ELSEVIER

Information and Software Technology 45 (2003) 195–201

INFORMATION AND SOFTWARE TECHNOLOGY

www.elsevier.com/locate/infsof

Performance and power evaluation of C++ object-oriented programming in embedded processors

Alexander Chatzigeorgiou*

Department of Applied Informatics, University of Macedonia, 156 Egnatia Str., 54006 Thessaloniki, Greece

Received 29 October 2001; revised 13 November 2002; accepted 14 November 2002

Abstract

The development of high-performance and lower 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. One way to confront the increasing complexity and decreasing time-to-market of embedded software is by means of modular and reusable code, forcing software designers to use objected oriented programming languages such as C++ [6]. 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. Profiling results indicate that C++ programs apart from being slower than their corresponding C versions, consume significantly more energy. Further analysis shows that this is mainly due to the increased instruction count, larger code size and increased number of accesses to the data memory for the object-oriented versions.
© 2003 Elsevier Science B.V. All rights reserved.

Keywords: Embedded systems; Object-oriented programming; Low power design

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 [5]. 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 is to reduce the dynamic power dissipation, which is due to charging/discharging of the circuit capacitances [3,5]. 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 hardware optimizations usually affect performance negatively. The software techniques that have been developed for reducing power dissipation 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 at system level may be higher and the resulting energy savings may also be 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++ [6,16]. 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 [12,24]. Equally important, is the integration of hardware description

* Tel.: +30-2310891886; fax: +30-2310891875.
E-mail address: achat@uom.gr (A. Chatzigeorgiou).

0950-5849/03/$ - see front matter © 2003 Elsevier Science B.V. All rights reserved.
PII: S0950-5849(02)00205-7

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 [25].

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 Refs. [4,8,11,18]. This inherent drawback of object-oriented languages has forced the software community to develop sophisticated compilers which attempt to optimize the performance of OOP [1,7,13,20,29]. An open standard defining a subset of C++ suitable for embedded applications has also been initiated [9].

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. The proposed power exploration methodology 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, to the author's best knowledge, 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 applied without considering energy issues, can affect 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 [10]. Moreover, it offers the advantage of an open architecture to the designer. In order to evaluate both programming styles in terms of power-performance, the OOPACK benchmark kernels [14] and some well-known algorithms [17] will be used as test vehicle.

The paper is organized as follows: Section 2 provides an overview of the sources of power consumption in an embedded system. Section 3 describes briefly the OOPACK benchmarks, while in Section 4 the process that has been followed for the comparisons will be presented and the experimental results will be discussed. Finally, we conclude in Section 5.

2. Sources of power consumption

There are mainly three sources of power consumption in an embedded system, with varying importance according to the architecture and target application. These are the processor, memory and interconnect of the system. The nature of these sources has been extensively studied during the last years [5] and modeling techniques to quantify their contribution have been developed. A brief discussion on each of the sources follows:

1. *Processor power consumption*, 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 [5]. To quantify this power component appropriate instruction-level power models have been developed. These models are based on the hypothesis that, it is possible [26] 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 [22,26,27]. It has been proved that the change in circuit state between consecutive instructions also 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. The overall range of the current that is being drawn from the power supply by any one instruction is between 5.55 and 11.61 mA

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

Table 1
Base costs for the ARM7 processor

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 2
Overhead costs (mA) for pairs of different instructions

	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

the total base costs and the total overhead costs. The energy that is dissipated in a time interval T is given by Ref. [5]:

$$E = \int_0^T P(t)\mathrm{d}t = V\int_0^T I(t)\mathrm{d}t \quad (1)$$

where $P(t)$ and $I(t)$ is the instantaneous power and current, respectively, while V is the supply voltage.

However, since the physical measurements that have been performed refer to the average current measured for each clock cycle [22], the integral diminishes to a product of the current (I_i) and the clock period (t) times the required number of clock cycles (N_i) for instruction #i. Thus, energy dissipation is calculated as:

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

where the sum accounts for all instructions in a program. I_{base_i} is the average current drawn by instruction #i and $I_{ovhd_{i,i-1}}$ the overhead cost for the sequence of instructions i and $i-1$ (which is not related to the number of cycles of any instruction).

