Source-Level Estimation of Energy Consumption and Execution Time of Embedded Software

Carlo Brandolese
Politecnico di Milano - DEI
Piazza Leonardo da Vinci, 32, Milano, Italy
carlo.brandolese@polimi.it

Abstract

Energy optimization of embedded software is of primary importance. Nevertheless, there is lack of accurate and usable methodologies and tools to estimate software performance (execution time, energy) and to allow a significant exploration of design alternatives. Current approaches use either instruction-level simulation (accurate but slow), or static-time source characterization (flexible but data-independent). This paper proposes a hybrid approach taking advantage of the strengths of both the above approaches. We present a fully automatic method for estimating the execution time and power consumption of a C program—run on a given architecture on given input data—based on statistically-accurate models for the architecture and for the compiler. Validation results against an ARM energy-enabled instruction-level simulator show an average absolute relative errors of 8.5%.

1. Introduction

Power consumption is a critical parameter in the design of embedded systems. In fact, the energy-efficiency of embedded software can determine the commercial success, and in cases indeed the feasibility of battery-operated devices. Nevertheless, embedded software designers have limited means to explore optimization alternatives.

Traditionally, power-enabled instruction-level simulators were used for the purpose. They are accurate, but very slow, and they provide little or no advice on how to modify a given source code in order to improve it. Since source-level transformations have been shown to produce the highest energy gains, it is critical that the designer has an accurate and fast source-level estimation engine. Performing source-level estimation is inherently difficult, because at the source-level it is not possible nor desirable (for performance and complexity reasons) to directly account for a number of compiler and architecture related effects. These effects include compiler optimizations, impact of memory hierarchy, pipelines and multiple functional units. We account for them effects with statistically-accurate models. To the best of our knowledge, our approach is the first to associate a cost to *each* entity at the source code level, such as operator, node, line of code and function. The original contributions supporting our approach are:

- a source analysis method which is fine-grained enough to allow accuracy in energy estimation and coarse-grained enough to allow efficient profiling, and general enough to model different compiler behaviors, and target architectures and instruction sets;
- an analytical model for the cost of source code execution which formally supports it.

More in detail, the main advantages offered by the methodology are the following:

- performance is several orders of magnitude better than a power-enabled instruction-set simulator;
- obtained statistics and measurements are related to source-level entities, therefore of immediate help for the developer;
- once the method is tuned, estimation does not require the availability of test boards or simulators;
- estimates can be obtained also for cores where architectural details (required to build a simulator) are not available.

To achieve this, the analysis has been structured in three phases:

1. The parsetree is analyzed, and elementary source-level cost contributions (*atoms*) are associated to nodes.

978-0-7695-3277-6/08 $25.00 © 2008 IEEE
DOI 10.1109/DSD.2008.43

2. An optimally-instrumented version of the code is rewritten, compiled and executed for profiling.

3. A post-processing engine applies compiler-dependent and architecture-dependent models of time, energy and space costs to the profile, determining the consumption of each node, line of code, function, file, etc.

This paper is organized as follows. In Section 2 we give a survey of the related work. In Section 3 we illustrate our methodology works by applying it to an example source code fragment. In Section 4 we give a mathematical formalization of the method. In Section 5 we present our instrumentation techniques that enable high performance profiling. In Section 6 we describe how we implemented the methodology in a usable tool flow. Section 7 illustrates experimental results. Section 8 concludes the paper.

2. Related work

Traditionally, software consumption estimation is performed with energy and cycle-accurate *instruction-set simulators* (ISS, e.g. [1]). These simulators are precise, but very computationally demanding, and not useful to determine the consumption information of source-level entities.

A first attempt to raise the abstraction level of the estimates is made in [2], by *coupling* an ISS with coarse-grained estimates obtained with `gprof` (a profiling tool based on program counter sampling). This can resolve costs at the function level, but not at finer elements. The approach is good for obtaining rough estimates or comparing strongly different algorithms, but not helpful for optimization: experience shows that most of the time is spent inside loops entirely contained in one function, and the approach cannot resolve at that resolution. Splitting critical loops in functions just to allow measurement is not a solution because it seriously perturbs the estimates.

Another approach, called *compilation-based performance estimation*, is presented in [3]. It exploits a real compiler (namely `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, but is specific to the `gcc` compiler and, again, a way to redistribute energy to source-level elements is not provided.

SoftExplorer [4, 5] implements a fast technique to estimate the consumption of data-dominated loops directly from the C source, based on functional-level modeling of the architecture and statistical parameters extracted from the assembly code. Unfortunately the technique is static, therefore is relies on the user to determine the run-time execution flow (i.e. the number of iteration of a loop), which is impractical for programs of realistic size. Additionally, the method cannot be extended to control-dominated programs.

