

A Fast, Dynamic, Fine-Detail, Source Level Technique to Estimate the Energy Consumed by Embedded Software on Single-Issue Processor Cores

Daniele Paolo Scarpazza* and Carlo Brandolese

Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italy

(Received: 22 December 2005; Accepted: 1 June 2006)

Embedded system designers frequently face the problem of estimating how much energy and time are spent in the different portions of their software. This is crucial to determine how fast their software runs on a platform, how much energy it consumes, where optimizations are needed, or what hardware it requires to ensure a given speed. This problem is not effectively solved by current approaches: Instruction-level simulation (ISS), static timing analysis (STA), and source-level instrumentation (SLI). ISS provides too low-level information and is too slow, STA cannot deal with dynamism, while current SLI tools only provide global or function-level estimates. We propose a novel technique to estimate time and energy consumed by individual operations in the program when running on a single-issue processor core. With this information, designers can better understand which portions of the source code cause the major consumptions, and apply optimizations there. We implemented the methodology in a tool flow which exhibits simulation times 10,000 shorter than a reference ISS, and shows good estimation accuracy.

Keywords: Embedded Systems, Energy, Power, Estimation, Optimization.

1. INTRODUCTION

This section motivates the need for fast, dynamic, fine-detailed source-level estimation techniques in the context of contemporary embedded software design, it explains why the approaches currently available do not satisfy these requirements, it introduces the basic traits of our proposed technique and explains why the technique is an answer to this need with reference to the estimation of software on single-issue processor cores.

There is an increasing demand for battery-operated pervasive systems (the ones which power mobile embedded applications, ambient intelligence, sensor networks, and wearable computing) featuring complex software applications. This demand challenges designers with strict energy constraints, while performance must be kept within specifications and acceptability, and costs contained. Building such systems for realistic scenarios is a difficult task: When compared to more traditional domains, methodologies, and tools are quite immature, and this significantly hinders progresses in this field.

Designers employ techniques to estimate the performance and energy consumption of their software for many purposes: For example to guide hardware dimensioning decisions (e.g., which processor to adopt, which speed grade, with how much memory, which type and size of batteries, etc.), to compare different algorithms or alternative implementations of the same algorithm, and to determine the critical sections of the code in terms of time or energy costs, which is crucial for steering the optimization choices.

These techniques need to satisfy the following requirements:

- they need to be *fast* because the complexity of modern embedded applications is high and quickly increasing. Simulating realistic-sized applications at the circuit level or gate level is unaffordable now, and also instruction-set simulation is quickly becoming unaffordable for application of sufficient complexity (e.g., video decoders). Whichever technique is cycle-accurate, or close to cycle accuracy is doomed to obsolescence very soon. Estimation techniques with a high performance are needed, even at the expenses of inferior accuracy;
- they need to be *dynamic*, because applications depend more and more on the contents they process: e.g., today with MPEG-2 motion prediction, tomorrow with MPEG-4

*Author to whom correspondence should be addressed.
 Email: scarpaz@scarpaz.com

object-based coding. With such a high workload variability, the gap between worst-case and typical consumed energy is very high. Therefore, employing static techniques (which are worst-case) leads to oversized, expensive systems, which are underutilized for the most time;

- they need to be *source-level* techniques, because designers nowadays write code mostly in high level languages rather than in assembly. Tools must be able to provide estimates for the entities in the source code. Additionally, the transformation which have been showed to provide the most gains are source-level ones, and only source-level estimation techniques can be used to steer source-level transformations.

- they need to be *fine grained* because the portions which consume most energy and time (“hot spots”) are typically small computational kernels inside loop bodies. Tools must be detailed enough to provide estimates at this degree of detail. Rougher detail levels (i.e., the function) are insufficient.

With reference to the time and energy consumed by the processor core (datapath and register file), there are no approaches in literature which satisfy the above requirements adequately at the same time: For example, instruction set simulation is very slow and produces low level information which is poorly usable to guide optimization, while static timing analysis fails to address many dynamic features of the code and leads to worst-case estimates which may be unrealistic. We devote Section 2 to survey the literature and detail all these claims. To the best of our knowledge, the technique we propose is the first to satisfy all these requirements, with reference to the time and energy consumed by the processor core. We do not address the time and energy consumed by the memory hierarchy, because the problem has been effectively addressed in literature. The technique we present is designed for single-issue processor cores.

We believe that our technique is the enabling technology to allow a short design exploration loop, which helps the designer in evaluating multiple solutions in a short time, and guides source-level optimization by indicating precisely which source elements are responsible for the most consumption. Our approach exhibits several other advantages. It externally behaves like GCC, thus enabling developers to estimate their projects without modifications. It needs architectural parameters of an architecture rather than the actual availability of hardware or of its simulator. Silicon vendors can tune it on cores for which architectural details remain secret, allowing their customers to estimate software consumption on those cores. Tuning the method on a new target platform involves little effort. The methodology can be used to obtain estimates on processors which are not yet available in hardware.

The remainder of this paper is organized as follows. In Section 2 we survey the related work. In Section 3 we illustrate our methodology. In Section 4 we describe how

we set up the experiments and the results we obtained. Finally, in Section 5 we draw the conclusions.

2. BACKGROUND

In this section we show that existing approaches for software analysis in the field of processor core consumption estimation do not address satisfactorily the requirements stated in the introduction: Source-level, fine-detail, fast and dynamic.

A traditional technique to estimate the performance of software is static timing analysis (STA). The primary objective of STA is to determine bounds on the execution time of a program on an given architecture: The worst-case execution time (WCET).^{1,3} STA does not satisfy the requirements stated above because it does not address dynamism. First of all, unbounded loops, recursion, dynamic function references make WCET reasoning undecidable,¹ while these features appear more and more frequently in modern applications. For example, it was quite possible to write an old-generation video decoder with bounded loops only, but it is unthinkable to write an MPEG-4 synthetic video decoder or a dynamic-topology ad-hoc wireless router in such a way. Additionally, designers are more interested in typical performance estimates in realistic cases, rather than theoretical worst cases. Nowadays, the gap between worst-case and typical workload is high and increasing, and this makes the WCET too conservative to be used as an indicator of typical performance. This is especially true for highly dynamic and data-dependent applications such as synthetic video decoding, interactive applications, dynamic topology networks, etc. Finally, STA becomes more and more complicated in the attempt to model current architectural features such as branch prediction, predication, prefetching, and cache policies.^{4,5}

Another traditional approach to estimate the energy consumption of a software is by using an Instruction-Set Simulator (ISS)⁶ such as SimIt-Arm,⁷ augmented with energy data, as in JouleTrack.⁸ These simulators provide good accuracy, but they are of little help to designers for two reasons: They provide estimates at a too low level of abstraction and they are too computationally demanding. As the abstraction level is concerned, they provide assembly-level estimates, while developers do not code in assembly language anymore nowadays, and only few developers have the necessary skills to optimize a source code on the basis of low-level estimates. Then, they are too slow because they exhibit simulation times which are many thousands times longer than the original applications.

