Software Quality Journal, 10, 355–371, 2002
© 2003 Kluwer Academic Publishers, Manufactured in The Netherlands.

Energy Metric for Software Systems

ALEXANDER CHATZIGEORGIOU
AND GEORGE STEPHANIDES achat@uom.gr, steph@uom.gr
*Department of Applied Informatics, University of Macedonia,
156 Egnatia Str., 54006 Thessaloniki, Greece*

Abstract. Acknowledging the intense requirement for low power operation in most portable computing systems, this paper introduces the notion of energy efficient software design and proposes metrics, for evaluating software systems in terms of their energy consumption. Considering the sources of power consumption in every digital circuit, and the fact that power is primarily dependent on the executing software, appropriate energy measures are derived, which can be extracted from the flowgraph of a program. The proposed measures are computed by applying rules common to the existing hierarchical measures of other internal software attributes, and form the basis for the definition of a software energy metric. This metric can be used in order to determine the level of energy consumption of any software system more efficiently than existing assembly-parsing techniques, with only a limited penalty in accuracy. Application to different implementations of algorithms, drawn from matrix algebra and multimedia, demonstrates the efficiency of the proposed energy metric for comparison purposes, and as an indicator for quality improvement.

Keywords: software metrics, energy consumption, embedded software

1. Introduction

Low power operation of high-performance computing systems has become a major research issue, in both industry and academia, due to the proliferation of portable and embedded devices. Power consumption is critical 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 (Chandrakasan, 1995). Often, a limited power budget is allowed to the designers, and the challenge to meet this design constraint 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.

The development of low power devices, relies on both the underlying hardware, as well as on the software executing on embedded processor cores (Fornaciari, 1998). Therefore, 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, summarized in (Benini 2000), is to reduce the dynamic power dissipation, which is due to charging/discharging of the circuit capacitances (Chandrakasan, 1995). 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. On the other hand, software methodologies usually address higher levels of the system design hierarchy, and consequently the energy savings resulting from software optimization are larger (Landman, 1996). Software techniques primarily targeted at performing a given task using fewer instructions, resulting in a reduction of the circuit switching activity, which is the main source of power consumption in CMOS circuits (Chandrakasan, 1995).

According to the above, it would be desirable to have metrics in order to evaluate the energy efficiency of given software (Brooks, 2000). Such a metric would be valuable in adopting the most efficient implementation among several programs, for a specific low power application, in guiding the hardware/software partitioning process (Fornaciari, 1998), and further in providing an indicator that could help the design team improve software quality. Considering the desired attributes for an effective software metric (Ejiogu, 1991) the required energy metric should be: simple and computable, intuitively persuasive, consistent, and as much as possible, programming language independent.

A large number of metrics concerning internal product attributes, such as program length, complexity, functionality, structuredness, and modularity, have been proposed and extensively applied to software products (Fenton, 1996; Jalote, 1997). Although energy consumption of a program refers to the dynamic behavior of a program, and depends on the hardware platform on which software executes, techniques based on the flowgraph model of a program will be extended in order to measure unambiguously, run-time related attributes. Since the main sources of energy consumption in computing systems are processor operation, and accesses to memory, measures for these dynamic characteristics have to be developed.

The research towards energy efficient software, has been initiated with the development of instruction level power models that estimate the energy consumption of the processor (Tiwari, 1994; Tiwari, 1996). These models can be applied either after the execution of the program under study, in order to obtain the trace of executed instructions (Sinha, 2001), or by employing static analysis techniques to estimate the number of executed instructions, at the source level (Malik, 1997). Based on the observation that inter-instruction energy costs differ for different pairs of instructions (Tiwari, 1994) several energy-optimizing techniques have been proposed that apply scheduling algorithms in order to minimize power consumption (Tiwari, 1996; Lee, 1997). Cycle-accurate instruction level models, based on switched capacitance estimation, have also been used for evaluating the integrated impact of software and hardware optimizations (Ye, 2000; Brooks, 2000). Previous literature concerning energy aware software design, also includes methodologies that target memory related power consumption, and apply code transformations in order to move memory accesses to smaller memory layers, with lower energy cost per access (Catthoor, 1998).

However, in all previous efforts, the energy consumption of a given program is always estimated upon the specific target architecture, without any attempts to define energy metrics that can be used for comparing software designs independently of the underlying hardware. Moreover, energy efficient software design has not been studied considering the combined effect of processor and memory power consumption, while no exploration of the power implications of actual programming constructs for which several alternatives exist, has been performed. It should also be noted, that existing methods employ the trace of assembly instructions for a given program, and the time required for this process becomes prohibitive for large applications. Making power/performance tradeoffs visible at higher levels of abstraction, has become a necessity, especially when large design spaces have to be explored in short times (Fornaciari, 1998; Brooks, 2000).