SIT [6] is the first real *2-way source-level technique*. By "2-way" we mean that not only energy information are estimated from source code, but that energy estimates are redistributed onto source-level entities, for the help of the developer. Here, like in our proposed approach, the source code is instrumented, though in a different way. More precisely, the input C code is promoted to C++ and overloaded instrumented versions of all the operators are provided. This avoids the difficulties of a complete parsing of the C language, but does not allow to resolve the energy contribution of single instances of syntax element, which we can do (e.g. "determine the cost of the '+' operator at line 123 of file `src.c`"). Additionally, our method is more efficient, since it does not necessarily imply the execution of profiling code at each operator invocation.

Determining the cost of the whole program also requires the knowledge about the cost of library function and operating system calls. In both cases the source code is usually not available, and dedicated *black-box techniques* should be applied [7, 8]. Our approach is designed since the beginning to be integrated with those techniques. All the above comparisons are summarized in Table 1.

3. Illustrative example

In this section we explain our method by illustrating how it applies to the fragment of code reported in Figure 1(a). First, the code is parsed according to the grammar of the C language [9]. This results in the parse tree shown in Figure 1(b). Secondly, the parse tree is decorated by associating zero or more language-level cost contributions (*atoms*) to each of its node. The result of the annotation is shown in Figure 1(c). Finally, as shown if Figure 1(d), each atom is translated into one or more virtual assembly-level instructions (*kernel instructions*).

Kernel instructions are a minimal set of abstract micro-assembly level instructions, not belonging to any specific microprocessor, that we introduced in our previous works [10, 11]. In that works, we proved that they can be used to characterize every architecture in terms of cost (execution time, energy absorption and memory size). There, we focused on assembly-level power estimation. The theory, methods and tools necessary to link with the source level are novel contribution of this paper. The decoration of nodes with atoms depends on the semantics of the language (e.g. as in [12]), whereas the kernel instructions related to each atom also account for the behavior of the specific compiler considered. Associating atoms to nodes as shown above requires a complete model of the grammar of the C language (association rules based on semantic properties of nodes) and a flexible tool-set to support the automatic application

	Cycle accurate	2-way static level	Dynamic phase	Static phase	Operator resolution
ISS	×			×	
ISS + `gprof`	×			×	
Compilation-based			×	×	
SoftExplorer			×		
SIT		×	×	×	
Proposed approach		×	×	×	×

Table 1. Comparison among different estimation techniques.

of those rules. In modeling the C language, all the 213 productions of the C syntax [9] have been analyzed. For each production in every possible execution scenario, an execution model in term of atoms has been identified. In some cases, this required considering additional parse-time information (e.g. nodes in the subtrees, data type of the result of sub-expression, etc.).

As an example, we consider the `for` statement, whose syntax is:

$$\begin{aligned}\langle \textit{iteration-statement} \rangle ::= \ & \texttt{for}_1\ (_2 \\ & \langle \textit{optional-expression} \rangle_3\ ;_4 \\ & \langle \textit{optional-expression} \rangle_5\ ;_6 \\ & \langle \textit{optional-expression} \rangle_7\ )_8 \\ & \langle \textit{statement} \rangle_9\end{aligned}$$

The translation scheme to produce kernel instructions is:

```
        τ(⟨optional-expression⟩3)
cond:   τ(⟨optional-expression⟩5)
        cond-jmp end
        τ(⟨statement⟩9)
        τ(⟨optional-expression⟩7)
        uncond-jmp cond
end:
```

where `cond-jmp` and `uncond-jmp` are the instructions for conditional and unconditional jump respectively, and the function $\tau(\cdot)$ returns the abstract translation of a node.

The execution cost of the above translation depends primarily on the actual execution flow occurred at runtime, as summarized below:

where the CJT, CJN and UJT are respectively the atoms representing the cost of conditional jumps taken and not taken, and unconditional jumps (always taken); in this case the translation to kernel instructions is trivial (kernel instructions exist for each atom) and will not be shown for sake of brevity.

Since the cost depends on the runtime execution flow, and the profile of each node accounts for its execution, it is possible to redistribute the cost of the `for` construct onto its children, in such a way that costs pertaining to the conditional portions of the parent are accumulated only when the corresponding child is executed.

