Energy Consumption Estimation Technique in Embedded Processors with Stable Power Consumption based on Source-Code Operator Energy Figures

Juan Castillo[1], Héctor Posadas[1], Eugenio Villar[1] and Marcos Martínez[2]

[1]University of Cantabria
E.T.S.I. Industriales y Telecom.
Avda. Los Castros s/n, 39005 Santander, Spain
{castillo, posadash,villar} @teisa.unican.es

[2]DS2
Charles Robert Darwin 2
Parc Tecnológic, 46980 Paterna, Valencia, Spain
marcos.martinez@ds2.es

***Abstract* — Power consumption has become one of the main concerns in embedded system design. Currently, design platforms are composed of generic and specific hardware devices and general-purpose processors running the application software. SW functionality has a major impact on the total system power consumption. Early estimation of the power consumption of the application SW is crucial in order to take the correct design decisions as soon as possible. In this paper we exploit the 'almost' constant relationship between machine instructions executed per second and the corresponding power consumption found in many embedded processors. Based on this property, a technique to estimate the SW power consumption has been developed based on source code simulation. Short simulation times are achieved with high accuracy, so the technique can be applied at early stages of the design flow. Energy consumption of an ARM9TDMI core when running some typical application codes has been estimated[1].**

***Keywords* — power estimation, embedded software, source-code simulation, ARM**

I. Introduction

Power consumption is one of the most important constraints in embedded system design. Two main reasons motivate the constant effort of the designers to reduce consumption. First, in many cases, embedded systems are designed for mobile applications using a limited life-time battery. Second, thermal dissipation is constrained by cost, weight and size limits. As a consequence, working temperature must be kept low enough to allow cheap packaging and/or maintain system reliability above the required level.

Current techniques for embedded system design constantly seek new improvements for maximum reusability. One common solution is platform-based design, where system platforms contain generic and specific purpose hardware devices controlled by application software executed on general purpose microprocessors. The development of the software accounts for more than 80% of the total effort of the design system [1]. Energy efficiency of software is lower than hardware. As a consequence, software execution represents an important percentage of the total energy consumption of the system. A design can be rejected because the energy consumption needed to perform software tasks is excessive. This possible drawback justifies the necessity of simulations to estimate this consumption at earlier stages of the design flow.

Many RISC embedded processors exhibit 'almost' constant power consumption per executed machine instruction. This is the case of most ARM processors [6]. This is based not only on the fact that there are small differences in power consumption among the different instructions, but also considering that these differences compensate each other when a sufficiently high number of instructions are executed. In this paper a technique is proposed for power estimation of embedded software execution exploiting this general characteristic.

Energy estimation is performed by assigning an energy cost to each C++ operator, and overloading the application software operators to keep track of the total energy consumed. To do so, SystemC [11] is employed as modeling language to perform the dynamic simulation. This technique achieves short simulation times and avoids cross-compiling the source code to obtain the consumption figures. These figures are obtained directly as simulation results.

As a consequence, software engineers can obtain a way to make early consumption estimations of their application code. With these results, they can consider the possibility of changing algorithms, reducing code or even implementing the target functionality in specific hardware, with the intention of reducing energy consumption.

Throughout the paper, both concepts of power and energy consumption will be used, assuming they are correlated by execution time.

The structure of the paper is the following. Section 2 introduces the state-of-the- art in software energy estimation and shows the contributions of the proposed technique. Section 3 demonstrates how energy consumption can be estimated from the number of machine instructions executed. Section 4 presents an approach to map C++ operators to RISC machine instructions. Section 5 shows how to perform energy estimations once operator weights have been extracted. The ARM9TDMI core will be characterized in terms of energy in section 6 and consumption estimation of some common algorithms is performed. Conclusions and future work will be discussed in section 7.

II. State of the art

Power estimation has become a very important aspect of embedded system design. With shorter time-to-market and higher manufacturing costs, it is unacceptable that a full

[1] This work has been supported by the Spanish MICyT and MEC through the MEDEA+ LoMoSA project and the TEC2005-03301 project.

design is rejected because of excessive power consumption when expensive prototypes have already been assembled. This justifies the importance of this field in embedded system design, and the effort made in developing new techniques for power estimation.