To raise the abstraction level of the estimates, Simunic et al.⁹ have coupled an ISS with a statistical profiling tool (gprof), which periodically samples the program counter at runtime, obtaining a per-function breakdown of the execution time. On the basis of these statistics, the authors

redistribute the ISS estimates on individual C functions. The approach is unable to determine the consumption of any element finer grained than the function, therefore the approach is not fine grained. This limits its practical usability: In fact, it is very common for critical sections to entirely appear inside a single function body, and this approach is unable to indicate, inside the function body, which lines of code or operators caused the major consumptions. It is not possible to force the resolution of the method by moving loop bodies into separate functions, because this would make the estimates meaningless. This is because contiguous statements would be now in separate functions, thus perturbing the compiler's output and adding a spurious call overhead. Additionally, this approach is not flexible: In each different such measurement the designer must modify, recompile, and simulate the code again and again. In contrast, we propose an approach which provides individual elements' estimates in a single run, with no need for manual modifications or multiple iterations.

Bormans et al.¹⁰ have proposed Atomium, an analysis tool which addresses the memory-related aspects of system design, by applying the Data Transfer and Storage Exploration (DTSE) methodology. This tool, now available as a commercial product,¹¹ employs a source-level analysis approach which is similar to ours, operating at the behavioral level of an application, expressed in C, but the technique accounts for the cost of memory accesses, and not for operations and control, which is our main focus. Therefore, this technique and ours form a perfectly complementary match: Together they allow to estimate the cost of executing a software on a processor-based platform.

Lajolo et al.¹² have proposed a compilation-based estimation technique. It employs a real compiler (GCC) to obtain the control-flow graph (CFG) of the code. The CFG is then annotated with information useful to derive a cycle-accurate model. By actually compiling the code, the technique accounts exactly for the effects of compilation optimizations, although it is specific to the GCC compiler. The technique cannot be described as source-level, because it does not provide a way to attribute cost estimates back to the source-level elements which caused them.

Julien et al.¹³ designed SoftExplorer, a fast technique to estimate the consumption of data-dominated loops from the C source, based on functional-level modeling of the architecture and statistical parameters extracted from the code. The technique exhibit several shortcomings. First of all, it is static, therefore it relies on the user to determine the run-time execution flow which is impractical for programs of realistic size: It is not reasonable to ask the user the actual number of iterations for each loop in the source code of a modern complex application such as a video decoder. Additionally, the technique is not source level: It operates on the source code only to extract "opaque" algorithmic parameters, which allow to perform global estimates, but not to determine the cost of elements in the source code.

Ravasi and Mattavelli have proposed Software Instrumentation Tool,¹⁴ a technique designed to profile code at the source-level. The approach is dynamic and source-level, but not fine-grained. It is able to estimate how many times additions operators between integer operands were executed globally or per-function, but it cannot go further into details, for example it cannot estimate the cost of the operations within a critical loop. This approach is faster than an ISS, but still significantly inefficient more: In fact, it employs an instrumentation technique which promotes the input C code to C++, so that overloaded version for all the operators can be provided, which contain profiling code. This means that every time an operator is executed, also profiling code is executed; this is much more inefficient than basic block profiling, of which our instrumentation technique is an improvement.

Other solutions, such as the Seoul National University Energy Scanner (SES),¹⁷ determine the system's energy consumption by performing actual electrical measurements. The approach satisfies all the requirements on speed, dynamism, source level, and fine detail, but it must be considered a *measurement* rather than an *estimation* technique. It may reach the highest possible accuracy, but it requires a complex hardware setup, which includes a host PC and a guest ARM board with low-noise power measurement equipment and a dedicated PCI interface. This makes the solution unfit for exploration on different platforms, or for platforms for which the models are available but the silicon is not.

Finally, we do not discuss thoroughly black-box techniques (such as the ones by Muttreja et al.¹⁸ or Brandolese et al.¹⁹) because they are intended to solve a different problem than ours. In fact, they aim at estimating the consumption of code whose source is not available, such as library function calls and operating system calls. Their output is a static model which does not provide any link to source-level entities. On the other hand, our technique is designed for a seamless integration with black box techniques: It can easily associate a cost derived from a black-box model to those source-level entities which correspond to library or operating system calls.

3. SOURCE LEVEL ANALYSIS

The core of our methodology is the determination of the time and energy cost syntax elements, more precisely of each node in the Abstract Syntax Tree (AST) of the input source code. We assume the simplifying hypothesis that the execution cost (C_i) of a node i can be expressed as the product of its single-execution cost (c_i) and its execution count (n_i):

$$C_i = c_i \cdot n_i \quad (1)$$

This approximation is valid within negligible errors because single-issue processor cores comprise components with a limited latency variability. The same approximation

would lead to larger errors if caches were included in the estimates.

We compute single-execution costs during a static analysis performed via an attribute grammar, based on an abstract translation model, and we determine execution counts by running an instrumented version of the original program over actual input stimuli. The overall process is illustrated with an example in Section 3.1, while the determination of single-execution costs and execution counts are described in Sections 3.2 and 3.3, respectively.

3.1. An Illustrative Example

This section illustrates how the proposed technique operates by means of an example, the FFT benchmark.²² For sake of brevity, we show the steps of our technique only on the following fragment of code:

```

74 for (i=rev=0; i < NumBits; i++)
75 {
76     rev=(rev << 1) | (index & 1);
77     index >>= 1;
78 }
    
```

In our technique, we first parse the code, obtaining the portion of the AST represented in Figure 1. For each node in the AST, we compute its single-execution cost as we describe in Section 3.2. Computing the single-execution costs results in associating to each node zero or more *atoms*. Atoms are source-level cost terms which are independent from the compiler and the architecture. It is possible to describe the cost of all the constructs of the

Table I. Single-execution costs expressed in terms of atoms.