Iterations	Cost
0	1 CJT
1	1 CJN + 1 UJT + 1 CJT
2	1 CJN + 1 UJT + 1 CJN + 1 UJT + 1 CJT
...	...
k	k CJN + k UJT + 1 CJT

Cost rules are expressed in a form that allows managing both *composition* and *hierarchy*. In the specific case of the `for`, the cost rules can be expressed as follows:

$$\begin{aligned}\text{cost}(\texttt{for}_1) &= CJT \\ \text{cost}(\langle \textit{statement} \rangle_9) &= \text{cost}(\tau(\langle \textit{statement} \rangle_9)) \\ &+ CJN + UJT\end{aligned}$$

A similar approach allows modeling all other control-flow statements. For expressions, cost rules are more complex than above, and include conditions on the type, size and level of indirection of the operands. For example, the cost of a `+=` operator is significantly different if the operands are both integer, both floating-point or mixed. These rules require the parser to support the full type system and scoping rules of the C language, because operand trees can be arbitrarily complex, and resolving the type of such expressions could mean traversing a number of symbol tables, `typedefs`, `structs` and `unions`. Support of the type-system is also needed to determine the cost of cast operations and type promotions/demotions.

All expressions that can be resolved at parse-time must be marked, and their run-time cost set to zero. In order to completely determine the size of data structures, the resolution of constant expressions with integer values must be determined at parse-time.

	Size	Available after	Comment
c_e, c_t	scalar	Post-processing	Final output
c_s	scalar	Post-processing	Final output, instrumentation/profiling not required)
S	scalar	Parsing	Depends on source code
s	$1 \times S$	Parsing	Depends on source code
P	scalar	Instrumentation	Depends on source code and on pivot choice strategy
p	$1 \times P$	Profiling	Depends on source, pivot choice, input
A, K	scalar	N/A	Fixed in the methodology
k_e, k_t	$1 \times K$	Modeling	Statistical model of the architecture
k_s	$1 \times K$	Modeling	Statistical model of the architecture and of the compiler
$\mathbf{R}$	$P \times S$	Instrumentation	Describes the choice of pivot nodes
$\mathbf{M}$	$S \times A$	N/A	Fixed in the methodology
$\mathbf{T}$	$A \times K$	Tuning	Statistical model of the compiler

Table 2. Summary of the mathematical symbols.

4. Mathematical model

In this section we formally define the cost model on which our method relies. Given a program run (the program and its input data), let c_e, c_t and c_s be the energy consumed by that execution (μJ), the execution time (clock cycles), and the space occupied by its binary code (bytes) respectively. For efficiency reasons, not all the nodes are instrumented, but only a representative subset, of $P = |p|$ nodes, called *pivots* (see Section 5 for details). Let S be the number of nodes in a given source code and $\mathbf{R}$ be the corresponding $(P \times S)$ *representation matrix*, defined as:

$$\mathbf{R} = \begin{bmatrix} R_{1,1} & R_{1,2} & ... & R_{1,S} \\ R_{2,1} & R_{2,2} & ... & R_{2,S} \\ \vdots & \vdots & \ddots & \vdots \\ R_{P,1} & R_{P,2} & ... & R_{P,S} \end{bmatrix} \quad (1)$$

The matrix indicates which nodes are represented by which pivot. An element $R_{i,j} = 1$ when the j-th node s_j is represented by the i-th pivot, otherwise $R_{i,j} = 0$. Each column contains exactly one element 1, since each node has exactly one pivot.

Let A be the number of atoms. Once the instrumented program is run, the profile $p = [p_1\ p_2\ ...\ p_P]$ is available. The element p_i is the execution count of the i-th pivot. From p and $\mathbf{R}$, it is possible to calculate the vector of execution counts of all nodes as:

$$s = p \cdot \mathbf{R} = [s_1\ s_2\ ...\ s_S] \quad (2)$$

where s_i is the execution count of the i-th source code node. As stated before, nodes are enriched by associating zero or more atoms to them.

Therefore, the node-to-atom *mapping matrix* $\mathbf{M}$ is:

$$\mathbf{M} = \begin{bmatrix} M_{1,1} & M_{1,2} & ... & M_{1,A} \\ M_{2,1} & M_{2,2} & ... & M_{2,A} \\ \vdots & \vdots & \ddots & \vdots \\ M_{S,1} & M_{S,2} & ... & M_{S,A} \end{bmatrix} \quad (3)$$

where A is the number of atoms. The i-th row of $\mathbf{M}$ accounts for the cost of the i-th node, calculated as:

$$cost(i\text{-th node}) = s_i \cdot \sum_{j=1}^{A} M_{i,j} \cdot cost(j\text{-th atom}) \quad (4)$$

The cost of each atom is, in turn, composed of the costs of the kernel instructions that the atom translates to. If K is the cardinality of the kernel instruction set, the *translation matrix* $\mathbf{T}$ is:

$$\mathbf{T} = \begin{bmatrix} T_{1,1} & T_{1,2} & ... & T_{1,K} \\ T_{2,1} & T_{2,2} & ... & T_{2,K} \\ \vdots & \vdots & \ddots & \vdots \\ T_{A,1} & T_{A,2} & ... & T_{A,K} \end{bmatrix} . \quad (5)$$