The aim of this paper, is to highlight the impact of software on the power consumption of the underlying hardware, and the importance of considering power as a design parameter in software engineering. To this end, the use of energy measures for evaluating a program's energy consumption will be investigated, while a software energy metric will be defined. The efficiency of the proposed metric, will be demonstrated through application on different implementations of four algorithms, which have been evaluated in terms of performance and energy consumption, both experimentally, using an embedded processor simulator, and by applying the proposed measures.

The paper is organized as follows: Section 2 provides an overview of the sources of power consumption in an embedded system. Section 3 introduces the proposed measures and software energy metric, while in Section 4 the proposed metric will be applied to a set of algorithms. Finally, we conclude in Section 5.

2. Sources of power consumption

When a program executes on an embedded processing unit, two main sources of power consumption, with varying importance according to the architecture and target application can be identified:

2.1. Processor power consumption

Processor power consumption is due to the operation of the processor circuitry during the execution of program instructions. Instruction decoding and execution 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 (Chandrakasan, 1995). To quantify this power component, appropriate instruction-level power models have been developed. These models are based on the hypothesis (Tiwari, 1994), that 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 (Tiwari, 1996; Sinevriotis, 1999). 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.

b. Overhead energy 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.

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.

2.2. Memory power consumption

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 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, is directly proportional to the number of accesses, and depends on the size and the number of ports of the memory, the power supply, and the technology.

According to the above, the number of executed instructions has a twofold impact on energy consumption, by means of instruction execution within the processor, and instruction fetching from memory. On the other hand, the contribution of memory accesses is considered only once; however the energy cost of a data memory access is usually much larger than that of an instruction memory access, or the energy that is consumed during the execution of an instruction in the processor. The relative energy values of these components will be taken into account in the definition of a software energy metric, as will be shown later.

3. Software energy evaluation

The energy measures that will be introduced, are based on the fact that the control flow of each program can be represented by a directed graph, namely the flowgraph of the program (Fenton, 1996). The basic S-graphs, which are considered for defining the family of allowed S-structured graphs, to which the flowgraph of an arbitrary program belongs, are shown in Figure 1, together with the corresponding C programs.

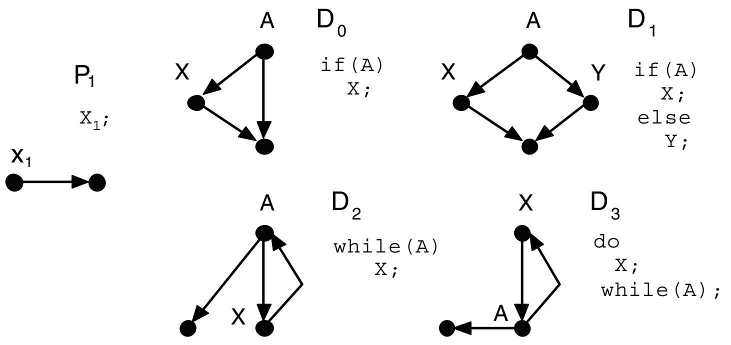


Figure 1. Basic *S*-graphs.

3.1. Hierarchical energy measures

In order to evaluate software designs in terms of their energy consumption, we define the following two hierarchical measures by assigning a value to each prime of the basic *S*-graphs, and a value to the sequencing and nesting functions:

a) Executed Instruction Count Measure (EIC), which corresponds to the number of executed assembly instructions considering a typical embedded integer processor core (Furber, 2000). This number is related to the processor energy consumption, as calculated using instruction level power models, as well as to the instruction memory energy consumption, since each executed instruction corresponds to an instruction fetch from the instruction memory.
b) Memory Access Count Measure (MAC), which is equal to the number of memory accesses to the data memory, in order to fetch program operands.

It should be mentioned, that here, the use of flowgraphs is extended for estimating nonstatic attributes. The inefficiency of flowgraphs in providing dynamic measures, is primarily due to the structures of repetition and selection, where the run-time behavior cannot be predicted, except for some cases. Although this subject belongs to the field of static analysis (Malik, 1997), for the purpose of this paper, a rather simplified approach is followed: For repetition structures, each loop is annotated, either with the exact number of loop iterations (in case it is explicitly expressed in the program), or with an estimate provided by the designer. Although an inaccurate estimate can lead to large absolute errors, especially when the depth of nesting is large, for the purpose of comparing several implementations of the same algorithm in order to determine the most efficient one, an estimate proves to be sufficiently accurate, as will be shown next. That is because the estimate will be applied to similar repetition structures, filtering out the effect of an inaccurate estimation.

For the case of selection, the largest energy consumption of both branches is considered to account for the upper bound of the program's energy consumption. This approach is similar to Shaw's simple timing schema approach (Shaw, 1989) for an `if-then-else` statement.