Node #	Line	Single-execution cost
#117	74	1 Assignment
#118	74	1 Assignment
#121	74	1 IntRelation
#123	74	1 IntAdd
#139	75	1 For
#127	76	1 BitwiseShift
#130	76	1 BitwiseOperation
#131	76	1 BitwiseOperation
#137	77	1 BitwiseShift

C language in terms of a limited number of atoms (around 50, the definition of atoms is subject to some degree of freedom). The single-execution costs of the nodes in the example is reported in Table I. The attribute grammar evaluates the nodes' costs exploiting the available contextual information such as the type of variables and expressions, their constancy, the direction of assignments, the size of the transferred data, etc. This is why two instances of the same operator (e.g. assignments in nodes #118 and #132) may have different costs, depending on their context.

Then, we profile the original code in order to determine the execution count n_i for each node. Profiling is performed by running an instrumented version of the code over real input data. The use of real input data is an advantage over static techniques, which either always assume the worst case or interrogate the user. We generate the instrumented version with an original technique which ensures minimal overhead, which is described in Section 3.3.

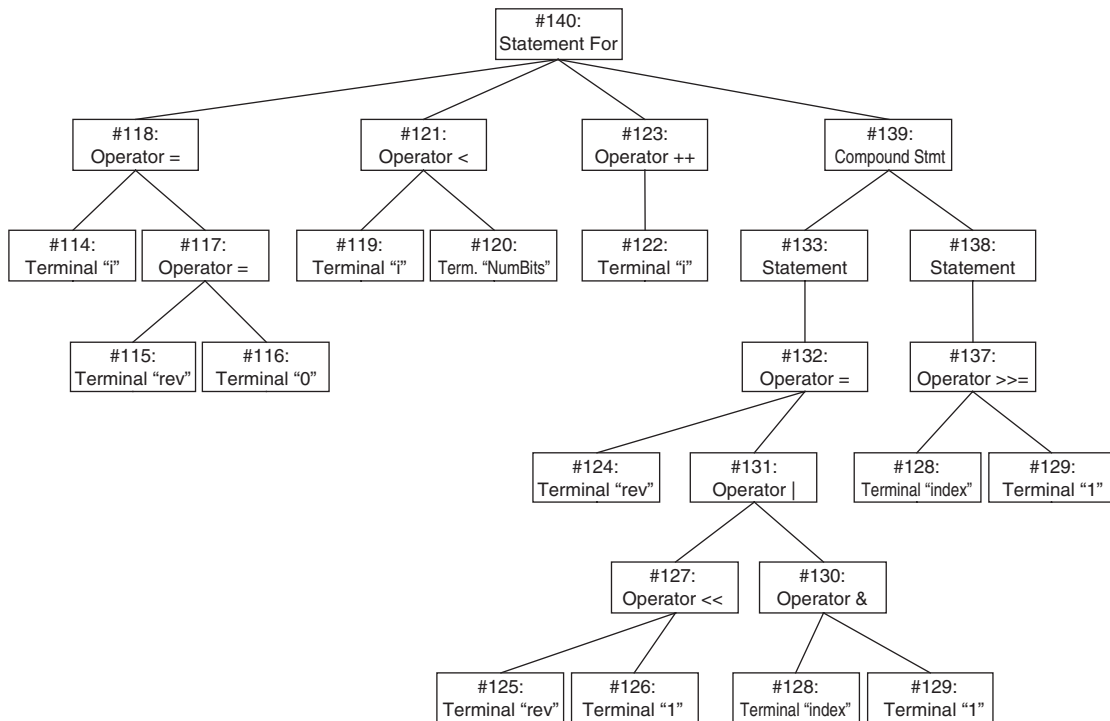


Fig. 1. The abstract syntax tree of the source code in the example.

Table II. Single-execution costs expressed in terms of abstract instructions.

Node#	Line	Single-execution cost
#117	74	1 alul
#118	74	1 alul
#121	74	1 cmpl
#123	74	1 alul
#139	75	1 jump
#127	76	1 alul
#130	76	1 alul
#131	76	1 alul
#137	77	1 alul

After profiling, we compute node costs C_i by multiplying n_i and c_i , then we convert these costs from atoms to *abstract instructions*. Abstract instructions are a set of micro-architectural level primitives which we have already introduced in Ref. [16]. They allow to describe any real instruction set in terms of energy- and time- costs. The mapping from atoms to abstract instructions depends on the compiler and the architecture. It is more convenient to use abstract instruction rather than real instructions because typical instruction sets comprise hundreds of instructions, whether abstract instruction sets comprise less than 10 instructions. Therefore, the number of measurements involved in the time- and energy- characterization of an entire real instruction set makes the operation impractical, while characterizing an abstract instruction set involves a limited number of measurements. All these claims are detailed in our previous work.¹⁶

The mapping from atoms to abstract instruction models the behavior of a given compiler. In such a model, an atom may correspond to zero or more abstract instructions. In our experiments, we have modeled the GCC compiler version 2.95 for ARM7 (with optimization flags-O2). Within this model, the single-execution costs corresponding to the nodes in the example map to abstract instructions as reported in Table II. Finally, each abstract instruction corresponds to an average number of clock cycles and an average consumed current during its execution, both determined by measurements. The final costs expressed in physical quantities appear as in Figure 2.

3.2. Determining the Cost of Syntax Elements

Our technique associates a single-execution cost to each node in the AST of the input source code. The task is

performed by a multi-visit attribute grammar which is based on cost models for the compiler and for the underlying architectures. This section describes first these models, then the details of the attribute grammar.

Our architecture model accounts for the time and energy costs due to the CPU core (datapath and register file) and is an adaptation of Ref. [16]. The model architecture is a single-issue, single-stage ideal executor, with an infinite number of general-purpose registers. This executor executes, one at a time, abstract instructions, which have a given latency and average current. This simplified model is simple to handle, but flexible enough to model accurately an arbitrary single-issue architecture in terms of time and energy costs. The model architecture is of load-store type, with three-operand ALU instructions. Any real instruction with complex addressing modes and explicit operands can be rewritten in a sequence of abstract instructions with identical global cost and semantics. For ease of model tuning, we partition the abstract instructions in 7 classes (alul, aluh, mvld, mvst, jump, cmpl, cmph), and we assume constant latency, absorbed current, and code size within each class. We do not estimate the impact of the cache hierarchy: Dedicated techniques exist in literature^{10,11} which are complementary and mutually helpful with ours. Our memory space is flat and it coincides with the register space, which can be accessed by name or by address. Scalars and temporaries are mapped onto fixed registers in the abstract compilation, and instructions operate on them ‘by name’ (register-direct addressing mode). Arrays and pointed objects are explicitly processed with load/store instructions, thus exposing memory operations.