This matrix accounts for the behavior of a given compiler. The i-th row defines the cost of the i-th atom as:

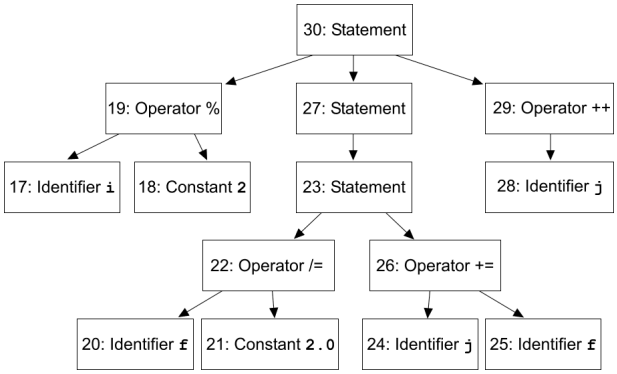
$$cost(i\text{-th atom}) = \sum_{j=1}^{K} T_{i,j} \cdot k_j \quad (6)$$

Statistical measurements performed on the target architecture provide energy, time and space costs for each of the kernel instruction, respectively:

$$\begin{array}{rcl} k_e & = & [\ k_{e1}\ \ k_{e2}\ \ ...\ \ k_{eK}\ ] \\ k_t & = & [\ k_{t1}\ \ k_{t2}\ \ ...\ \ k_{tK}\ ] \\ k_s & = & [\ k_{s1}\ \ k_{s2}\ \ ...\ \ k_{sK}\ ] \end{array} \quad (7)$$

```
8  if (i % 2) {
9    f /= 2.0;
10   j += f;
11 }
12 else
13   j++;
```

(a) Source code



(b) Parse tree

Node	Atom
19	1 Modulo
22	1 FloatDiv
26	1 MixedAddInt
27	1 CJN + 1 UJT
29	1 IntAdd + 1 CJT

(c) Nodes annotated with atoms

Node	Kernel instructions
19	1 alu-heavy
22	1 fp-div
26	1 fp-st + 1 fp-addsub + 1 fp-ldi
27	1 cond-jmp-n + 1 uncond-jmp-t
29	1 alu-light + 1 cond-jmp-t

(d) Nodes annotated with kernel instructions

Figure 1. Phases of the methodology.

The above definitions allow us to express the estimates of energy, time and space costs of a given program execution in a compact way, precisely:

$$c_e = p \cdot \mathbf{R} \cdot \mathbf{M} \cdot \mathbf{T} \cdot k_e^T \quad (8)$$

$$c_t = p \cdot \mathbf{R} \cdot \mathbf{M} \cdot \mathbf{T} \cdot k_t^T \quad (9)$$

$$c_s = \mathbf{1}_A \cdot \mathbf{M} \cdot \mathbf{T} \cdot k_s^T \quad (10)$$

Syntax element	Instrumentation technique
expr	`(` `p(`*id*`,` *expr* `)`
statement	`{` `p(`*id*`);` *statement* `}`
type fn`(`*args*`)` `{` *body* `}`	*type fn*`(`*args*`)` `{` `p(`*id1*`);` `{` *body* `}` `p(`*id2*`);` `}`

Table 3. Instrumentation transformations

These three equations, along with the definition given above of the matrices involved, summarize the proposed methodology. Table 2 lists the notation introduced in this Section.

5. Efficient instrumentation

Profiling nodes via source instrumentation requires a sufficient set of transparent probe-inserting transformations, able to reach any desired type of node. The set of transformation chosen is illustrated in Table 3, where `p(id)` indicates a call to the probing function. Expressions are included in parenthesized *comma expressions* while statements are wrapped in *compound statements*. The same happens to function bodies, with additional code added at every return point in order to allow call stack simulation, and per-invocation cost calculation. As anticipated, instrumentation of all nodes is unnecessary, highly inefficient and, in some cases, syntactically impossible. Instead, probes should be inserted in the minimum number of points sufficient to determine the execution count of every node in the program: therefore, an efficient instrumentation strategy is required.

Given a set of nodes $A = \{a_1, a_2, ...\}$, we define an *instrumentation* I over A as a partition of A where, for each subset A_i, one node p_i is chosen as pivot. More formally:

$$I = \{(p_1, A_1), (p_2, A_2), \ldots, (p_n, A_n)\} \quad (11)$$

such that:

1. $\bigcup_{i=1}^{n} A_i = A$,
2. $A_i \cap A_j = \emptyset \quad \forall i \neq j \in \{1, 2, \ldots, n\}$, and
3. $p_i \in A_i \quad \forall i \in \{1, 2, \ldots, n\}$.