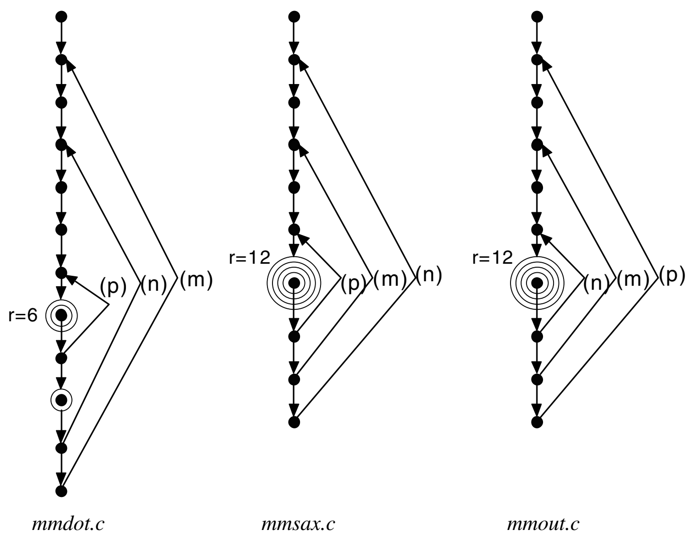


Figure 2. Matrix–matrix multiplication flowgraphs, annotated with the number of loop iterations and memory accesses.

Apart from annotating edges in a flowgraph that belong to structures which employ repetition, each node in the flowgraph is annotated with the number of memory accesses that are performed in the corresponding program statement. To indicate that a number of *i* memory accesses are performed at a specific node *n*, the node is encircled with *i* concentric circles (Figure 2). Moreover, to facilitate the calculation of number of executed instructions, each node which corresponds to a program statement containing more than one arithmetic operator, is marked with the number of arithmetic operators (*r*) in this statement. In this way, it is also possible, by just observing the flowgraph, to locate which portions of the program are computationally intensive, or access memory heavily (Figure 2).

At this point, it should be made clear that the proposed approach suffers from the hypothesis that a given programming structure is converted to the same assembly code, independently of the compiler used. However, the primary aim, is to compare several alternatives for the same application, in terms of energy in early design stages, where the relative accuracy is much more important than the absolute accuracy (Landman, 1996; Fornaciari, 1998). Therefore, it is assumed that all program variations will benefit from the applied compiler optimizations to the same degree. For example, the ARM compiler that has been used in this work, performs a number of optimizations, such as common subexpression elimination, loop invariant motion and constant folding. If in a program, a high-level statement is compiled to assembly code employing one or more optimizations, it is assumed that a similar statement in another variation of the program will be compiled using the same optimizations. In this way, the drawback of not considering a specific compiler during the process of extracting from a flowgraph, the number of executed instructions is partially relaxed. The impact of several compiler optimizations on energy consumption has been investigated in (Kandemir, 2000). Another point worth mentioning, is that embedded processors often employ data caches, which can affect significantly, the energy/performance tradeoff of the system (Shiue, 1999; Kandemir, 2000). In this case, a cache simulator can be used, and according to the estimated cache miss ratio, the energy consumption due to accesses to the main memory and cache layer, should be appropriately calculated (Dasigenis, 2001).

Considering the above approximations, the rules (Fenton, 1996) for the two energy measures are defined below:

Executed instruction count measure I:

- **Prime functions.** $I(P_1) = r + m$, where r is the number of basic mathematical operators (excluding division), and m is the number of memory accesses within the statement represented by P_1, respectively, and for each prime $F \neq P_1$

$$I(F) = \begin{cases} i \cdot [3 + \max((r+m)_X, (r+m)_Y)], & F = D_1 \\ i \cdot [3 + r + m], & F \neq D_1 \end{cases},$$

where i represents the number of loop iterations (by default, for D_0 and D_1, i=1). In case $F = D_1, X, Y$ refer to the two nodes of D_1. (The above expression assumes, for the sake of simplicity, that all prime flowgraphs correspond to an execution of 3 instructions (two for setting a condition by loading appropriate register values and performing the comparison, and one for branching to the beginning of the loop)). Memory accesses (m) are counted here, because each access is assumed to contribute one load/store assembly instruction to the code.

- **Sequencing function.**

$$I(F_1; F_2; \ldots; F_n) = \sum I(F_i)$$

- **Nesting function.**

$$I(F(F_1, F_2, \ldots, F_n)) = \begin{cases} i \cdot \left[(3 + \max\left(\sum I(F_i)_X, \sum I(F_i)_Y\right))\right], & F = D_1 \\ i \cdot \left[3 + \sum I(F_i)\right], & F \neq D_1 \end{cases}$$

for each prime $F \neq P_1$, where i the number of loop iterations corresponding to F, and max refers to the maximum instruction count measure of both branches (true and false) of F.

Memory access count measure M:

- **Prime functions.**

$$M(P_1) = m,$$

where m is the number of memory accesses within the statement represented by

P_1, and for each prime $F \neq P_1$

$$I(F) = \begin{cases} i \cdot \max(m_X, m_Y), & F = D_1 \\ i \cdot m, & F \neq D_1 \end{cases},$$

where i represents the number of loop iterations (by default, for D_0 and D_1, $i = 1$).