Traditional methodologies perform power estimation at low abstraction levels such as gate or RT level [2][3]. This implies cross-compiling the source code, introducing the binary code in the memory model and simulating the SW functionality directly with a logic model of the processor. These methodologies are not useful for high-level power estimation, due to their long simulation time and the fact that the HW platform is not completely known at early stages of the design process.

A higher level approach has been proposed in [4], where different components of the embedded architecture are modeled at cycle-accurate level in terms of energy. Estimation errors are within 5%, but nonetheless, cycle-accurate simulations also require a great computational cost.

The next step in software energy estimation was done using the Instruction Set Architecture of the target core processor [5][6]. The full set of machine instructions is characterized in terms of energy consumption, assigning a base cost for each instruction and an overhead due to the state changes in the processor FSM and datapath. Application software must be compiled and assembled to perform the power estimation. Although faster than RTL, these techniques are still too slow to analyze complex systems.

Direct source-code analysis achieves shorter estimation times. Some methodologies have been proposed using this approach [7][8]. Nonetheless, they are usually based on specific models of processors, making the portability among different architectures more difficult. In [9], a source-code, SW simulation methodology using run-time execution time estimation is proposed showing the viability of dynamic SW analysis.

This paper proposes a general, portable strategy to estimate software consumption from source-code simulation. The technique exploits the characteristic of many embedded processors which exhibit statistically constant power consumption per executed instruction. Application software is dynamically analyzed, and energy estimation is performed by assigning an energy weight to each operator. These weights are extracted from the underlying architecture and the approximate number of machine instructions necessary to perform the corresponding operator. Short simulation times are achieved due to its low computational cost.

III. Source code energy estimation

The technique presented in this work performs source-code energy estimation from the operators' individual energies. The operator energy is obtained by adding the energy consumed by all machine instructions executed to perform the operation.

Assuming N is the total number of C operators and control statements, the total energy E_T necessary to execute the whole application code is:

$$E_T = \sum_{n=1}^{N} E_n \qquad (1)$$

where E_n is the total amount of energy consumed related to operator 'n'. E_n depends on how many times this operator is executed and the corresponding execution energy of each one.

In this work, operations with variables and operations with immediate values are considered different, so, in fact:

$$N = 2 * N_{op} + N_{ctr}$$

where Nop is the number of C operators, and Nctr is the number of C control statements. This will be explained in more detail in section 4.

The energy consumed can be different each time the operator is executed. Depending on the data location (memory - registers) or the data size, the number of assembler instructions used can be different.

If Mn is the number of executions of operator 'n' and Enm the energy for each single operator execution:

$$E_n = \sum_{m=1}^{Mn} E_{nm} \Rightarrow E_T = \sum_{n=1}^{N} \sum_{m=1}^{Mn} E_{nm} \quad (2)$$

The energy required by one operator in a single execution can be correlated with the number of machine instructions necessary to implement it. It can be calculated in the following way:

$$E_{nm} = \sum_{i=1}^{I_{nm}} E'_i \qquad (3)$$

where Inm is the number of machine instructions required to execute the operator 'n' in execution 'm' and E'_i the energy of the corresponding machine instruction.

Substituting (3) into (2), the result is:

$$E_T = \sum_{n=1}^{N} \sum_{m=1}^{M_n} \sum_{i=1}^{I_{nm}} E'_i \qquad (4)$$

Simplifying this approach, we can make use of the mean value:

$$\overline{E'_{nm}} = \frac{\sum_{i=1}^{I_{nm}} E'_i}{I_{nm}}$$

And therefore:

$$E_{nm} = \sum_{i=1}^{I_{nm}} E'_i = I_{nm} \cdot \overline{E_{nm}'}$$

Furthermore, in most embedded RISC processors, when the executed code is large enough, energy consumption per instruction can be considered constant without great loss of accuracy [6]. Therefore, if all Mn are high enough, E' can be assumed to be independent of 'm' and 'n':

$$\forall n \in (1, N)\ \exists M_{n\min} \ /\ \overline{E'_{nm}} \cong \overline{E'}$$
$$\forall m \in (1, M_n) \wedge \forall M_n > M_{n\min} \quad (5)$$

With this assumption, $\overline{E'}$ can be calculated from the processor datasheet information, independently of the specific design source code. The value of this parameter will be obtained in section 6.