We employ a compiler model which realizes a good trade-off between simplicity and accuracy (see Fig. 3). It is an original contribution of this paper. It consists of an abstract translation model which generates assembly code directly from the AST. The model is formally defined as an attribute grammar, it is consistent with the architectural model, and it approximates the effects of real instruction selection algorithms. It accounts for simple versions of architecture-independent optimizations such as constant folding, null sequence elimination, strength reduction. The model could be either made more complex or corrected with statistical tuning in order to account for the effects of spilling, coalescing, and other optimizations, such as common sub-expression elimination, constant propagation

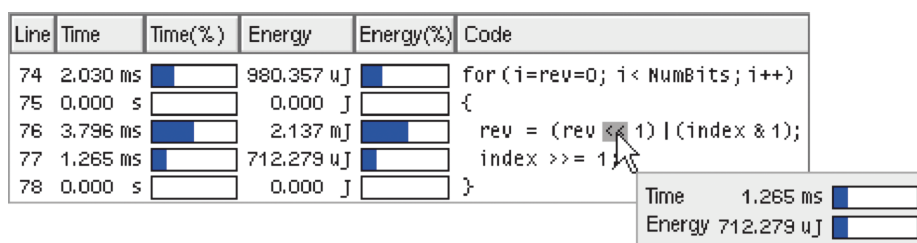


Fig. 2. Result of the estimation of the code in the example.

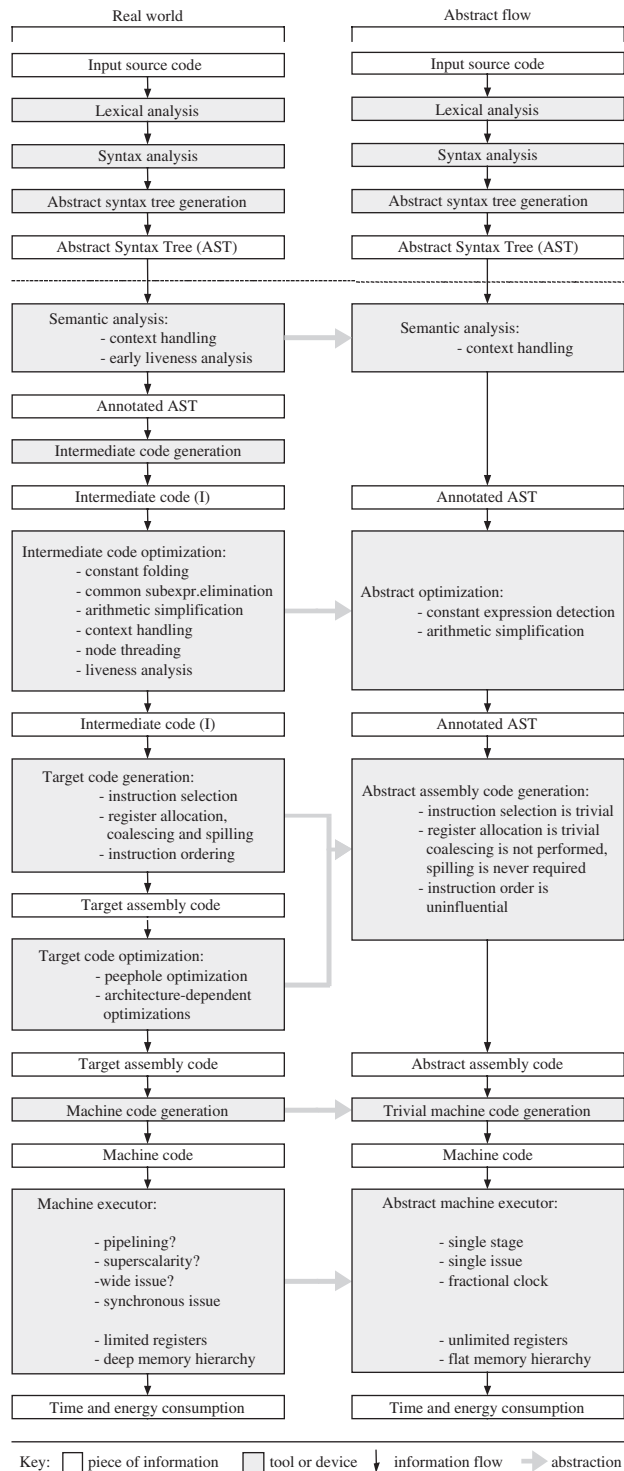


Fig. 3. The compiler and machine models we adopt.

or register promotion. Nevertheless, our experience with GCC 2.95 for ARM suggests that the small prediction errors do not justify any additional effort to refine the model. In other cases (different compilers and architectures), the model is open to extensions and statistical corrections. We deliberately do not model those optimizations which can be easily expressed (or are tradi-

Table III. Attributes in the attribute grammar which computes single-execution costs.

Attribute	Meaning	Type	Defined for:	
			Expressions	Statements
c	Single-execution cost	Synthesized	✓	✓
ci	Inherent cost	Synthesized	✓	✓
cc	Conversion cost	Synthesized	✓	✓
cf	Flow-control cost	Inherited		✓
t	Type	Synthesized	✓	
r	Restricted type	Inherited	✓	
k	Constancy	Synthesized	✓	
e	Constant value	Synthesized	✓	
v	Valueness	Inherited	✓	
b	Register-boundedness	Synthesized	✓	
f	Translation flavour	Inherited	✓	✓

tionally expressed) at the source level (i.e., some loop optimizations, code motion/replication, software pipelining, function-inlining, scalar replacement...) to provide the designer with the possibility to explore their effects at the source level, by comparative runs of our flow. Further details of the compiler model are available in Ref. [20].

