Timing and Energy Estimation of C Programs
[SPECIAL ISSUE ON POWER AWARE EMBEDDED COMPUTING]

CARLO BRANDOLESE
WILLIAM FORNACIARI
FABIO SALICE
DONATELLA SCIUTO
Politecnico di Milano

This paper affords the problem of analyzing the timing and energetic aspects of software for embedded applications. The main goal of the approach is to enable design space exploration over different microprocessors, development environments and coding alternatives. The approach embodies the benefits of static and dynamic analysis within a formal mathematical framework and takes full advantage of the accuracy of low–level methodologies while operating at source code level. The experimental assessment of the methodology considered C programs derived from real–world applications and confirmed its accuracy and effectiveness.

Categories and Subject Descriptors: X.Y.Z [**TBD**]: TBD—*TBD*

General Terms: Power estimation, Embedded systems, Timing analysis

1. INTRODUCTION

Nowadays, the market trend is moving quickly toward a massive use of embedded systems in all sorts of application fields (automotive, domotics, mobile phones, devices for multimedia and Internet-browsing, etc.). Within such a wide range of environments, portable and/or real–time systems represent a significant segment of the associated market. As far as portable applications are concerned, battery life is a key factor and therefore energy consumption control is one of the goals the embedded systems' designer must pursuit. On the other hand, embedded systems for real-time, possibly mission–critical applications, poses severe constraints on the timing requirements whose violation, in many cases, could lead to catastrophic effects. The intrinsic conflict related to energy and time, along with the implementation costs, imposes an accurate design space exploration since the trend to use oversized components, to easily comply with fast response, is hard to fit with the need of *squeezing* the systems to cope with low energy requirements.

In this scenario, the fast growth of the complexity of the required functionalities,

Authors' address: C. Brandolese, W. Fornaciari, F. Salice and D. Sciuto, Politecnico di Milano, Piazza L. Da Vinci, 32 – 20133 Milano, Italy, Tel. +39 02 23954.269, Fax. +39 02 23954.254, {brandole,fornacia,salice,sciuto}@elet.polimi.it

the time–to–market contraction, and the product life–cycle shortening are factors that naturally lead to reuse standard components as much as possible. Furthermore, significant parts of the system implementation are shifted toward software since its intrinsic flexibility allows the designer to change even core sections of the product without hardware reorganization until the latest steps of the design and, in some cases, also after its release. As a result, a careful analysis of the design alternatives is desirable, in order to deal with the intrinsic conflict among flexibility, modularity, cost and performance. The implementation of an embedded system requires an investigation of a large variety of the possible synergies between the software and the hardware. In particular, different target processors and different software/hardware partitioning solutions should be analyzed. A concurrent design flow (co–design) is thus a valuable strategy to perform design space exploration. In such a flow, one of the early steps deals with the identification of a set of admissible hardware/software partitioning alternatives. Such solutions must fulfill the constraints and requirements of the design.

It is quite understandable that the hardware/software partitioning step requires a set of strategies, heuristics and metrics coping with the intrinsic complexity of the problem and with the wideness of the design space. Among the aids for the investigation, particular attention should be paid to metrics. In fact, the evaluation in terms of energy, time, cost, etc. of a point in the design space has to be as fast and accurate as possible in order to ensure an effective exploration of alternative realizations. Do note that, due to flexibility requirements and the considerable impact of the software component in terms of execution time and energy consumption, a correct estimation of these figures is crucial for the success of an embedded system.

In recent years, different approaches have been presented in literature for software analysis, either for energy or for execution time estimation, ranging from *gate–level* simulations to *high–level* static methods. Excluding circuit– and gate–level techniques [Mehta et al. 1996; Landman and Rabaey 1993], which are accurate but both very slow and strongly processor–dependent, other approaches attack the problem at different levels of abstraction and from different points of view. The goal of these investigations is to identify a tradeoff between fast estimation and accuracy. A possible, though simplistic, classification of such methodologies considers two orthogonal aspects: *estimation paradigm* (static vs. dynamic) and *abstraction level* (high–level vs. low–level).

Static approaches [Malik et al. 1997; Macii et al. 1998] analyze the specification without performing any simulation (at high level) or execution (at low level). These strategies produce a rapid, coarse–grained software characterization. Conversely, dynamic approaches [Giusto et al. 2001] simulate the specification taking into account the impact of the environment. Such solutions, though extremely time consuming, lead to finer–grained and more accurate results.

Low level estimation methods [Hsieh and Pedram 1998; Lazarescu et al. 2000; Macii et al. 1998] are based on the analysis (static or dynamic) of the compiled specification, with explicit reference to a set of models of the processor, the instruction–set and the memory architecture. Such estimation paradigms generally lead to very accurate results since at the level of abstraction they operate, accurate information is available. This, of course, is not in favor of estimation efficiency, and thus often prevents a significant exploration of the design space.