As a consequence, in equation (4), the mean energy consumption per instruction can be extracted from the mean number of assembler instructions necessary to implement

the operator:

$$E_T \cong \sum_{n=1}^{N} \sum_{m=1}^{M_n} I_{nm} \cdot \overline{E'} = \overline{E'} \sum_{n=1}^{N} \sum_{m=1}^{M_n} I_{nm} \quad (6)$$

The number of instructions for each operator is architecture-dependent and it is the objective of the next section.

IV. Energy consumption of C++ operators in RISC machines

The energy estimation method for the source-code operators is shown in Figure 1:

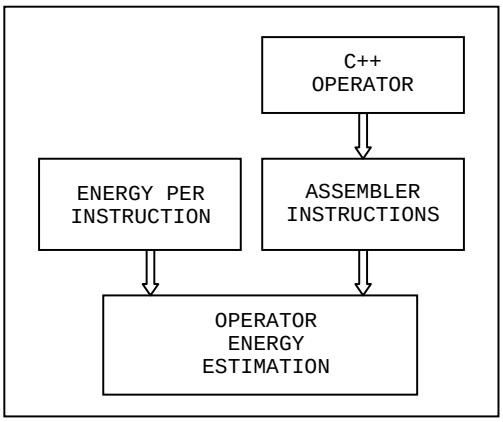

Figure 1

The estimation technique consists basically in extracting the number of machine instructions, which are necessary to execute each source code C operator. Although these values are architecture-dependent, the methodology proposed to extract them can be applied to all RISC processors. It is important to notice that the number of instructions can vary depending of several factors, such as the memory access mode or immediate value loads. The approximation applied is to consider a mean number of instructions $\overline{I_n}$ per C operator when the size of the code is large enough. This simplification has been proven in [12].

$$\forall n \in (1, N)\ \exists M_{n\min} \ /$$

$$I_{nm} \cong \overline{I_n} \forall m \in (1, M_n) \wedge M_n > M_{n\min}$$

Using this approach in (6), the equation becomes:

$$E_T \cong \overline{E'} \cdot \sum_{n=1}^{N} \sum_{m=1}^{M_n} I_{nm} \cong \overline{E'} \cdot \sum_{n=1}^{N} \sum_{m=1}^{M_n} \overline{I_n}$$

And therefore:

$$E_T \cong \overline{E'} \sum_{n=1}^{N} M_n \cdot \overline{I_n} \quad (7)$$

As a consequence, considering limitations in (6) and (7), the total energy consumption of a program can be calculated. Furthermore, an approximate mean energy cost ($\overline{E_n}$) can be defined for each operator independently of the SW code of each specific design. This value is the product of the mean number of assembler instructions executed by each operator, and the mean energy of an assembler instruction. Therefore, the new equation obtained is:

$$E_T \cong \overline{E'} \cdot \sum_{n=1}^{N} M_n \cdot \overline{I_n} = \sum_{n=1}^{N} M_n \cdot (\overline{I_n} \cdot \overline{E'})$$

$$\overline{E_n} \cong \overline{I_n} \cdot \overline{E'} \quad (8) \quad \Rightarrow E_T \cong \sum_{n=1}^{N} M_n \cdot \overline{E_n} \quad (9)$$

Based on this equation, a study must be made to correlate as closely as possible each C operator with the related, approximate number of machine instructions. C++ operators can be classified according to their hardware interaction. To do so, the C operators and control statements can be grouped in four types depending on their effect in the execution.

A. Operators that modify general-purpose registers

The first group considered is composed of those operators that modify the general purpose registers. The operators of this group are:

```
++   --   +( )   -( )   !( )   ~( )   =   +
-   *   /   %   +=   -=   *=   /=   %=   &   ^
|   &=   ^=   |=   <<   <<=   >>   >>=
```

In general, these operators can be used in three contexts:

(a). var1 *op* imm1;

(b). var1 *op* var2;

(c). var1 *op* var2 *op2* var3;

In the first case, the operator accepts one argument as a variable and the other as an immediate value. It requires a memory access to load ‘var1’ and other instructions to load the immediate value and execute the operation.