- **Sequencing function.**

$$M(F_1; F_2; \ldots; F_n) = \sum M(F_i)$$

- **Nesting function.**

$$M(F(F_1, F_2, \ldots, F_n)) = \begin{cases} i \cdot \max(\sum M(F_i)_X, \sum M(F_i)_Y), & F = D_1 \\ i \cdot \sum M(F_i), & F \neq D_1 \end{cases}$$

for each prime $F \neq P_1$, where i the number of loop iterations corresponding to F.

3.2. Software energy metric

Considering the average energy cost of an instruction (Sinevriotis, 2001), that of a data memory access (assuming a RAM data memory), and that of an instruction memory access (assuming a ROM instruction memory), the following simple software energy metric (SEM) is proposed, based on the hierarchical measures derived in the previous paragraph:

$$SEM = EIC + 2 \times MAC \tag{1}$$

In Equation (1), *EIC* accounts for the energy consumption, due to the execution of instructions within the processor, and due to fetches from the instruction memory, while *MAC* accounts for the energy dissipation due to accesses to the data memory. *SEM* preserves the intuitive feeling about energy consumption (Pressman, 1997), that if a program A consumes more energy than program B, *SEM(A)* is not only larger than *SEM(B)*, but the ratio of the actual energy of program *A* to that of program *B*, is close to the ratio of *SEM(A)* over *SEM(B)*. This will be shown through the examples of the next section. Consequently, the proposed metric is also consistent in its use of units. Finally, it should be noted that *SEM* is relatively independent of the programming language, since the hierarchical measures on which it is based, have been extracted, considering the assembly equivalent of common programming structures assuming a general purpose processor.

In the above equation for SEM, it can be observed, that although the number of executed instructions has a twofold contribution to the energy consumption of the system as explained previously (processor execution, plus instruction fetches), the number of memory accesses is multiplied by a factor of 2. That is, because the energy cost of an access to a RAM memory (where data are stored/retrieved) is significantly larger than that of an access to a ROM memory, or the energy consumed by the processor in order to execute an instruction. Only recently developed embedded DRAM*s*, which are integrated on the same die with digital logic, can shift this balance, so that computations have an energy cost similar to data memory access (Benini, 2000). However, this implementation increases the cost significantly, and is therefore only used in a limited number of designs.

4. Results

To evaluate the efficiency of the proposed measures and metrics, a set of applications from matrix algebra will be examined first. Let us consider the following three programs in *C* (Table 1), which all perform matrix–matrix multiplication employing: a) a dot product computation (mmdot.c); b) a generalized SAXPY operation (mmsax.c); and c) an outer product update (mmout.c) (Golub, 1996). The matrix multiplication is mathematically formulated as:

$$C = AB \quad \left(\Re^{m\times p} \times \Re^{p\times n} \to \Re^{m\times n}\right) \tag{2}$$

For reasons for direct transformation to a flowgraph, the classical notation of the above multiplication using `for( )` statements, has been converted employing `while( )` statements. It should be mentioned, that for the versions employing the SAXPY operation, and the outer product to work correctly, the *C* matrix should be zeroed before the computations. The flowgraphs for the above programs, annotated with the number of loop iterations (on edges), the number of memory accesses (encircled nodes), and arithmetic operators (on nodes) are shown in Figure 2.

The decomposition tree for the flowgraph of program mmdot.c, which illustrates the hierarchy of primes, is shown in Figure 3. (Fenton, 1996). Applying the rules that

Table 1. Matrix-matrix multiplication algorithms

mmdot.c	mmsax.c	mmout.c
`i=0;`	`k=0;`	`k=0;`
`while (i<m)`	`while (k<n)`	`while (k<p)`
`{`	`{`	`{`
` j=0;`	` j=0;`	` i=0;`
` while (j<n)`	` while (j<m)`	` while (i<m)`
` {`	` {`	` {`
`  c=0;`	`  i=0;`	`  j=0;`
`  k=0;`	`  while (i<p)`	`  while (j<n)`
`  while (k<p)`	`  {`	`  {`
`  {`	`   *(&C[0][k]+j*n)=`	`   *(&C[0][0]+i*n+j)=`
`   c=c+`	`   *(&A[0][0]+i+j*p)*`	`   *(&A[0][k]+i*p) *`
`     (*(&A[i][0]+k))*`	`   (*(&B[0][k]+i*n))+`	`   (*(&B[k][0]+j)) +`
`     (*(&B[0][j]+k*n));`	`   *(&C[0][k]+j*n);`	`   *(&C[0][0]+i*n+j);`
`   k++;`		
`  }`	`   i++;`	`   j++;`
`  C[i][j]=c;`	`  }`	`  }`
`  j++;`	`  j++;`	`  i++;`
` }`	` }`	` }`
` i++;`	` k++;`	` k++;`
`}`	`}`	`}`