The computation of single-execution costs according to the above models is performed by an attribute grammar composed by the 11 attributes, described in Table III. We express the single execution cost as the sum of an inherent cost ci , a conversion cost cc , and a flow control cost cf :

$$c_i = ci_i + cc_i + cf_i \quad (2)$$

The inherent cost expresses the cost of carrying out operations and transferring data as required by the semantics of the current node. The conversion cost expresses the cost of performing the type conversions (explicit or implicit) as required by the C language for the current node, depending on its context. The flow control cost expresses the cost of jumps which may be involved in carrying out the evaluation of the node, including short-circuit evaluation. We compute the value of these costs according to rules which, in turn, depend on properties of the same node, and possibly its father and children nodes. The attributes and the rules are an original contribution of this work. A brief explanation of the remaining attributes follows.

Attributes k tells whether an expression is constant-value or not. If it is, then e assumes that constant value. We take the effort of fully determining constant expressions, because constant expressions are resolved and eliminated at compile-time by compilers; therefore they have no execution cost.

Attribute t is the result type of a node, represented in an appropriate type representation which is beyond the scope of this paper. To resolve t , we implement the entire type system and declaration semantics of the C language. Type information is necessary to evaluate the cost of operations which depend on operand type (e.g., arithmetics),

and to determine when conversions take place, and what is their cost.

Attribute v is the *valueness* of an expression, that is, whether its R-value or the L-value (or both, or none) are used. In memory-oriented operations, the valueness determines if a read or a write operation is performed, and allows to distinguish associated costs. Immediate right operands of simple assignments '=' have 'L' valueness, while immediate right operands of compound assignments (e.g. '+ =', '- =', '* =', ...) have 'RL' valueness, and operands of unary '&' have no valueness. In all the other cases, the valueness is 'R.' These rules are associated with a number of the exceptions, whose discussion is beyond the purpose and the space constraints of this paper.

Attribute r , the *restricted type*, is required to define the cost of expressions involving the dot '.' operator, which exhibits unique anomalies. r is the type of the value actually subject to a data transfer operation in the assembly translation of the father of a dot node, which is a (non strict) subset of t . In case of a struct member assignment, it is r who determines its cost, rather than t .

Attribute b , the *register-boundedness* is required to distinguish assignments which have no cost (e.g., due to coalescing) from ones which actually involve transfer of data.

The above attributes are sufficient to fully determine ci and cc . When the cost of flow-control instructions (cf) is concerned, another attribute is needed: f , the *translation flavor*. The value of cf is zero for expressions which do not cause splits or joins in the control flow (e.g., arithmetic operators), and nonzero for logical and relational expressions, and for iteration, selection, jump, and call instructions. cf depends on how a node was translated depending on its context, i.e. on f . The same expression may be translated differently depending on whether its numerical or logical value is relevant in its context. Knowing which flavor is selected is important because flavors differ in cost. For example, the translation of expression 'a+b,' when appearing in a larger arithmetic expression, such as '(a+b)*c,' has a translation consisting of an add instruction, leaving its result in a register to be used by the translation of the '*' operator. On the other hand, when 'a+b' appears as a condition of an 'if' statement, its translation comprises a conditional jump instruction, which leads the execution flow either to the 'then' or 'else' branches. The computation of the actual flavor of each symbol is done according to the abstract translation model described below.

The evaluation of attributes f and cf relies on a formal model of translation from C code to abstract instructions which we created. This model also is expressed as an attribute grammar (not to be confused with the previous one) defined over the AST of the C language, with four synthesized attributes: T , R , TT , and TF , which can be evaluated in a single, bottom-up pass. Statement nodes only have one single-entry single-exit translation, stored in attribute T . Expression nodes also have T , and

also two single-entry double-exit translations, attributes TT and TF , called the *jump-if-true* and *jump-if-false* translations, respectively. T is used when the numerical value of the expression is required (as in the example '(a+b)*c' above), which is left in a register R . TT , and TF are used when the logical value is required (as in 'if (a+b) ...') Our model describes how to compute T , R , TT , and TF of a node in terms of the same attributes of its child nodes, providing a complete translation scheme from C to abstract assembly, with register allocation and instruction selection. The scheme is short-circuit correct, free from jumps to jumps and useless move instructions, and we have verified its consistency against the output of GCC over the ARM, Intel 486, and UltraSparc architectures.

The complete set of rules for the determination of the attributes in the above attribute grammars, together with all the details which have been omitted for sake of brevity, are available in Ref. [20].

3.3. Determining Execution Counts

Our technique needs to determine the execution count n_i per each node. To do so, it prepares an instrumented version of the original code, compiles it, and runs it over realistic input data. This section describes the method we employ in order to generate this instrumented code.

We propose a profiling method, named *generalized basic block profiling*, which yields exact rather than approximated execution counts (as methods based on statistical program counter sampling do), can be performed on a host platform rather than the target platform, and guarantees minimal runtime overhead. It is an improved version of Ball-Larus basic block profiling.²¹ In partitions all the nodes in classes named *Generalized Basic Blocks* (GBB), and selects for profiling one node in each class, named *pivot*. Generalized basic blocks are maximal classes of nodes which are always executed the same amount of times, irrespectively of their position in the source code. This technique is more efficient than basic block profiling because GBBs are, in general, larger than basic blocks. For example, consider the fragment of code in Figure 4(a), where we assume that there are no other flow control statements (e.g., goto, break, continue, return statements, or setjmp()/longjmp() system calls). With basic block profiling, this code corresponds to 7 basic blocks, so 7 probes are inserted, as in Figure 3(b). On the other hand, generalized basic block profiling recognizes that code in sections A, D, and F is executed always the same number of times, and thus only 5 probes are inserted, see Figure 3(c). With reference to the example of Section 3.1, the nodes in that source code belong to three different generalized basic blocks, as shown in Figure 5 (where node #81 is a node which appears before the fragment).