In the second case, both operators are variables. It requires two memory accesses (one for each operand) and one operation.

In the last case, one of the arguments is the result of other operation, not directly a variable. Considering that the result of previous operator is stored in a processor register, only assembler instructions for a memory access and the operation are required.

Memory accesses are always load accesses except for the ‘=’ operator, where one memory access is a store. The remaining operators that include ‘=’ together with another operation, also add a store instruction to the machine instructions described above.

Apart from that, other analysis is necessary. Depending on the immediate value size or the place where the variable is, the number of assembler instructions used can be different. Thus, the energy cost for a single operator in the code has several possible values.

When one operand is an immediate value, a study of the architecture is required. Most RISC processors include a field in the instruction word where the immediate value can be stored. Its width is architecture-dependent, generally 16-bit. There is an option to load both parts of the 32-bit register, so the two values can be stored. Other architectures perform this task in different ways.

When operands are variables, a study of memory access modes is necessary. Local variables are stored in the stack, and the most common access mode is a base plus an offset.

The base is a pointer to the top of the stack and the offset is the relative position of the current element. The access memory instruction word includes a field where immediate offsets are stored. If the current offset is greater than the maximum value of the field, a general-purpose register is employed; consequently a different number of machine instructions are required.

Nevertheless, as explained in section 3, only two energy costs are accepted for each operator, one for (a) and another one for cases (b) and (c). These costs are obtained before the simulation starts and are constant for all programs in the same platform. They are not obtained for each simulation but only once for a platform. To do so, an analysis of all possibilities is done. Then, weighted average values are calculated.

Thus, source-code energy estimations can have an error depending on the deviation between the weighted average value extracted previously and the actual $\overline{E}_n$ of the analyzed design.

B. Operators that modify state registers

These operators are usually associated with control statements. They are commonly used to modify the processor state registers, and then, decide the path to be executed. However, these operators can also be used to assign a value to a variable, so they can also modify general-purpose registers. This means that the approach used with the operands in the previous group has to be extended. The operands of this group are:

```
==   !=   >   <   >=   <=   &&   ||
```

In this case, operands are designed to directly modify the state register. If the result is used as an argument of another operator a general-purpose register also has to be modified. Thus, the state register value has to be analyzed to obtain the information required to assign this value. This implies more machine instructions than operators in the first group.

The operator energy cost obtaining methodology is similar. All cases are evaluated and weighted mean costs are calculated based on typical test programs.

C. Control execution flow

These operators decide the program execution flow based on previously modified state registers. The statements are:

```
'for' 'if' 'while' "jump to function"
"jump to member function" 'break'
'continue' 'return' 'do-while'
```

The steps to perform a conditional jump in a program execution are:

(1) Use the ALU to get the result of the comparison
(2) Update the state register
(3) Jump according to the state register value

RISC architectures usually group (1) and (2) in a single instruction and (3) is implemented as a conditional jump instruction.

Statements 'for', 'if', 'while' and 'do-while' contain conditional and unconditional jumps. In fact, 'do-while' can be modeled together with 'while' statement. 'Break' and 'continue' represent unconditional jumps, and are associated with 'for' or 'while' statements, so can be considered inside then.

Jump to function leads to an overhead in the execution flow, since it is necessary to save current PC, stack pointer, state registers and user registers. Jump to member function requires a more complex calculation of the destination jump address, so it has a different weight.

D. Array access

The operator '[ ]' performs array accesses. An array is a set of homogeneous objects stored consecutively in memory. An array access is performed reading a pointer to the first element and adding an offset to calculate the address of the desired element.

Architectures implement array accesses with different memory access modes. The most employed mode in RISC architectures is 'base plus offset', where the base address (the array pointer) is stored in a processor's register and the offset is expressed as an immediate value in the corresponding field of the instruction word. If the current element's offset is bigger than the maximum value of the field, additional instructions or different access modes are required. Since embedded software must consider memory size constraints, it is assumed that all memory accesses fit into the offset field.