Though extremely valuable, to the best of our knowledge, all the approaches currently presented in literature (see [Benini and Micheli 2000] for a comprehensive survey) are strongly specific and targeted to either accuracy or efficiency. Moreover, most of them do not consider retargetability as a crucial issue.

The approach proposed in this paper tries to afford the problem of energy and timing estimation of the software components of an embedded system under a more general point of view. In particular, it combines the efficiency of high–level static estimation with the accuracy of low–level dynamic analysis within a rigorous conceptual framework based on well–defined and flexible mathematical models.

2. PROBLEM FORMULATION

The goal of the proposed approach is to achieve efficiency, retargetability and accuracy in energy and execution time estimation of software programs, with particular emphasis to embedded applications. To this purpose, the adopted strategy consists in a sharp decoupling of static and dynamic analyses to enhance the generality and enforce efficiency. The static portion of the flow, in addition, has been decomposed into a high–level, architecture–independent and a low–level, technology dependent analyses. The complete flow is sketched in Figure 1.

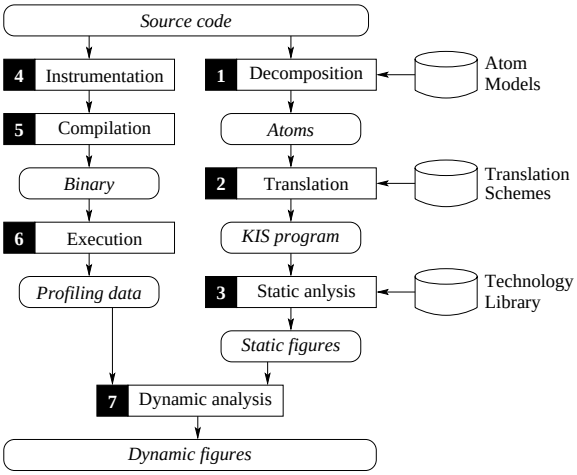


Fig. 1. Source code estimation flow

An efficient analysis framework must necessarily operate on source code, since, at this level of abstraction, the specification allows capturing coarser–grained properties at the same level at which a designer typically conceives the application. Nevertheless, the elementary operations of a high–level language are far too complex to be analyzed as a whole. Step (1) in the flow depicted in Figure 1 is devoted to decompose the application specification into elementary constructs, henceforth referred to as *atoms*. Section 3.1 introduces the idea of atom. While the semantic of atoms is platform–independent, their implementation strongly varies from one architecture to another, depending on the details of different assembly instruction–sets. Step (2) of the analysis flow is devoted to fill this gap between high–level,

complex statements and low–level, simple assembly instructions. This is achieved by representing source language atoms by means of pseudo–assembly (*Kernel Instruction Set*) instruction sequences, forming a *KIS program*. The detailed description of the *KIS* and of the formal procedure used to derive it are treated in Section 3.2, while the most significant translation templates are reported in Section 6.

At this phase of the flow, the entire specification is represented by means of simple operations, whose energy and timing properties can be more easily modeled. It is worth noting that the platform–independence is still maintained at this level since both *Atom Models* and *Translation Templates* (see, again, Figure 1) have been designed with this purpose. Translation templates, in particular, refer to an ideal architecture/compiler pair whose characteristics are clarified at the beginning of Section 5. The low–level, technology–independent model is processed, at step (3) of Figure 1, by associating a specific cost to each *KIS* instruction based on a set of processor–specific *Technology libraries* collecting the timing and energy figures of specific target architectures. Some approaches to low–level power modeling are presented in [Russell and Jacome 1998; Tiwari et al. 1996; C.Brandolese et al. 2000; 2001]. It is worth noting that different target alternatives can be easily explored at this point, without re–analyzing and re–translating the original source code, but rather by simply choosing a different technology library.

All the steps described thus far lead to static figures. It is clear, however, that sensible design choices can only be made on the basis of a dynamic characterization, which is thus mandatory. Operating at the source code level implies two other significant advantages related to retargetability. On one hand, the nature of all high–level languages makes them independent from the target architecture and, on the other hand, the source code can be easily profiled on a generic host machine without loss of generality. To this purpose, the original source code is suitably instrumented and compiled on a generic host machine (steps (4) and (5)). The *binary* program obtained is executed in step (6) to produce source–level profiling data. This phase can be repeated for all input data sets the designer is interested in. Finally step (7) combines the static timing and energy figures with profiling data and produces the dynamic estimates. A rigorous mathematical model, presented in Sections 4 and 5, constitutes the statistical basis of the whole methodology.

In conclusion, the proposed flow constructs a technology– and data–independent model of the source program that can be efficiently *specialized* for different target architectures and environmental stimuli. This allows the designer to obtain accurate results by simply selecting the target architecture and feeding the model with the desired input data. To better clarify all the estimation steps just described, Section 7 presents a detailed description of a case–study, along with the results obtained on some benchmarks. Some concluding remarks are finally drawn in Section 8.