Once the GBBs and the pivots are determined, our technique rewrites the original code in order to insert profiling calls at each pivot. When a pivot is a statement "s;," it

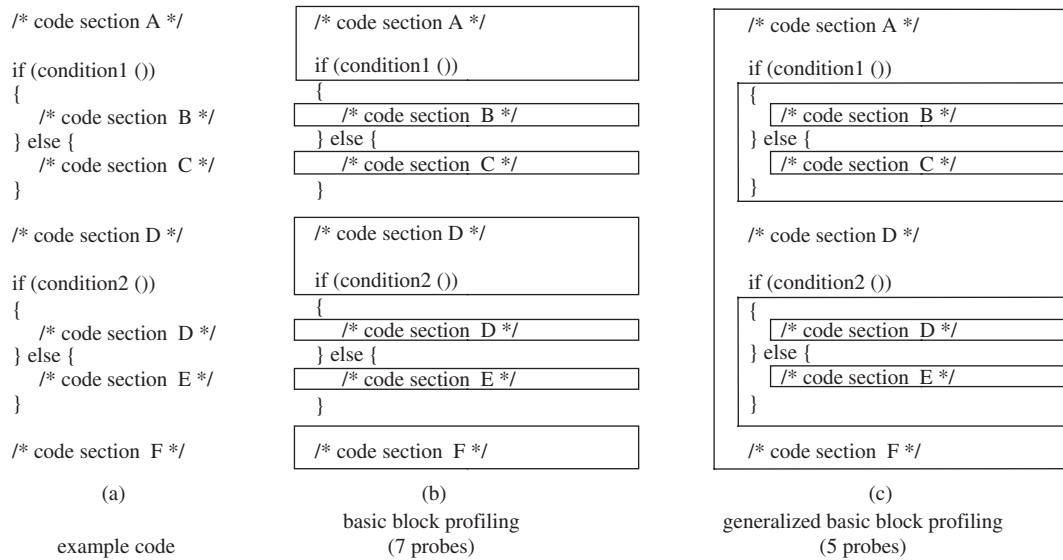


Fig. 4. Example illustrating generalized basic block profiling.

is rewritten as a compound statement “{__profile__(nnn); s;},” where __profile__() is function which increments the execution count for the node whose number is passed as an argument. Because of conditional expressions and short-circuit evaluation, also expression nodes can be pivots. When a pivot is an expression “e,” it is rewritten within a comma expression “(__profile__(nnn), e).”

For example, the code fragment from Section 3.1 is rewritten as follows:

```

74 for ( i = rev = 0; (__profile__(121), i < NumBits);
      (__profile__(123), i++))
75 {
76     rev = (rev << 1) | (index & 1);
77     index >>= 1;
78 }
    
```

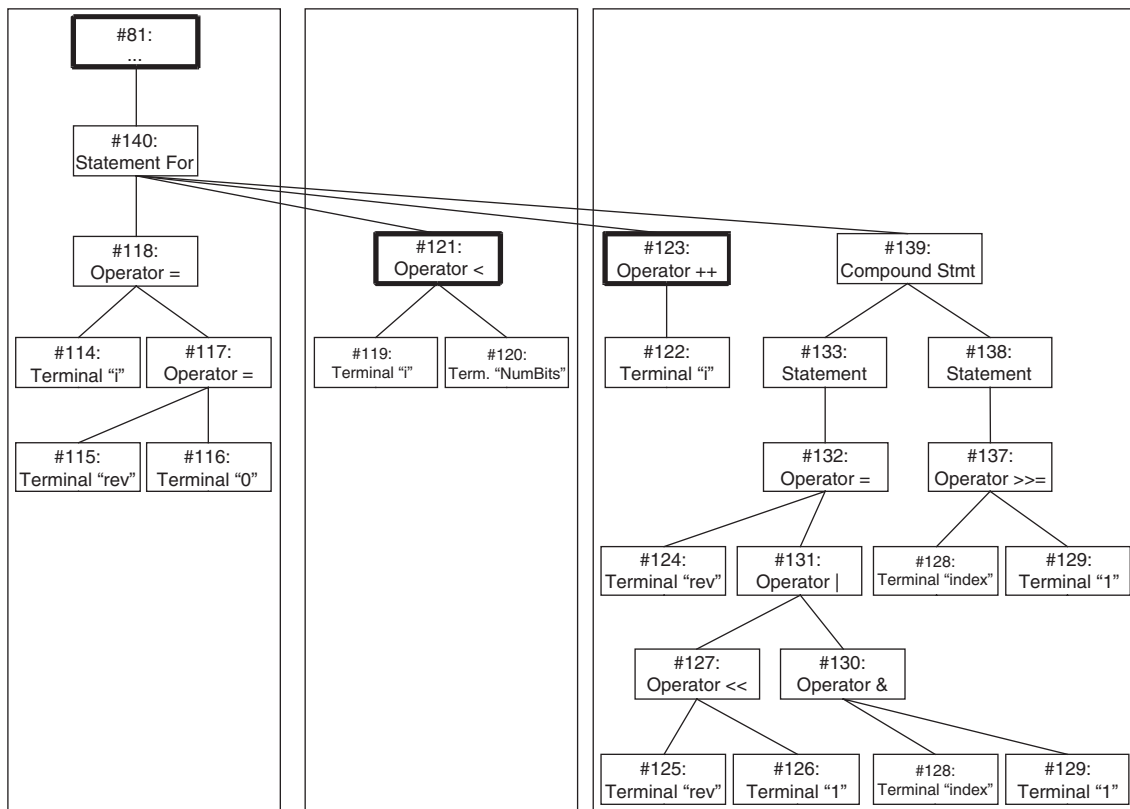


Fig. 5. The generalized basic blocks corresponding to the code in the example. A thicker frame denotes the nodes selected as pivots.