Following these considerations, all $\overline{I}_n$ required to obtain the $\overline{E}_n$ parameters required in (9) can be obtained. An example will be presented in section VIII.

V. Energy estimation process

Once $\overline{I}_n$ is obtained, energy per assembler instruction is required to apply equation (8). However, it cannot usually be obtained directly from processor vendor information. This information usually only contains values for mean power dissipation, frequency or cycles per instruction. Thus, a new equation that allows us to estimate the mean energy per instruction using these figures is required.

To obtain this equation, first we can estimate the energy per cycle consumed in the processor, dividing the mean power and the frequency. This represents the mean energy of all machine instructions ($\overline{E''}_{ck}$) of a generic program. A generic program is one that is large enough and that uses all instructions in the same ratio as usual programs.

The energy consumed by the processor (E''_{ckj}) in a clock cycle 'j' can be considered as the addition of all pipeline stage energies:

$$E''_{ck_j} = \sum_{s=1}^{St} E''_{s_j} \quad ,$$

where St is the number of stages and E''_{sj} the energy consumption of stage 's' in cycle 'j'.

This means that if we know the mean energy per cycle, we can say that it can be considered as the addition of all mean pipeline stage energies in a generic program.

$$\overline{E''_{ck}} = \sum_{s=1}^{St} \overline{E''_{s}} \quad (10)$$

Apart from that, it is known that the energy of an instruction 'i' (E'_i) is:

$$E'_i = \sum_{s=1}^{St_i} E'_{s_i}$$

where St_j is the number of pipeline stages instruction 'i' uses ($St_j \leq St$), and E'_{si} the energy of stage 's' for instruction 'i'. Considering that the processor has St stages, the previous equation implies that remaining stages requires 0 energy.

$$E'_i = \sum_{s=1}^{St} E'_{s_i} / E_{s_j} = 0 \; \forall \; St_j < s \leq St$$

Furthermore, we have to take into account that an instruction can require more that one cycle in a single stage. Considering energy per stage, the result is:

$$E'_i = \sum_{s=1}^{St} E'_{s_i} = \sum_{s=1}^{St} \sum_{c=1}^{Ck_{si}} E'_{sc_i}$$

where Ck_{si} is the number of clock cycles of instruction 'i' in stage 's' and E'_{scj} energy of instruction 'j' in stage 's' and cycle 'c'.

However, to apply (8) means only energy per instruction is required. To obtain this, it can be assumed that there is a mean energy per stage and cycle, considering the mean number of clock cycles in a stage can be simplified using the processor efficiency Φ. Applying this, the mean energy can be computed as:

$$\overline{E'} = \sum_{s=1}^{St} \frac{\overline{E_s}}{\Phi} = \frac{1}{\Phi} \cdot \sum_{s=1}^{St} \overline{E_s} \quad (11)$$

where $\overline{E'_s}$ is the mean energy consumption of stage 's' for a generic program. Thus applying (10), the addition of all $\overline{E'_s}$ of St stages in a clock cycle is equal to $\overline{E''}$ for the same generic program. So finally, the mean energy of a machine instruction is:

$$\overline{E'} = \frac{1}{\Phi} \cdot \sum_{s=1}^{St} \overline{E_s} = \frac{\overline{E_{ck}}}{\Phi} = \frac{\overline{P}}{\Phi \cdot f} \quad (12)$$

VI. Energy estimation process

Once operators are characterized in terms of machine instructions, energy estimation can be performed. The proposed approach is to perform source-code dynamic energy estimation to apply (9). SystemC is used for this purpose.

Using SystemC, software engineers do not have to change the source code of their projects, since it is a C++ class library. Basically, application code is encapsulated in a SystemC module and executed to perform the energy estimation. Figure 2 shows the proposed approach.

However, standard SystemC presents some limitations for software modeling. The lack of Operating System facilities or execution time control can restrict the modeling capabilities. To solve this, an external SystemC library called PERFidiX [9] has been used.

Furthermore, specific hardware can be described in SystemC. PERFidiX provides OS facilities to communicate these components with the SW tasks. This advantage allows co-simulation of Hardware-Software (HW/SW). Since embedded software is strongly hardware dependent, this technique supposes an important advance in the design of embedded systems.