An instrumentation is *safe* when all nodes inside a given A_i execute the same number of times. A *trivial* instrumentation I_T is that in which every node is a pivot:

$$I_T = \{(a_1, \{a_1\}), (a_2, \{a_2\}), ..., (a_n, \{a_n\})\}. \quad (12)$$

Conversely, an *optimal* instrumentation I_O must satisfy the following conditions:

```
/* code section 1 */
a = 3;
if (a % 2) {
  /* code section 2 */
}
/* code section 3 */
```

Figure 2. All sections are equivalent but complex calculations are necessary to discover this property.

1. I_O is safe, and
2. $\nexists I' \in \mathcal{I}(A) : I'\text{is safe } \wedge\ |I'| < |I_O|$.

It can be proved that if $I \in \mathcal{I}(A)$ is optimal, then $|I| = |A/\sim|$. A useful parameter to measure the quality of a given instrumentation is the *efficiency* $\eta(I)$ defined as:

$$\eta(I) = \frac{|A/\sim|}{|I|}. \tag{13}$$

According to this definition, I_T and I_O are the limiting cases with respect to the efficiency. It is, in fact, easy to prove that:

$$\forall I \in \mathcal{I}(A): \quad \eta(I_T) = \frac{|A/\sim|}{|A|} \leq \eta(I) \leq 1 = \eta(I_O) \tag{14}$$

A convenient formal way to define a *instrumentation strategy* is as an equivalence relation $\sim$ over elements of A. In fact, any equivalence relation over A induces a partition $A/\sim = \{A_1, A_2, \ldots\}$. Then, any arbitrary choice of the pivot for each A_i is irrelevant both for correctness and for efficiency. Therefore, determining a good instrumentation strategy basically consists in determining a good node equivalence relation $\sim$.

Very efficient $\sim$-relations induce large equivalence classes, but might be excessively complex to evaluate. Consider, for example, the excerpt shown in Figure 2. Nodes in code section 1, 2 and 3 should be all equivalent since the `if` condition is always true[1]. However, the determination of cases like above require a complex symbolic evaluation engine, which is beyond cost-effectiveness in the context of this work.

A traditional solution is basic block instrumentation [13]: given nodes a and b, it considers $a \sim b$ iff a, b belong to the same basic block. This is a good starting point, but at source-level further optimizations are possible. Figure 3 illustrates such a case: sections 1 and 4 are evidently

[1]provided that `a` is not declared `volatile` and no `goto` instructions involve sections 1–3.

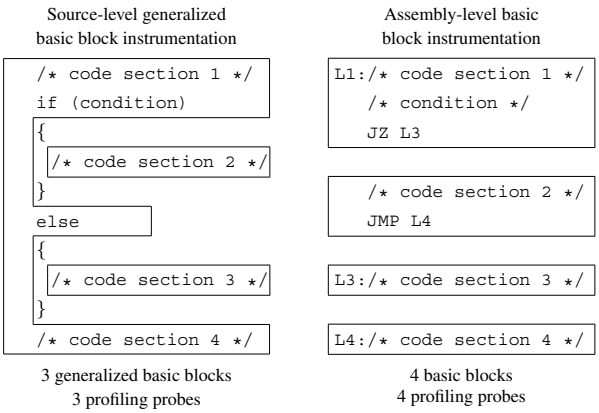


Figure 3. A sample section of code where generalized basic block instrumentation is more efficient than basic block instrumentation.

always executed together, but assembly-level tools cannot recognize this and would insert 4 probes, when only 3 are actually needed. The gain is evident in case of `if` chains.

The technique we propose is based on the definition of $\sim$-relation given below and we will refer to it as *generalized basic block instrumentation*.

Definition 1 *Two nodes A and B are* equivalent*, that is $A \sim B$, if and only if they belong to the same function,* and *it can be stated that they are always executed the same number of times without examining any conditional expressions.*

Hence, a generalized basic block is a maximum set of nodes that are executed the same number of times, and it is not required to examine conditional expressions to say that. The above definition can be proved to be safe; additionally, it is simple to check at parse time, and still very efficient. Our implemented instrumentation engine finds optimal instrumentations according to the above definition, and statistics over real code show that it generates instrumentations with equivalence classes containing on the average 100 nodes. This means that on the average, only 1 probe is executed every 100 nodes, resulting in run-times only 2 times slower than the original code and approximately 11,000 times faster than basic-block instrumentation.