2. *Memory power consumption*, 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 is 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, the number of ports of the memory, the power supply and the technology. For a given technology, power supply and number of ports, the consumed energy can be expressed as:

$$E_i = f_i \cdot F(S_i) \quad (3)$$

The relation between memory power and memory size is between linear and logarithmic [28]. According to Ref. [15], the capacitance that is being switched in a memory module, which in turn determines the dissipated energy per access, is a polynomial of the number of bits and the number of words in the memory array.

3. *Interconnect power consumption*, 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. According to a recent study [21] the interconnect energy consumption is around 7% of the total system energy.

3. OOPACK benchmarks

OOPACK is a small suite of kernels [14] 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 Ref. [20]: 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 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 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 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 'lightweight' objects, they may incur a high cost if compiled inefficiently. In the above 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 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 [2], 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.

The trace file is then parsed by the profiler, in order to obtain the number of data memory accesses. The profiler that has been developed for this study, implements a parser that 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 we obtain the total energy cost for the processor by counting all instruction occurrences and by assigning them a base and an overhead energy cost that depends upon the instruction type and the addressing mode.

Finally, the number of executed instructions and the code size is used as input to a memory power model (developed by an industrial vendor), 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.

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, for the particular examples, the OO 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

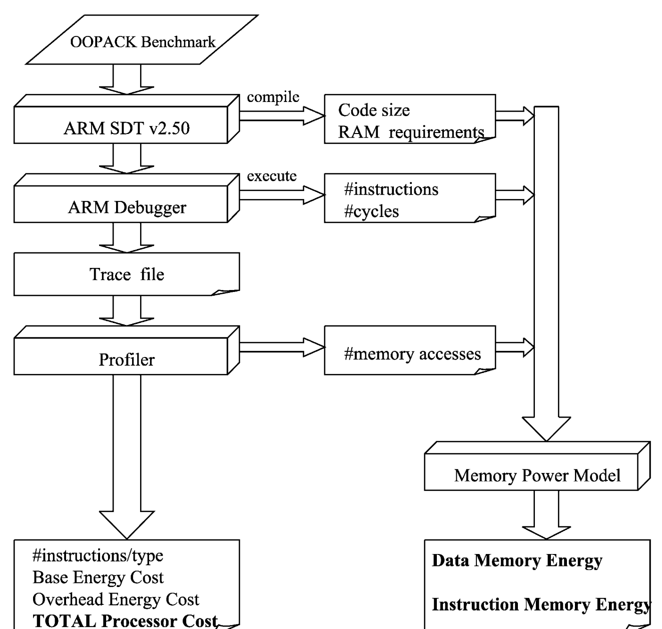

Fig. 1. Experiment set up for evaluating performance and power.

Table 3
Performance comparison between C_style and OOP_style for all kernels

Benchmark	Code size (bytes)	Instructions	Cycles
Max_c	180	50,536	77,118
Max_oop	212	56,032	91,605
OOP Penalty	*17.78%*	*10.88%*	*18.79%*
Matrix_c	308	5,402,229	8,303,851
Matrix_oop	424	5,625,529	9,051,974
OOP Penalty	*37.66%*	*4.13%*	*9.00%*
Iterator_c	260	433,042	635,096
Iterator_oop	356	450,049	677,103
OOP Penalty	*36.92%*	*3.93%*	*6.61%*
Complex_c	620	1,041,241	1,606,642
Complex_oop	804	1,084,256	1,710,665
OOP Penalty	*29.68%*	*4.13%*	*6.47%*

and because the examples have been selected to exercise object-oriented techniques. 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 [19] when writing object-oriented code. At this point, it is worth mentioning that the object-oriented paradigm can achieve code savings if the application offers possibilities for reuse. The OOPACK benchmarks that have been discussed, being narrow in the number of performed tasks, offer no opportunities for reusing code. However, the software of embedded systems, due to their limited resources, implements usually a single functionality and consequently there is limited room for exploiting the reuse advantages of object-oriented design.

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 almost the same for both programming styles since the read-write data on which the programs operate (vectors and matrices) are not altered. For example, in the *Iterator* kernel, the data memory size is dominated by the two arrays of *double* with 1000 elements, for both C-style and OOP-style. The differences in memory size between the C and OOP-style for the *Matrix* and *Iterator* benchmarks are due to multiple class instantiations, each one requiring its own attributes (e.g. indices). 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 Ref. [4] that one of the most striking differences between C and C++, is that C++ programs issue more loads and stores than C programs. From a power consumption point of view, this effect 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.