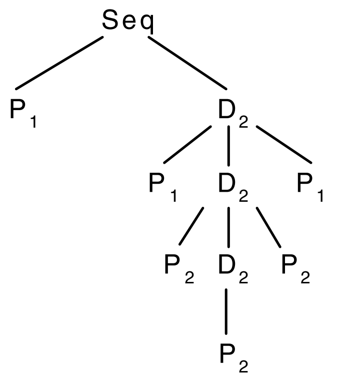


Figure 3. Decomposition Tree for the flowgraph of program mmdot.c.

have been defined in the previous section, for the proposed hierarchical measures to the flowgraph of the mmdot.c example, the Executed Instruction Count measure is given by:

$$\begin{aligned}
I(F) &= I(P_1;\, D_2(P_1;\, D_2(P_2;\, D_2(P_2);\, P_2);\, P_1)) \\
&= I(P_1) + I(D_2(P_1;\, D_2(P_2;\, D_2(P_2);\, P_2);\, P_1)) \\
&= I(P_1) + m \cdot [3 + I(P_1;\, D_2(P_2;\, D_2(P_2);\, P_2);\, P_1)] \\
&= I(P_1) + m \cdot [3 + I(P_1) + I(D_2(P_2;\, D_2(P_2);\, P_2)) + I(P_1)] \\
&= I(P_1) + m \cdot [3 + I(P_1) + n \cdot (3 + I(P_2;\, D_2(P_2);\, P_2)) + I(P_1)] \\
&= I(P_1) + m \cdot [3 + I(P_1) + n \cdot (3 + I(P_2) + I(D_2(P_2)) + I(P_2)) + I(P_1)] \\
&= I(P_1) + m \cdot [3 + I(P_1) + n \cdot (3 + I(P_2) + p \cdot (3 + I(P_2)) + I(P_2)) + I(P_1)] \\
&= 1 + m \cdot [3 + 1 + n \cdot (3 + 2 + p \cdot (3 + 9) + 3) + 1] \\
&= 1 + m \cdot (5 + n(8 + p \cdot 12))
\end{aligned}$$

while the Memory Access Count measure can be calculated as:

$$\begin{aligned}
M(F) &= M(P_1;\, D_2(P_1;\, D_2(P_2;\, D_2(P_2);\, P_2);\, P_1)) \\
&= M(P_1) + M(D_2(P_1;\, D_2(P_2;\, D_2(P_2);\, P_2);\, P_1)) \\
&= M(P_1) + m \cdot [M(P_1;\, D_2(P_2;\, D_2(P_2);\, P_2);\, P_1)] \\
&= M(P_1) + m \cdot [M(P_1) + M(D_2(P_2;\, D_2(P_2);\, P_2)) + M(P_1)] \\
&= M(P_1) + m \cdot [M(P_1) + n \cdot M(P_2;\, D_2(P_2);\, P_2) + M(P_1)] \\
&= M(P_1) + m \cdot [M(P_1) + n \cdot (M(P_2) + M(D_2(P_2)) + M(P_2)) + M(P_1)]
\end{aligned}$$

$$
\begin{aligned}
&= M(P_1) + m \cdot [M(P_1) + n \cdot (M(P_2) + p \cdot M(P_2) + M(P_2)) + M(P_1)] \\
&= 0 + m \cdot [0 + n \cdot (0 + p \cdot 2 + 1) + 0] \\
&= 2 \cdot m \cdot n \cdot p + m \cdot n
\end{aligned}
$$

In order to evaluate the actual number of executed instructions and memory accesses, the ARM7 embedded processor core has been chosen as target architecture, which is widely used in embedded applications, due to its promising MIPS/mW performance. Moreover, it offers the advantage of an open architecture to the designer (Furber, 2000).

The experimental process that has been followed in order to evaluate software design in terms of power, is shown in Figure 4. Each code was compiled using the *C* compiler of the ARM Software Development Toolkit. 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.

This trace file was then parsed serially by a separate profiler, that has been developed, in order to obtain the number of data memory accesses, and the energy that is consumed within the processor during the execution of a program. The parser has built-in look-up tables, containing physical measurements (Sinevriotis, 2001) of

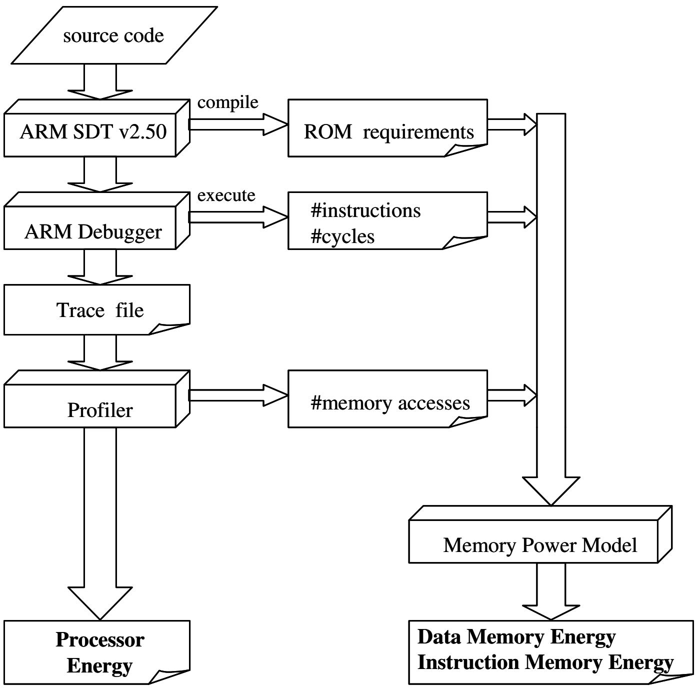


Figure 4. Experiment set up for evaluating energy consumption.