6. Implemented tool flow

We completely implemented our methodology in the tool flow depicted in Figure 4. The flow externally acts as the GNU `gcc` compiler, thus allowing current projects, no matter how complex, to be compiled with their own unmodified `makefiles`. All projects in [14] compiled success-

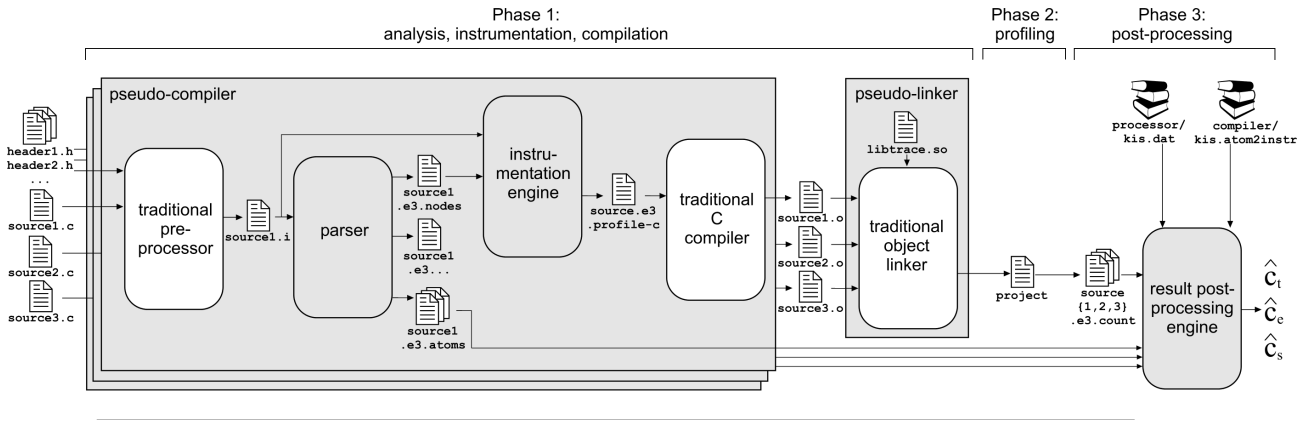


Figure 4. A representation of the proposed and implemented estimation flow. The "pseudo-compiler" box behaves as a C compiler, but internally decompose the code according to our methodology, instruments it accordingly and compiles it. The "pseudo-linker" box behaves like a traditional linker, only it adds our profiling library. Elements with a gray background have been specifically developed for this approach.

fully in our flow. Our `pseudo-compiler` tool preprocesses, parses and instruments input code according to our method, then compiles it with `gcc`. The parser, designed in `bison` [15] and `flex` [16] starting from a free ANSI C syntax [17, 9], is the most complex tool of the suite: it maintains a complete type system for all the identifiers, attributes atoms to nodes and select pivots. Our `pseudo-linker` stealthily adds a library containing the implementation of the probe function. The resulting executable is equivalent to its non-instrumented version, only it generates profiling information. That information is processed by our post-processing tool, which actually performs the numerical calculation of estimates via Eqn. (8)-(10). Incidentally, we internally use and externally mimic `gcc`, rather than another compiler, just for convenience (i.e. it is free software). This has no impact on the generality of our method: our models can be tuned on any compiler (e.g. in our experiments a Norcroft compiler was modeled). Also we can use any compiler in our flow, because the the instrumented code is compiled only to determine profiles, and no compiler optimizations can affect them.

7. Experimental results

We have conducted experiments to evaluate the accuracy and the performance of our estimation flow. We used an energy-accurate version of ARMulator 2.10 as our golden model (the same used in [18], tuned on parameters from [2, 19]). Experiments were run on an HP L2000-44 system, equipped with four PA8500 CPUs running at 440 MHz, 4 GB of RAM, the HP-UX 11.0 operating system, GNU gcc 2.95.2 compiler (used for our flow), Norcroft ARM C compiler 4.90 (used in compiling ISA-simulated benchmarks).

To evaluate the efficiency of our flow, we simulated seven benchmarks from MiBench [14] with both the power-enabled ARMulator and our tool set. Results, summarized in Table 5, show an average speedup of roughly 11,000 times. Moreover, the instrumented code runs approximately only 2 times slower than original code.

To evaluate the accuracy of our methodology, we trained our flow on a number of training source code fragments (like the *scenarios* in [4]) and the respective outputs from ARMulator. The obtained energy and time data have been used for least-square fitting of the elementary cost contributions k_e and k_t. Then, the tuned estimation flow has been validated over significant portions of industrial benchmarks (where library function calls were eliminated), exploiting the kernel instructions characterized at the previous step. The results regarding the 35 cases used for validation are plotted in Figure 5, and numerical indicators summarizing the quality of results are reported in Table 4. In each of the plots, for each validation case, the real time or energy is plotted on the horizontal axis, and the corresponding data (as estimated by our flow) on the vertical axis. A diagonal line is drawn to visually help the reading of the graph. Points above the diagonal are overestimates; points below are underestimates. As figures suggest, there is close correspondence between real and estimated data. The average modulo relative error is 8.41% for clock cycles and 8.50% for energy.

