kernels

Benchmark	RAM size (bytes)	Mem_accesses
OOPACK1_c	8008	8043
OOPACK1_oop		16035
OOP Penalty		*99.37 %*
OOPACK2_c	21600	1226765
OOPACK2_oop		1555328
OOP Penalty		*26.78 %*
OOPACK3_c	16008	79063
OOPACK3_oop		95063
OOP Penalty		*20.24 %*
OOPACK4_c	32000	256992
OOPACK4_oop		304996
OOP Penalty		*18.68 %*

Table 4: Memory comparison between C_style and OOP_style for all kernels

From a power perspective, this increases energy dissipation even further since according to the physical measurements [23] base and overhead costs for Load/Store instructions are slightly higher than for other instructions. Moreover, increased number of loads/stores results in more data memory accesses, which can be very power consuming when large data memory sizes are used.

In Table 5 the energy that has been calculated using instruction level and memory power models is presented for all system components that have been considered. For the programs under study, the most energy consuming system component is the processor. The overall energy overhead might not be critical for general purpose applications when performance and power constraints are relaxed, but should certainly affect the decision whether to use object-oriented code, when designing high-performance and low power systems, such as portable multimedia processing units.

Benchmark	Processor	Instr. Memory	Data Memory	System
OOPACK1_c	0.220	0.0181	0.0287	0.267
OOPACK1_oop	0.253	0.0206	0.0572	0.331
OOPACK2_c	18.148	2.234	6.882	27.264
OOPACK2_oop	19.534	2.406	8.726	30.666
OOPACK3_c	1.272	0.176	0.388	1.836
OOPACK3_oop	1.382	0.189	0.466	2.037
OOPACK4_c	3.353	0.472	1.724	5.549
OOPACK4_oop	3.632	0.517	2.046	6.195
Avg. OOP Penalty	*9.90 %*	*9.61 %*	*41.22 %*	*14.76 %*

Table 5: Comparison of energy consumption for all system components (in mJ)

It should be mentioned that the relatively large differences in code size between C-style and OOP-style are partially reflected in the instruction memory energy results due the extremely small-sized applications that have been selected. For real applications, with larger code size a significant increase of the instruction memory energy for the OOP style should be expected, however it should be noted that power dissipation increases sub-linearly with memory size. The results are shown in graphical form in Fig. 2, to provide an overview of each system component contribution to the total energy consumption.

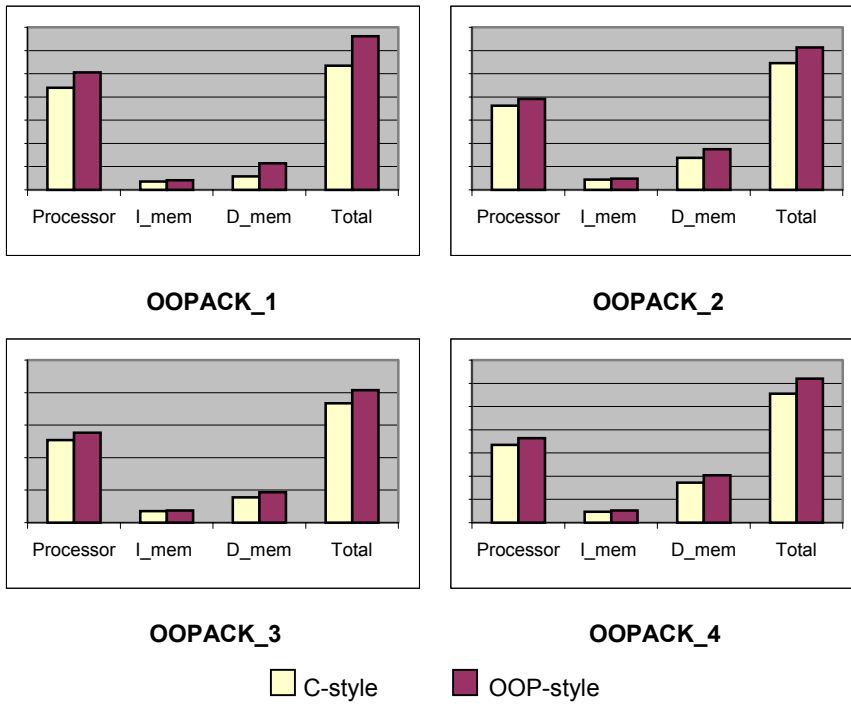

Fig.2 : Energy comparison between C-style and OOP-style for all system components