Table 4
Memory comparison between C_style and OOP_style for all kernels

Benchmark	RAM size (bytes)	Mem_accesses
Max_c	8020	8043
Max_oop	8020	16,035
OOP penalty		*99.37%*
Matrix_c	21,620	1,226,765
Matrix_oop	21,656	1,555,328
OOP penalty		*26.78%*
Iterator_c	16,020	79,063
Iterator_oop	16,048	95,063
OOP penalty		*20.24%*
Complex_c	32,020	256,992
Complex_oop	32,020	304,996
OOP penalty		*18.68%*

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

Benchmark	Processor	Instr. memory	Data memory	System
Max_c	0.220	0.0181	0.0287	0.267
Max_oop	0.253	0.0206	0.0573	0.331
Matrix_c	18.148	2.234	6.886	27.264
Matrix_oop	19.534	2.406	8.736	30.666
Iterator_c	1.272	0.176	0.388	1.836
Iterator_oop	1.382	0.189	0.467	2.037
Complex_c	3.353	0.472	1.725	5.549
Complex_oop	3.632	0.517	2.047	6.195
Avg. OOP penalty	*9.90%*	*9.61%*	*41.39%*	*14.76%*

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. The average OO penalty is the mean value of the percentage by which energy is increased for each of the OOPACK benchmarks. 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.

To extend our analysis to programs not especially written for object-oriented applications, Tables 6 and 7 present experimental results for the Gauss–Jordan elimination algorithm with full pivoting (for a 2×2 set of linear equations), for integration of a simple function employing the trapezoidal rule and for the QuickSearch algorithm [17]. All algorithms have been implemented in C and C++. (The data memory energy is negligible for these programs). To

Table 6
Performance comparisons of Gauss–Jordan, integral calculation and QuickSort algorithms

Benchmark	Code size (bytes)	Instructions	Cycles	No. of data mem. accesses
GaussJ_c	1096	2092	3485	496
GaussJ_oop	1184	2712	5056	968
OOP penalty	*8.03%*	*29.64%*	*45.08%*	*95.16%*
Integral_c	776	4672	6807	467
Integral_oop	912	10,386	17,745	3247
OOP penalty	*17.53%*	*122.30%*	*160.69%*	*595.29%*
QuickSort_c	508	4365	7610	763
QuickSort_oop	908	6439	13,373	3557
OOP penalty	*78.74%*	*47.51%*	*75.73%*	*366. 19%*

Table 7
Energy comparison of Gauss–Jordan, integral calculation and QuickSort algorithms

Benchmark	Processor energy (mJ)	Instruction mem. energy (mJ)	Total system energy (mJ)
GaussJ_c	0.00802	0.001892	0.009912
GaussJ_oop	0.01154	0.002487	0.014027
OOP penalty	*43.89%*	*31.45%*	*41.52%*
Integral_c	0.01504	0.003789	0.018829
Integral_oop	0.04034	0.008734	0.049074
OOP penalty	*168.22%*	*130.51%*	*160.63%*
QuickSort_c	0.01706	0.003287	0.020347
QuickSort_oop	0.03200	0.005415	0.037415
OOP penalty	*87.57%*	*64.74%*	*83.88%*

summarize our findings from these applications and the OOPACK benchmarks, the following observations can be made: object-orientation increases both the number of executed instructions as well as the number of accesses to the data memory. The code, which for embedded systems applications cannot take advantage of reuse possibilities, also increases for the OOP style. In terms of energy consumption, the effect of the instruction count on both the processor and the instruction memory power and the significant increase of data memory accesses, introduces a significantly penalty to the total system power for OO programs.

5. Conclusions

Although object-oriented programming is gaining increased acceptance, embedded system designers should consider 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 one of the primary requirements, object oriented techniques can result in an energy dissipation overhead in all system components such as the processor core, the instruction and data memories. According to experimental results for the ARM7 processor, object-oriented programming can increase the number of executed instructions as well as the number of memory accesses, increasing proportionally the instruction level and memory power consumption. Moreover, since reuse possibilities of OOP can often not be exploited in embedded software, object-orientation can also increase the code size, contributing even more to larger energy dissipation.

Acknowledgements

The author would like to thank the anonymous reviewers for their constructive remarks, which have helped to improve this paper.