Table 2. Comparison between simulated and calculated results for the proposed measures

	mmdot.c		mmsax.c		mmout.c		Avg. Error
	Sim.	Calc.	Sim.	Calc.	Sim.	Calc.	
#instructions	382	363	495	517	474	502	5.12%
	56	56	96	96	96	96	0%
#memory	48 loads		72 loads		72 loads		
accesses	8 stores		24 stores		24 stores		

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. The accuracy between the energy that is calculated based on ARM simulations, and the actual energy that is measured on a real ARM processor has been found to be sufficient: (Theokharidis, 2000) and (Sinevriotis, 2001) report an average error between simulation results, and actual measurements of 1.7% and 7.1%, respectively.

Finally, the number of executed instructions is 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, is used to compute the energy consumption of the data memory.

In order to investigate the efficiency of the proposed measures on characterizing the behavior of the underlying hardware, each program was simulated on the ARMulator. The results, for $m = 2, p = 3$, and $n = 4$, are shown in Table 2. As it can be observed, the number of memory accesses is estimated without error, while the number of executed assembly instruction presents a limited average error, due to the assumptions made concerning the compilation of *C* code to assembly instructions.

In Table 3, the actual energy consumption for all system components (processor, instruction and data memory) is presented, based on the power models that have been developed, and using the simulation data from the ARM processor simulator.

Finally, to investigate the efficiency of the proposed software energy metric (SEM), the normalized energy consumption, and the normalized value of SEM over the *mmdot.c* program, are displayed in Table 4. In spite of the inevitable approximations that have been made in order to model dynamic characteristics of the programs, the error is relatively low. This justifies the rationale of developing a software energy metric, being the possibility to compare in terms of energy, different alternatives in the implementation of a specific algorithm or system (Landman, 1996).

Table 3. Energy consumption of matrix multiplication algorithms (mJ)

	mmdot.c	mmsax.c	mmout.c
processor energy	0.001997	0.002687	0.002592
data_mem energy	0.000252	0.000432	0.000432
instr_mem energy	0.000342	0.000443	0.000424
total energy	$2.59 \cdot 10^{-3}$	$3.56 \cdot 10^{-3}$	$3.45 \cdot 10^{-3}$

Table 4. Normalized Energy consumption and SEM for matrix multiplication algorithms

	mmdot.c	mmsax.c	mmout.c
actual energy	1	1.375	1.332
SEM	1	1.493	1.461
error		8.58%	9.68%

The second example is also drawn from matrix algebra, and refers to setting up a *Hilbert* matrix (Van Loan, 2000). For its computation, either a double-loop script can be used, or taking advantage of the symmetric structure of this matrix, the calculation of the array elements can be performed in half the time. The two programs are shown in Table 5.

Similarly, from the flowgraphs of the two programs, the hierarchical measures for the number of executed assembly instructions, and memory accesses, has been extracted. Based on these measures, the value of the software energy metric has been evaluated, and its normalized value over the first program, is shown in Table 6, compared to the normalized value of the actual energy consumption.

Energy metrics can be further exploited, since based on the extracted SEM metric, the behavior of a program can be explored for several input data, without the need to simulate the program. For the Hilbert matrix example, the energy consumption of both programs can be explored for several matrix dimensions. In the diagram of Figure 5, the SEM metric for both algorithms is plotted versus n.

To investigate the efficiency of the proposed metric for larger applications, results have also been derived from the multimedia domain, and specifically from the full-search motion estimation, and the two-dimensional three-step logarithmic search algorithm (Bhaskaran, 1999). These algorithms, which play an important role in video encoding standards (e.g. MPEG-X, H.26X), calculate the motion vector for each block in a frame, using the mean absolute error (MAE) as a matching criterion, however, the three-step logarithmic search employs a heuristic search

Table 5. Alternatives for setting up a Hilbert matrix

hilbert1.c	hilbert2.c
`i=1;`	`i=1;`
`while(i<=n)`	`while(i<=n)`
`{`	`{`
`  j=1;`	`  j=i;`
`  while(j<=n)`	`  while(j<=n)`
`  {`	`  {`
`    A[i][j]=1/(i+j-1);`	`    A[i][j]=1/(i+j-1);`
`    j++;`	`    A[j][i]=A[i][j];`
`  }`	`    j++;`
`  i++;`	`  }`
`}`	`  i++;`
	`}`

Table 6. Normalized energy consumption and SEM for Hilbert programs

	hilbert1.c	hilbert2.c
actual energy	1	0.798
SEM	1	0.742
error		7.02%

strategy, which reduces the computational complexity. For each algorithm, the original and one transformed code are examined, where the latter has been extracted from the original one, by applying data-reuse transformations, where copies of data from inner loops that exhibit high reuse are moved to additional arrays in outer loops (Catthoor, 1998). These applications make intensive use of control structures, having a maximum depth of nesting of 6 and 7, respectively.