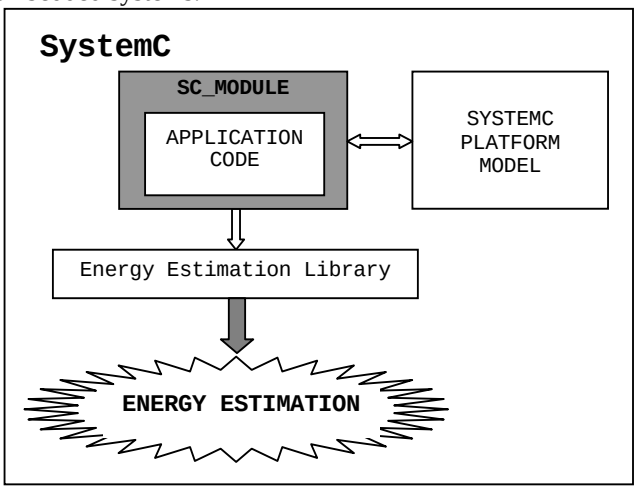

Figure 2

However, the main reason why PERFidiX has been used is that it also allows a dynamic analysis of source code. Thus, energy estimation is achieved by extending the C++ operators overloading of this library [12]. This overload performs a count of the energy consumed up to the current code line. To do so, each time an operator is executed, its corresponding mean energy cost is added to the total amount of energy of the current segment of code.

Using this solution, the total amount of energy of each code segment is obtained together with the corresponding execution time. Thus, the mean power required by each part of the code, and by the whole program can be easily estimated.

VII. Case study: ARM9TDMI core

To validate the power consumption estimation methodology, an ARM9TDMI core has been characterized.

Declared mean power consumption of this core for a 250 nm process, f = 200 MHz and Vdd 2.5 V, is $\overline{P}$ = 150 mW. An additional important parameter is the average number of instructions per clock cycle. In the microprocessor used, the efficiency is Φ = 0.67. These values are provided by ARM [10].

Thus, applying (12) to these parameters, for the ARM9TDMI, $\overline{E'}$ = 1.125 nJ / Instruction. The energy per operator can be obtained and test program energies can be estimated.

Tables 1 and 2 present energy costs for all N operators. They are calculated using the mean number of instructions and $\overline{E'}$ estimated above using (8). The mean number of instructions is calculated applying considerations in (7).

Division and modulus operators are not directly performed by the ALU in RISC processors; they require additional hardware or specific routines provided by the compiler. In this case, operations are performed by software. To estimate division and modulus operator consumptions, both mean execution times have been measured, thus calculating the equivalent machine instructions with the clock period.

Table 1 shows mean energy cost for each C operator when operated with variables or a variable and an immediate value.

Table 2 presents mean energy cost for control statements.

C++	E_n (nJ) (variab.)	E_n (nJ) (immediate)	C++	E_n (nJ) (variab.)	E_n (nJ) (immediate)
++	3.375	3.375	!=	3.375	2.25
--	3.375	3.375	>	3.375	2.25
+()	0	0	<	3.375	2.25
-()	1.125	1.125	<=	3.375	2.25
!()	3.375	3.375	>=	3.375	2.25
~()	1.125	1.125	&&	1.125	1.125
=	2.25	2.25	||	1.125	1.125
+	2.25	1.125	&	2.25	1.125
-	2.25	1.125	^	2.25	1.125
*	2.25	1.125	|	2.25	1.125
/	11.25	11.25	&=	3.375	3.375
%	33.75	33.75	^=	3.375	3.375
+=	3.375	3.375	|=	3.375	3.375
-=	3.375	3.375	<<	2.25	1.125
*=	3.375	3.375	<<=	3.375	3.375
/=	14.625	14.625	>>	2.25	1.125
%=	18	18	>>=	3.375	3.375

Table 1

Statement	E_n (nJ)
for	2.25
if	1.6875
while	2.25
array	6.75
jump to function	10.125
jump to member function	11.25

Table 2

With these costs, some common application algorithms have been simulated: bubble sort, FIR filter (Finite Impulse Response), several operations over array elements, a Fibonacci series generator and a Quicksort algorithm.