In order to assess the gaussianity of the estimation error, which is a desirable property, we plotted the cumulative

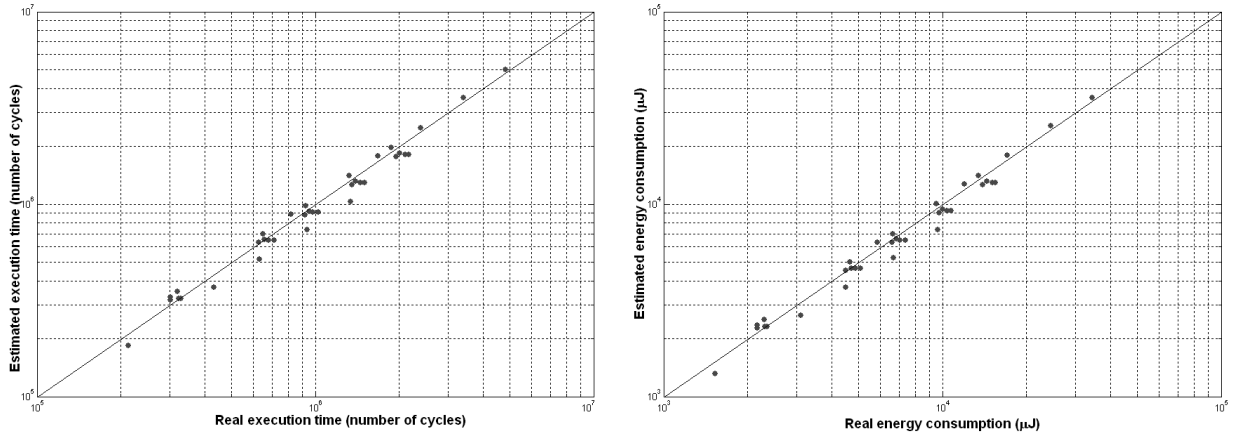


Figure 5. Actual and estimated execution time (clock cycles) and energy (μJ).

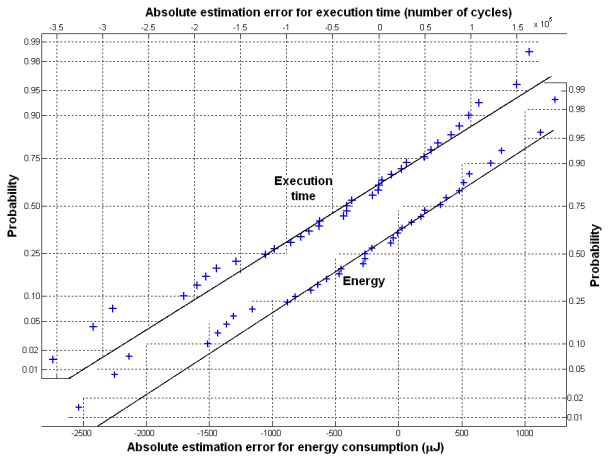


Figure 6. Close-to-normal distribution of the estimation error

Quality of estimation of the execution time

	c_t (real)	$\hat{c}_t$ (estim.)	$\hat{c}_t - c_t$ (deviation)	$\hat{c}_t - c_t$ % (dev. %)
Average value	1 244 360	1 195 393	48 967	-4.2024
Standard deviation	957 230	976 602	127 760	9.3006
Coefficient of correlation: $\rho(c_t, \hat{c}_t) = 0.99147$				

Quality of estimation of the energy

	c_e (real)	$\hat{c}_e$ (estim.)	$\hat{c}_e - c_e$ (deviation)	$\hat{c}_e - c_e$ % (dev. %)
Average value	8 913.78	8 552.46	-361.32	-4.3857
Standard deviation	6 844.37	6 993.89	926.76	9.3151
Coefficient of correlation: $\rho(c_e, \hat{c}_e) = 0.99126$				

Table 4. Accuracy result summary

Benchmark	Simulation time (s) IS level	source level	Speed-up	Full speed runtime	Profiling overhead
basicmath	2 175.0	0.05	43 500.0	0.04	1.25
crc32	60.2	0.02	3 010.0	0.01	2.00
dijkstra	292.5	0.24	1 218.8	0.01	24.00
fft	6 345.9	0.33	19 230.0	0.23	1.43
jpeg	183.2	0.14	1 308.6	0.06	2.33
sha	115.0	0.02	5 750.0	0.01	2.00
susan	142.8	0.02	7 140.0	0.01	2.00
Average			11 359.3		2.22

Table 5. Performance result summary

distribution function of the estimation error over a "normal probability plot" (see Figure 6), a set of axes where the horizontal is linear, and the vertical is pre-distorted in such a way that a gaussian cdf is a straight line. The dotted lines in the plot actually show the cdf of a gaussian distribution with same mean and variance as our error population. The plot clearly shows that the estimation error closely approximates a gaussian distribution.