Tables 7 and 8 summarize the simulated and calculated data for both algorithms (original and transformed codes), and illustrate the accuracy of the *SEM* metric. As it can be observed, the accuracy in the estimation of the executed instruction count for the logarithmic-search algorithm is limited, due to the fact that small errors within the most inner loops accumulate easily, because of the large number of iterations. However, since similar approximations are made for both variations of the program, the effect of these errors on the value of *SEM* leads to sufficiently accurate results.

It is worth mentioning, that although the proposed approach offers reduced accuracy compared to low-level techniques that parse the trace of assembly instructions in order to estimate energy (Tiwari, 1996; Ye, 2000; Sinevriotis, 2001; Sinha, 2001) it achieves a significant speedup. While the construction and reading of a flowgraph takes time, in the order of a few seconds, the process of generating and parsing the trace file employed in previous methodologies, is extremely time consuming, and time becomes prohibitive for large applications: For the full search motion estimation algorithm, the total time required to generate and parse the trace file, was approximately 6 hours on a Pentium III 500MHz processor.

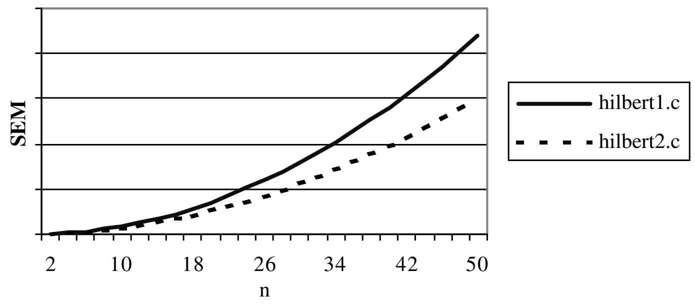


Figure 5. Software Energy Metric versus matrix dimensions for Hilbert programs.

Table 7. Results for the Full Search motion estimation algorithm

	Full Search (Orig.)	Full Search (Transf.)
#instructions	258684372	165043023
EIC	253098209	162585658
#memory accesses	11144498	13105826
MAC	11449350	13151358
Norm. Energy	1	0.70
Norm. SEM	1	0.68

Table 8. Results for the three step logarithmic search motion estimation algorithm

	3_step_log Search (Orig.)	3_step_log Search(Transf.)
#instructions	59769369	49654998
EIC	52268393	40453625
#memory accesses	2725335	2726435
MAC	2737746	2832390
Norm. Energy	1	0.80
Norm. SEM	1	0.79

5. Conclusions

Low power operation of embedded software is a major concern in the design of portable devices. Although hardware decisions can affect the energy consumption of a digital system, switching activity, which is the main source of power consumption, is primarily determined by the software executing on embedded processor cores. In this paper, the need to define appropriate energy metrics for evaluating software implementations, in terms of power consumption of the underlying hardware, is discussed. A pair of hierarchical energy measures extracted from a program flowgraph is proposed for quantifying the number of executed instructions, and the number of data memory accesses. Based on these measures, a software energy metric is defined for evaluating the energy consumption, due to processor operation, and due to accesses to the instruction and data memory. The proposed metric is evaluated on applications from matrix algebra, and the multimedia domain proving its consistency and accuracy, and highlighting the importance of considering energy during software design. It is believed, that further research into this subject can lead to the definition of more sophisticated metrics, which will help software designers to improve software quality in terms of power consumption.

References

Benini, L. and De Micheli, G. 2000. System-level power optimization: Techniques and tools, *ACM Transactions on Design Automation of Electronic Systems*, 5(2): 115–192.

Bhaskaran, V. and Konstantinides, K. 1999. *Image and Video Compression Standards: Algorithms and Architectures*, 2nd ed., Boston, Kluwer Academic Publishers.

Brooks, D., Tiwari, V., and Martonosi, M. 2000. Wattch: A framework for architectural-level power analysis and optimizations, *Proc. International Symposium on Computer Architecture*, Vancouver, BC, Canada, pp. 83–94.

Catthoor, F., Wuytack, S., De Greef, E., Balasa, F., Nachtergaele, L., and Vandecappelle, A. 1998. *Custom Memory Management Methodology: Exploration of Memory Organization for Embedded Multimedia System Design*, Boston, Kluwer Academic Publishers.

Chandrakasan A. and Brodersen R. 1995. *Low Power Digital CMOS Design*, Boston, Kluwer Academic Publishers.