5 Conclusions

Object-oriented programming is widely accepted as a methodology for writing modular and reusable code. In embedded applications however, designers should consider when taking hardware/software decisions, the performance penalty that is introduced by the use of object-oriented code. In this paper, it has been demonstrated, through the compilation and execution of benchmarks on an embedded processor simulator, that OOP can result in a significant increase of both execution time and power consumption. In embedded systems where low power operation is the primary requirement, object oriented techniques can result in an energy dissipation overhead in all system components such as the processor core, the instruction and data memories. Since compilers usually cannot optimize code to reach the level of procedural programming performance, the number of executed instructions increases, increasing proportionally the instruction level power consumption. Moreover, care should be taken when opting for object oriented style, especially in large programs, since data abstraction can lead to a large code size increase resulting in a significantly higher power consumption of the instruction memory.

References

1. Chandrakasan A. and Brodersen R.: Low Power Digital CMOS Design. Kluwer Academic Publishers, Boston (1995)
2. Benini L. and De Micheli G.: System-Level Power Optimization: Techniques and Tools. ACM Transactions on Design Automation of Electronic Systems, vol. 5, (2000) 115-192
3. Cockx A. J.: Whole program compilation for embedded software: the ADSL experiment. 9th International Symposium on Hardware/Software Codesign (CODES'2001), Copenhagem, Denmark, (2001)
4. Plauger P.J.: Embedded C++. Embedded Systems Conference (ESC'99), Chicago, Illinois, USA, (1999)
5. Sommerville I.: Software Engineering. Addison-Wesley, Harlow (1995)
6. Harrison R., Samaraweera L.G., Dobie M.R., and Lewis P.H.: Comparing Programming Paradigms: an Evaluation of Functional and Object-Oriented Programs. Software Engineering Journal, vol. 11, (1996) 247-254
7. SystemC Homepage, http://www.systemc.org
8. Haney S.W.: Is C++ Fast Enough for Scientific Computing?. Computers in Physics, vol. 8, (1994) 690-694
9. Robinson A.D.: C++ Gets Faster for Scientific Computing. Computers in Physics, vol. 10, (1996) 458-462
10. Calder B., Grunwald D., and Zorn B.: Quantifying Behavioral Differences Between C and C++ Programs. Journal of Programming Languages, vol. 2, (1994) 313–351
11. Robinson A.D.: The abstraction penalty for small objects in C++. Parallel Object-Oriented Methods and Applications Conference (POOMA'96), Santa Fe, New Mexico, (1996)
12. Collin S., Colnet D. and Zendra O.: Type Inference for Late Binding. The SmallEiffel Compiler. JMLC'97, Linz, Austria, (1997), pp. 67-81
13. Zendra O., Colnet D. and Collin S.: Efficient Dynamic Dispatch without Virtual Function Tables. The SmallEiffel Compiler. OOPSLA'97, Atlanta GA, USA, (1997), pp. 125-141
14. Rotithor H., Harris K., and Davis M.: Measurement and Analysis of C and C++ Performance. Digital Technical Journal, vol. 10, (1999) 32-47
15. Embedded C++ Homepage, http://www.caravan.net/ec2plus
16. Furber S.: ARM System-on-Chip Architecture. Addison-Wesley, Harlow (2000)
17. ARM software development toolkit, v2.50, Copyright 1995-98, Advanced RISC Machines.
18. Kuck & Associates (KAI): C++ Benchmarks, Comparing Performance. http://www.kai.com/C_plus_plus/benchmarks/_index.html
19. Tiwari V., Malik S. and Wolfe A.: Power Analysis of Embedded Software: A First Step Towards Software Power Minimization. IEEE Transactions on VLSI Systems, vol. 2, (1994), 437-445
20. Tiwari V., Malik S., Wolfe A. and Lee T.C.: Instruction Level Power Analysis and Optimization of Software. Journal of VLSI Signal Processing, vol. 13, (1996) 1-18
21. Sinevriotis G. and Stouraitis Th.: Power Analysis of the ARM 7 Embedded Microprocessor. 9th Int. Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS'99), Kos, Greece, (1999), pp. 261-270
22. Catthoor F., Wuytack S., De Greef E., Balasa F., Nachtergaele L, Vandecappelle A.: Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design. Kluwer Academic Publishers, Boston (1998)
23. Sinevriotis G. and Stouraitis Th.: SOFLOPO: Low Power Software Development for Embedded Applications. Public Final Report, European Commission, ESD Best Practice: Pilot Action for Low Power Design, (2001)