To validate the results obtained, the exact number of assembler instructions for each algorithm has been extracted from an open-source ARM ISS (Instruction Set Simulator): the arm-elf-gdb. Compilation has been done with GNU C++ compiler for ARM architecture: "arm-elf-g++".

Energy consumption of example programs are obtained and compared with the results obtained directly from source code. Table 3 shows the results for the algorithms mentioned before. We include the real number of instructions executed with the ISS, the equivalent energy consumption and the energy estimation obtained with the proposed methodology. Relative error is also reported. As can be seen, errors lower than 11% are achievable with this methodology.

Algorithm	Machine Instructions	Energy from ISS (uJ)	Estimated Energy (uJ)	Error (%)
Bubble Sort	11051	12.43	13.2	6.19
FIR	4294	4.83	5.34	10.56
Array	6413	7.21	6.88	4.58
Fibonacci	3705	4.17	4.05	2.88
Quicksort	24912	28.03	25.44	9.24

Table 3

VIII. Conclusions

New system designs require higher levels of abstraction due to their increasing complexity. Consequently, new estimation techniques for performance measurement are necessary, in accordance with these levels.

This paper describes a technique to estimate energy consumption of embedded processors while executing application software at the first stages of the design process. Source-code simulation is performed, achieving accurate results with short simulation times.

SystemC is selected as description language. Since it is a C++ class, it is possible to encapsulate the application code in a SystemC module and simulate it with a SystemC description of the execution platform.

Processor consumption can be characterized by associating an energy cost to each source code operator and control statement. These energy costs are extracted from the number of machine instructions necessary to execute the corresponding operator. This approach is valid in those processors in which consumption per executed machine instruction is almost constant when the executed code is large enough. Once the costs have been obtained, they can be directly reused for all designs developed on the same HW platform.

Consumption estimation can be done by overloading the operators. Operator consumptions are added when executed to obtain the total energy consumption of the SW. This task can be performed with a specific library or an update in an external tool. In this work, the PERFidiX tool [9] has been used to implement operator overloading for time and energy estimation, with a low development effort.

Future work will be centered on analyzing the impact of cache memories in energy consumption, since the current methodology only considers the processor core. Compiler optimizations must also be considered as they have a great impact on the quality of the assembler code generated.

References

[1] ITRS: "International Technology Roadmap for Semiconductors: 2005 Edition", http://www.itrs.net/Common/2005ITRS/Home2005.htm.

[2] T. H. Krodel, "PowerPlay – Fast Dynamic Power Estimation Based on Logic Simulation", 1991.

[3] F. N. Najm, "A Survey of Power Estimation Techniques in VLSI Circuits", IEEE: Transactions on VLSI Systems, 1994.

[4] T. Simunié, L. Benini, G. De Micheli, "Cycle-Accurate Simulation of Energy Consumption in Embedded Systems", DAC, 1999.

[5] V. Tiwari, S. Malik, A. Wolfe, "Power Analysis of Embedded Software: A First Step towards Software Power minimization", ACM Conference, 1994.

[6] A. Sinha, A.P. Chandrakasan, "Joule-Track, A Web Based Tool for Software Energy Profiling", DAC, 2001.

[7] J. Laurent, E. Senn, N. Julien, E. Martin, "Power Consumption Estimation of a C-algorithm: A New Perspective for Software Design", ACM LCR Conference, 2002.

[8] C. Brandolese, W. Fornaciari, L. Pomante, F. Salice, D. Sciuto, "A Multi-level Strategy for Software Power Estimation", XXX.

[9] H. Posadas, J. Adámez, E. Villar, Francisco Escuder (DS2), Francisco Blasco (DS2), "RTOS modeling in SystemC for Real-Time embedded SW simulation: A POSIX model", Design Automation for Embedded Systems, V.10, N.4, Springer, 2005.

[10] S. Furber, "ARM, System-on-chip Architecture. 2nd Ed", Addison-Wesley, 2000.

[11] www.systemc.org.

[12] H. Posadas, F. Herrera, P. Sánchez, E. Villar and F. Blasco: "System-level performance analysis in SystemC", in Proc of

DATE, 2004