Dasigenis, M., Kroupis, N., Argyriou, A., Tatas, K., Soudris, D., Zervas, N., and Thanailakis, A. 2001. A memory management approach for efficient implementation of multimedia kernels on programmable architectures, *Proc. IEEE Computer Society Annual Workshop on VLSI*, Orlando, Florida, pp. 171–176.

Ejiogu, L. 1991. *Software Engineering with Formal Metrics*, QED Publishing.

Fenton, N.E. and Pfleeger, S.L. 1996. *Software Metrics: A Rigorous and Practical Approach*, London, International Thomson Computer Press.

Fornaciari, W., Gubian, P., Sciuto, D., and Silvano, C. 1998. Power estimation of embedded systems: A hardware/software codesign approach, *IEEE Trans. on VLSI Systems*, 6(2): 266–275.

Furber, S. 2000. *ARM System-on-Chip Architecture*, Harlow, Addison-Wesley.

Golub, G.H. and Van Loan, C.F. 1996. *Matrix Computations*, Baltimore, John Hopkins University Press.

Jalote, P. 1997. *An Integrated Approach to Software Engineering*, New York, Springer Verlag.

Kandemir, M., Vijaykrishnan, N., Irwin, M.J., and Ye, W. 2000. Influence of compiler optimizations on system power, *Proc. Design Automation Conference*, Los Angeles, CA, pp. 304–307.

Landman, P. 1996. High-level power estimation, *Proc. International Symposium on Low Power Electronics and Design*, Monterey, CA, pp. 29–35.

Lee, M.T.-C., Tiwari, V., Malik, S., and Fujita, M. 1997. Power analysis and minimization techniques for embedded DSP software, *IEEE Trans. on VLSI Systems*, 5(1): 123–135.

Malik, S., Martonosi, M., and Li, Y.-T.S. 1997. Static Timing Analysis of Embedded Software, *Proc. Design Automation Conference*, Anaheim, CA, pp. 147–152.

Pressman, R.S. 1997. *Software Engineering: A Practitioner's Approach*, New York, McGraw-Hill.

Shaw, A.C. 1989. Reasoning about time in higher-level language software, *IEEE Transactions on Software Engineering*, 15(7): 875–889.

Shiue, W.-T. and Chakrabarti, C. 1999. Memory exploration for low power, embedded systems, *Proc. Design Automation Conference*, New Orleans, Louisiana, pp. 140–145.

Sinevriotis, G. and Stouraitis, Th. 1999. Power analysis of the ARM 7 embedded microprocessor, *Proc. Int. Workshop on Power and Timing Modeling, Optimization and Simulation*, Kos, Greece, pp. 261–270.

Sinevriotis, G. and Stouraitis, Th. 2001. SOFLOPO: low power software development for embedded applications, ESPRIT ESD-LPD Project 25403, Public Final Report (http://www.vlsi.ee.upatras.gr/soflopo/index.html).

Sinha A. and Chandrakasan, A. 2001. JouleTrack—A web based tool for software energy profiling. *Proc. Design Automation Conference*, Las Vegas, Nevada, pp. 220–225.

Tiwari V., Malik, S. and Wolfe, A. 1994. Power analysis of embedded software: A first step towards software power minimization, *IEEE Transactions on VLSI Systems*, 2(4): 437–445.

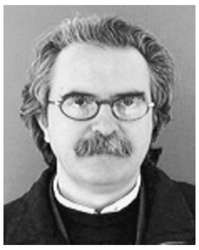
Tiwari, V., Malik, S., Wolfe, A., and Lee, T.C. 1996. Instruction level power analysis and optimization of software, *Journal of VLSI Signal Processing*, 13(2): 1–18.

Theokharidis, M. 2000. Measuring energy dissipation of ARM7TDMI processor instructions, University of Dortmund, Dept. of Computer Science. (http://ls12-www.cs.uni-dortmund.de/publications/theses).

Van Loan, C.F. 2000, *Introduction to Scientific Computing*, Upper Saddle River, Prentice Hall.

Ye, W., Vijaykrishnan, N., Kandemir, M., Irwin, M. J. 2000, The design and use of SimplePower: A cycle-accurate energy estimation tool, *Proc. Design Automation Conference*, Los Angeles, CA, pp. 340–345.

Alexander Chatzigeorgiou is a Lecturer in the Department of Applied Informatics at the University of Macedonia, Thessaloniki, Greece. He received the Diploma in electrical engineering, and the Ph.D. degree in computer science, from the Aristotle University of Thessaloniki, Greece, in 1996 and 2000, respectively. From 1997 to 1999, he was with Intracom S.A. Greece, as a Telecommunications Software Designer. His research interests are in software engineering, object-oriented design, and low-power hardware/software design.



George Stephanides is an Assistant Professor in the Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece. He holds a Ph.D. degree in Applied Mathematics from the University of Macedonia. His current research and development activities are in the applications of mathematical programming, scientific computing, and application specific software.