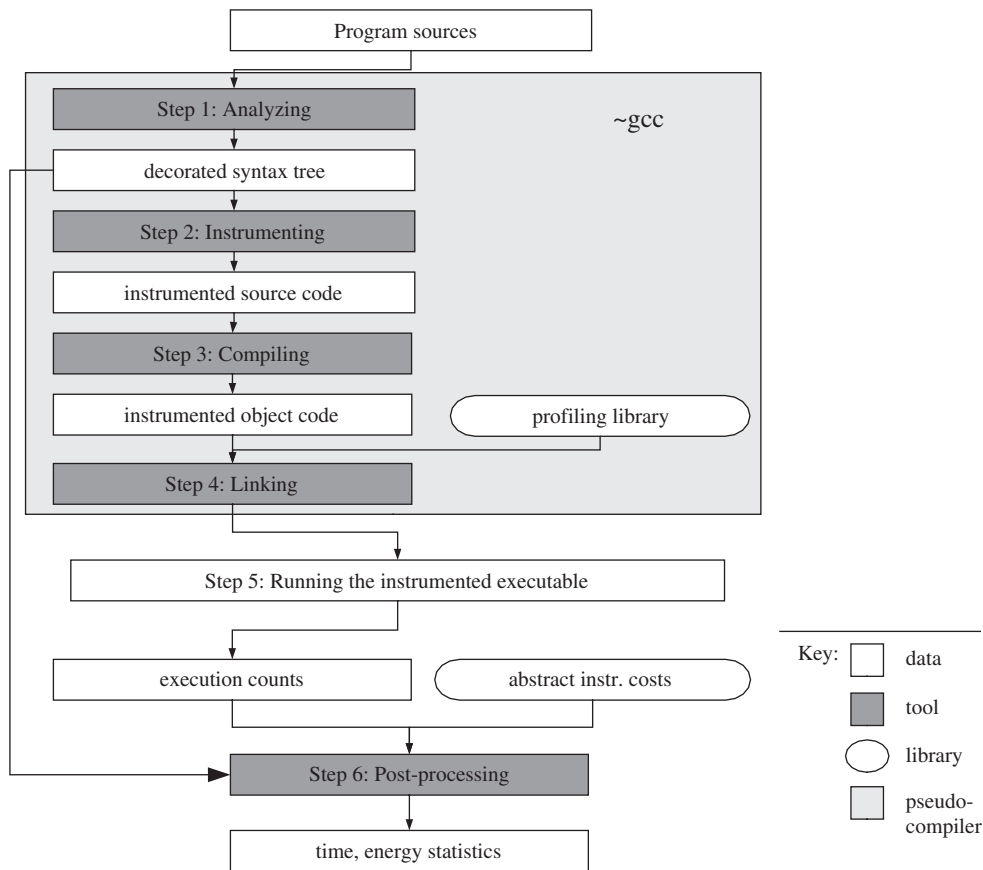


Fig. 6. The tool flow which implements our proposed methodology.

Only two profiling probes are inserted. Nodes #117 and #118 are executed the same number of times as node #81 (in the code before the fragment, not reported here), while all the nodes corresponding to the loop body are in the same GBB as node #123 (as from Fig. 5).

Note that these transformations are transparent: The resulting code preserves the same semantics. On the average, the application of this technique results in GBB containing approximately 100 nodes. This means that only 1 probe is executed every 100 nodes, which results in simulation times only 2.2 times longer than the original code.

4. EXPERIMENTAL RESULTS

In this section we describe the setup and the results of the experiments we conducted in order to evaluate the accuracy of our methodology.

We have implemented it in the form of a tool flow (Fig. 5), which externally acts as the popular compiler GNU GCC. This allows current projects to be built transparently within our framework, with their original make-files and source files, without any modification by the user. The tool flow accepts as an input any ANSI C 90 compliant source code. Our tool determines the AST of the input code, decorates nodes with their single-execution consumptions, and selecting the pivots for profiling. Then

it rewrites an instrumented source code with profiling calls inserted at each pivot. The code is compiled with a regular compiler, linked against our profiling library and run on real data. Finally, profiles are post-processed, determining the consumption of all the nodes in accordance with models described.

Our methodology is general and could be adapted to any single-issue processor. Nevertheless, we choose a specific architecture and simulator in order to evaluate its accuracy. We have chosen SimIt-Arm.⁷ It simulates a popular processor and it is currently the reference point in ISSs for the embedded design community. We employ SimIt Arm v. 2.0.3, which includes the GNU GCC v. 2.95 compiler. We have augmented SimIt results with absorbed current data, reusing the same platform information as JouleTrack.⁸ A platform hosting a StrongArm SA-1100 processor running at 206 MHz and with a supply voltage of 1.5 V. We have compared our flow against SimIt over a set of benchmarks taken from MiBench,²² the reference benchmark suite in the embedded community which includes automotive, consumer, network, office, security, and telecom applications. Note that SimIt simulates entire applications, including the C library (which must be statically linked) and user-mode portions of operating system calls, while our flow focuses on the application, for which the source code is given. Therefore, we have employed

Table IV. Experimental results: Estimated versus reference energy and execution time for the benchmarks.

Benchmark	Energy (mJ)			Execution time (ms)		
	Ref.	Estim.	Error (%)	Ref.	Estim.	Error (%)
adpcm-s	46.1	41.9	-9.1	166.3	156.4	-6.0
adpcm-l	910.2	722.1	-20.7	3289.9	2710.5	-17.6
bitcount-s	65.7	55.0	-16.3	242.8	204.0	-16.0
bitcount-l	981.9	977.1	-0.5	3628.6	3649.2	+0.6
blowfish	1067.0	748.3	-29.9	3742.7	3371.0	-9.9
CRC32	38.3	35.4	-7.5	132.2	129.6	-2.0
FFT-s	207.9	207.1	-0.4	764.6	770.3	+0.7
FFT-l	3213.2	3264.8	+1.6	11851.5	12142.5	+2.5
IFFT-s	205.1	207.3	+1.1	755.1	771.0	+2.1
IFFT-l	3181.8	3266.2	+2.7	11744.7	12147.8	+3.4
jpeg	87.9	91.2	+3.8	309.9	328.5	+6.0
rijndael	63.8	71.4	+12.0	221.3	257.3	+16.3
sha-s	22.1	21.9	-0.9	78.9	78.6	-0.4
sha-l	229.4	224.7	-2.1	820.0	818.3	-0.2
susan	133.2	128.0	-3.9	477.9	473.0	-1.0

those benchmarks where the overall weight of library and operating system calls was either negligible or easily estimable separately. We report in the results in Table IV, with the full list of the benchmarks we employed, together with their consumed energy and execution time estimated, respectively with SimIt-Arm and our methodology. Figures show close correspondence between reference and estimated data. The estimates exhibit good quality-of-result indicators: The coefficients of correlation between real and estimated data are 0.9960 for energy and 0.9987 for execution time. The average modulo percentage errors are 7.49% for energy and 5.65% for execution time. As far as simulation speed is concerned, statistics over the same benchmarks show, for our source-level flow, simulation times which are on an average 10,350 times shorter than SimIt-Arm. In short, 1 second of source-level simulation replaces 3 hours of ISS simulation.

5. CONCLUSION

This paper describes a technique to estimate the time and energy consumed by each element of a given source code, when the program is run with given, real input data over a single-issue processors. The technique is devoted to the core portions (datapath and register file) of the processor, and its estimates can be easily integrated with estimates regarding the memory hierarchy, generated with dedicated tools in literature.