8. Conclusions

This paper described a methodology to determine the cost of executing a C program, providing estimates for source-level entities. The method is formally founded and supported by a complete set of tools. It proved to be accurate and remarkably faster than instruction-set simulation. The proposed approach exhibits several advantages with respect to other existing approaches, the most remarkable being that it operates and provides cost estimates at the same level at which the application is described (i.e. at source-level), it does not require the target onto which the code will be targeted but rather relies on a pre-characterization

of that platform (processor, caches, memory and compiler) and it is much faster and comparably accurate to other less efficient techniques.

References

[1] D. Brooks, V. Tiwari, and M. Martonosi, Wattch: A Framework for Architectural Level Power Analysis and Optimizations, in *Proc. ISCA00*, 2000, pp.83-94;

[2] T. Simunic, L. Benini and G. De Micheli, Energy Efficient Design of Battery-Powered Embedded Systems in Special Issue of *IEEE Transactions on VLSI*, May 2001;

[3] M. Lajolo, M. Lazarescu, A. Sangiovanni-Vincentelli, A compilation-based Software Estimation Scheme for Hw/Sw co-simulation, in *Proc. CODES'99*, Roma, Italy, pp. 85-89;

[4] E. Senn, N. Julien, J. Laurent and E. Martin, Power Consumption Estimation of a C Program for Data-Intensive Applications, in *Proc. Workshop on Complexity-Effective Design*, Anchorage, Alaska, USA, May 2002;

[5] N. Julien, E. Senn, J. Laurent and E. Martin, Power Estimation of a C algorithm on a VLIW Processor, in *Proc. ISHPC-IV*, Kansai Science City, Japan, May 2002;

[6] M. Ravasi and M. Mattavelli, High-level algorithmic complexity evaluation for system design, in *Journal of Systems Architecture*, n. 48 (2003), pp. 403-427;

[7] A. Muttreja, A. Raghunathan, S. Ravi and N. K. Jha, Automated energy/performance macromodeling of embedded software, in *Proc. DAC*, June 2004, pp. 99-102;

[8] C. Brandolese et al., Library Functions Timing Characterization for Source–Level Analysis, in *Proc. DATE'03*, Munich, Germany, March 2003, pp. 1132-1133;

[9] J. Degener, ANSI C Yacc grammar, `www.lysator.liu.se/c/ANSI-C-grammar-y.html`;

[10] C. Brandolese, W. Fornaciari, F. Salice and D. Sciuto, Energy Estimation for 32-bit Microprocessors in *Proc. CODES 2000*, Mission Bay, San Diego, May 2000;

[11] C. Brandolese, W. Fornaciari, F. Salice and D. Sciuto, An Instruction-level Functionality-based Energy Estimation Model for 32-bit Microprocessors, in *Proc. Design Automation Conf.*, Los Angeles, CA, USA, June 2000;

[12] H. Schildt, American National Standard for Programming Languages-C : ANSI/ISO 9899-1990, *The Annotated ANSI C Standard*, Osborne McGraw-Hill, Berkeley, CA, USA, 1990;

[13] T. Ball and J. R. Larus, Optimally profiling and tracing programs, in *ACM Transactions on Programming Languages and Systems*, Vol. 16, July 1994, pp. 1319-1360;

[14] M. R. Guthaus et al., MiBench: A free, commercially representative embedded benchmark suite, in *Proc. IEEE 4th Annual Workshop on Workload Characterization*, Austin, TX, USA, Dec. 2001;

[15] The Free Software Foundation, Bison, `www.gnu.org/manual/bison-1.35/bison.html`

[16] The Free Software Foundation, Flex, `www.gnu.org/manual/flex-2.5.4/flex.html`

[17] J. Degener, ANSI C grammar, Lex specification, `www.lysator.liu.se/c/ANSI-C-grammar-l.html`;

[18] P. Yang et al., Managing Dynamic Concurrent Tasks in Embedded Real-Time Multimedia Systems, in *Proc. 15th Intl. Symposium on System Synthesis (ISSS 2002)*, Kyoto, Japan, Oct. 2002, pp. 112-119;

[19] E.-Y. Chung, L. Benini and G. De Micheli, Energy Efficient Source Code Transformation based on Value Profiling in *Proc. Intl. Workshop on Compilers and OSes for Low Power*, Oct. 2000;