References

[1] G. Aigner, U. Hoelzle, Eliminating virtual function calls in C++ programs, 10th European Conference on Object-Oriented Programming (ECOOP'96), Linz, Austria (1996).
[2] ARM software development toolkit, v2.50, Copyright 1995–98, Advanced RISC Machines.
[3] L. Benini, G. De Micheli, System-level power optimization: techniques and tools, ACM Transactions on Design Automation of Electronic Systems 5 (2000) 115–192.
[4] B. Calder, D. Grunwald, B. Zorn, Quantifying behavioral differences between C and C++ programs, Journal of Programming Languages 2 (1994) 313–351.
[5] A. Chandrakasan, R. Brodersen, Low Power Digital CMOS Design, Kluwer, Boston, 1995.
[6] A.J. Cockx, Whole program compilation for embedded software: the ADSL experiment, Ninth International Symposium on Hardware/Software Codesign (CODES'2001), Copenhagen, Denmark (2001) 2001.
[7] A. Diwan, K.S. McKinley, J.E.B. Moss, Using types to analyze and optimize object-oriented programs, ACM Transactions on Programming Languages and Systems 23 (2001) 30–72.
[8] K. Driesen, U. Hölzle, The direct cost of virtual function calls in C++, 11th Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'96), San Jose, CA, USA, 1996, pp. 306–323.
[9] Embedded C++ Homepage, http://www.caravan.net/ec2plus.
[10] S. Furber, ARM System-on-Chip Architecture, Addison-Wesley, Harlow, 2000.
[11] S.W. Haney, Is C++ fast enough for scientific computing?, Computers in Physics 8 (1994) 690–694.
[12] R. Harrison, L.G. Samaraweera, M.R. Dobie, P.H. Lewis, Comparing programming paradigms: an evaluation of functional and object-oriented programs, Software Engineering Journal 11 (1996) 247–254.
[13] U. Hölzle, O. Agesen, Dynamic vs. static optimization techniques for object-oriented languages, Theory and Practice of Object Systems 1 (1996) 167–188.
[14] Kuck and Associates (KAI), C++ Benchmarks, Comparing Performance, http://www.kai.com/C_plus_plus/benchmarks/_index.html.

[15] P.E. Landman, J.M. Rabaey, Architectural power analysis: the dual bit type method, IEEE Transactions on VLSI Systems 3 (1995) 173–187.

[16] P.J. Plauger, Embedded C++, Embedded Systems Conference (ESC'99), Chicago, Illinois, USA (1999).

[17] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, Numerical Recipes in C: the Art of Scientific Computing, Cambridge University Press, Cambridge, 2002.

[18] A.D. Robinson, C++ gets faster for scientific computing, Computers in Physics 10 (1996) 458–462.

[19] A.D. Robinson, The abstraction penalty for small objects in C++, Parallel Object-Oriented Methods and Applications Conference (POOMA'96), Santa Fe, New Mexico, 1996.

[20] H. Rotithor, K. Harris, M. Davis, Measurement and analysis of C and C++ performance, Digital Technical Journal 10 (1999) 32–47.

[21] T. Šimunić, L. Benini, G. De Micheli, Energy-efficient design of battery-powered embedded systems, IEEE Transactions on VLSI Systems 9 (2001) 15–28.

[22] G. Sinevriotis, Th. Stouraitis, The power analysis of the ARM 7 embedded microprocessor, Ninth International on Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS'99), Kos, Greece (1999) 261–270.

[23] G. Sinevriotis, Th. Stouraitis, SOFLOPO: low power software development for embedded applications, Public Final Report, European Commission, ESD Best Practice: Pilot Action for Low Power Design, 2001.

[24] I. Sommerville, Software Engineering, Addison-Wesley, Harlow, 1995.

[25] SystemC Homepage, http://www.systemc.org.

[26] V. Tiwari, S. Malik, A. Wolfe, Power analysis of embedded software: a first step towards software power minimization, IEEE Transactions on VLSI Systems 2 (1994) 437–445.

[27] V. Tiwari, S. Malik, A. Wolfe, T.C. Lee, Instruction level power analysis and optimization of software, Journal of VLSI Signal Processing 13 (1996) 1–18.

[28] S. Wuytack, J.Ph. Diguet, F. Catthoor, H. De Man, Formalized methodology for data reuse exploration for low-power hierarchical memory mappings, IEEE Transactions on VLSI Systems 6 (1998) 529–537.

[29] O. Zendra, D. Colnet, S. Collin, Efficient dynamic dispatch without virtual function tables. The SmallEiffel compiler, Object-Oriented Programming, Systems, Languages and Applications Conference (OOPSLA'97), Atlanta GA, USA (1997) 125–141.