The method is based on an abstract translation model, which formally describes how arbitrary C code is translated to a set of abstract assembly instructions. It proved to be accurate and remarkably faster than instruction-set simulation. It provides to the designers a detailed insight in the source-level causes of energy and time consumption. Additionally, it externally operates as regular GCC compiler, therefore it can be applied transparently to most existing

projects with no changes. Finally, it does not require the availability of the platform onto which the code will be targeted, but rather relies on a pre-characterization of that platform.

References

1. P. Puschner and C. Koza, Calculating the maximum execution time of real-time programs. *J. Real-Time Systems* (1989), Vol. 1, pp. 159–176.
2. K. Suzuki and A. Sangiovanni-Vincentelli, Efficient software performance estimation methods for hardware/software co-design. *Proceedings of the 33rd Conference on Design Automation*, Las Vegas, Nevada, USA (1996), pp. 605–610.
3. S. Malik, M. Martonosi, and Y. T. S. Li, Static timing analysis for embedded software. *Proceedings of the 34th Conference on Design Automation*, Anaheim, California, USA (1997), pp. 147–152.
4. K. Chen, S. Malik, and D. I. August, Retargetable static timing analysis for embedded software. *Proceedings of the International Symposium on Systems Synthesis*, Montréal, Québec, Canada (2001), pp. 39–44.
5. A. Hergenhan and W. Rosenstiel, Static timing analysis of embedded software on advanced processor architectures. *Proceedings of Design, Automation, and Test in Europe*, Paris, France (2000), pp. 552–559.
6. D. Brooks, V. Tiwari, and M. Martonosi, Wattch: A Framework for architectural level power analysis and optimizations. *Proceedings of the 27th International Symposium on Computer Architecture*, Vancouver, British Columbia, Canada (2000), pp. 83–94.
7. W. Qin and S. Malik, Automated synthesis of efficient binary decoders for retargetable software toolkits. *Proceedings of the 40th Design Automation Conference*, Anaheim, CA, USA (2003), pp. 220–225.
8. A. Sinha and A. P. Chandrakasan, JouleTrack: A web-based tool for software energy profiling. *Proceedings of the 38th Design Automation Conference*, Las Vegas, NV, USA (2001), pp. 220–225.
9. T. Simunic, L. Benini, and G. De Micheli, Energy efficient design of battery-powered embedded systems. *Special Issue of Transaction on Very Large Scale of Integration Systems* (2001), Vol. 9, pp. 18–28.
10. J. Bormans, K. Denolf, S. Wuytack, L. Nachtergaele, and I. Bolsens, Integrating system-level low power methodologies into a real-life design flow. *Proceedings of the 9th International Workshop on Power and Timing Modeling Optimization and Simulation*, Kos, Greece (1999), pp. 19–28.
11. G. Arnout, PowerEscape, Maximizing data efficiency for power and performance. *White paper* (2005), <http://www.powerescape.com/technology/papers/>;
12. M. Lajolo, M. Lazarescu, and A. Sangiovanni-Vincentelli, A compilation-based software estimation scheme for hardware/software co-simulation. *Proceedings of the 7th International Workshop on Hardware/Software Codesign*, Rome, Italy (1999), pp. 85–89.
13. E. Senn, N. Julien, J. Laurent, and E. Martin, Power consumption estimation of a C program for data-intensive applications. *Proceedings of the 12th International Workshop on Integrated Circuit Design Power and Timing Modeling, Optimization and Simulation*, Seville, Spain (2002), pp. 332–341.
14. M. Ravasi and M. Mattavelli, High-level algorithmic complexity evaluation for system design. *J. System Architecture* (2003), pp. 403–427.
15. C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto, Energy estimation for 32-bit microprocessors. *Proceedings of the 8th International Workshop on Hardware/Software Codesign*, San Diego, California, USA (2000), pp. 24–28.

16. C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto, An instruction-level functionality-based energy estimation model for 32-bit microprocessors. *Proceedings of the 37th Conference on Design Automation*, Los Angeles, CA, USA (2000), pp. 346–351.
17. D. Shin, H. Shim, Y. Joo, H.-S. Yun, J. Kim, and N. Chang, Energy-monitoring tool for low-power embedded programs. *IEEE Design and Test of Computers* (2002), pp. 7–17.
18. A. Muttreja, A. Raghunathan, S. Ravi, and N. K. Jha, Automated energy/performance macromodeling of embedded software. *Proceedings of the 41th Design Automation Conference*, San Diego, CA, USA (2004), pp. 99–102.
19. C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto, Library functions timing characterization for source-level analysis. *Proceedings of the 2003 Design, Automation and Test in Europe Conference and Exposition*, Munich, Germany (2003), pp. 1132–1133.
20. Daniele Paolo Scarpazza, A Source-Level Estimation and Optimization Methodology for Execution Time and Energy Consumption of Embedded Software. Ph.D. Thesis, Dipartimento di Elettronica e Informazione, Politecnico di Milano (2006).
21. T. Ball and J. R. Larus, Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems* (1994), Vol. 16, pp. 1319–1360.
22. M. R. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, MiBench: A free, commercially representative embedded benchmark suite. *Proceedings of the 4th Workshop on Workload Characterization*, Austin, TX, USA (2001), pp. 3–14.

Daniele Paolo Scarpazza

Daniele Paolo Scarpazza received his M.S. in Electrical Engineering & Computer Science in 2001 from University of Illinois at Chicago. He received a M.S. and Ph.D. degrees in “Information Engineering” in 2002 and 2006, respectively from Politecnico di Milano.

Carlo Brandolese

Carlo Brandolese received his Degree in Electronic Engineering in 1995 from Politecnico di Milano. He has been working until 1997 at Central R&D Labs of the Italian telecom company Italtel as CAD Engineer, being responsible of the FPGA design flows and methodologies. He received in 1998 his M.S. in Information Technology from Cefriel (Politecnico di Milano) working on system-level design and co-design issues. Finally, in 2001, he earned his Ph.D. in Information Technology and Design Automation from Politecnico di Milano with a dissertation on the analysis and optimization of power consumption of heterogeneous embedded system. Since 1998 he is consultant at Cefriel in the Embedded System Design Unit and since 2003 he is assistant professor at Politecnico di Milano. His current interests range from low-power design, software power analysis and optimization, system-level design methodologies and co-design.