3. LANGUAGE DECOMPOSITION

3.1 Source language decomposition

To model a high–level language in a constructive and hierarchical way it is necessary to define what an "elementary component" is. In the following, elementary components will be referred to as *atoms*. The most critical point in defining the

atoms of a language is the choice of the granularity at which the language should be analyzed. To this purpose it is valuable to describe the language (C, in this case) using the formalism of grammars. The starting point is the definition of a set of *terminal symbols*: these symbols are the basic building blocks of the atoms of the language and, as a consequence, determine the granularity of the analysis. It is important to note that some of the terminal symbols adopted in this context would be non–terminals in the complete grammar of the language and would be not disjoint but rather related by a production. The essential terminals are, as in usual grammars, operators, keywords and special symbols such as `=`, `(`, `)`, `{`, `}`, `for`, `return` and others. In addition, to the purpose of the definition of atoms, it is useful introducing new terminals for variables, and expressions. Using this set of terminals a few productions can already be built. Consider, as an example the productions of Figure 2.

st-break	→	`break ;`
st-return	→	`return` *expr* `;`
st-assign	→	*var* `=` *expr* `;` | *var* `[` *expr* `]` `=` *expr* `;`

Fig. 2. Productions for some simple statements

The partial grammar of Figure 2 clarifies how the terminals combines to give more complex portions of the language. In a similar manner, it is possible to define more complex statements, such as `while` or `if`. An excerpt of their grammar is reported in Figure 3.

st-while	→	`while (` *expr* `)` *stmt*
st-if	→	`if (` *expr* `)` *stmt* | `if (` *expr* `)` *stmt* `else` *stmt*

Fig. 3. Productions for two complex statements

In these productions the non–terminal symbol *stmt* represents a generic statement defined by the partial grammar of Figure 4.

stmt	→	*simple-stmt* | *block-stmt*
simple-stmt	→	*st-assign* | *st-break* | ...
block-stmt	→	`{` *stmt-list* `}`
stmt-list	→	*stmt* | *stmt-list stmt*

Fig. 4. Definition of the generic statement

Similarly, it is possible to define all the statements and constructs of the C language. The outcome is an incomplete grammar Γ that, on one hand, lacks the productions for all the symbols that have been assumed as terminals while, on the other hand, allows defining in a formal way the concept of atom at the desired level of granularity.

DEFINITION 1. *The terminal symbols on the right–hand side of a production of the grammar* Γ *constitute an atom whose name is defined in the left–hand side of the production.*

According to this definition and referring, as an example, to the production for the `while` statement, the atom whose name is *while* is constituted by the keyword `while` and the couple of parentheses enclosing the conditional expression *expr*. The condition and the body of the `while` construct, on the other hand, are not part of the *while* atom itself.

3.2 Assembly language decomposition

The source code of a program can thus be seen as a set of atoms. Each atom expresses a high–level operation that will be eventually implemented as a suitable sequence of assembly instructions, according to either a simple translation template or a more complex compilation algorithm. As an example, let us consider two different atoms: a loop construct such as `while` or `for`, and an arithmetic expression. The former can be translated according to a template scheme since its syntax is fixed. The syntax of the expression, though clearly defined, allows it to grow arbitrarily complex and thus no templates can be envisioned but rather each expression must be processed according to tree–visiting algorithms.

The translation of a source code can thus be seen as a sequence of assembly instructions each composed by an *operation code* and a certain number of *operands* of different types. The operation code is strictly related to the tasks that must be performed to implement the desired high–level functionality and thus is fixed. The number of operands supported by the assembly language is a characteristic of the instruction–set and is thus also fixed. Due to the reasons discussed above, the number of operands of the instruction–set is fixed once the target processor has been selected and predictions on which operation codes will be used can be made based on the knowledge of the most popular compilation techniques and translation templates.

Original instruction
`ADD [R0,R4], [R2+], R3`

(a)

Transformed code
`ADD R0, R4, R5`
`LOAD [R5], R6`
`LOAD [R2], R7`
`ADD R2, #1, R2`
`ADD R6, R7, R3`

(b)

Fig. 5. Sample decomposition of complex addressing modes

The addressing modes of the operands of the assembly instructions, on the other hand, are extremely hard to predict: they depend, in fact, on a number of factors such as the class of the target processor (RISC, CISC, VLIW, etc.), its architecture (number of general purpose registers, data path complexity, etc.) and the compiler (optimization techniques, etc.). To cope with this problem the following strategy has been adopted. A generic instruction using complex addressing modes can always be decomposed in a suitable sequence of instructions using only simple addressing

modes such as immediate register direct or register indirect. For example consider the instruction shown in Figure 5(a): the first operand uses the indexed addressing mode, the second uses the register indirect with auto–increment addressing mode and the third is a register direct; this instruction can be replaced by the code of Figure 5(b) exploiting simple addressing modes only.

It is reasonable to assume that a processor providing complex addressing modes has a number of units specifically dedicated to their management. These units are optimized and thus, probably, their use requires a shorter time than the execution of the corresponding sequence of instructions exploiting simple addressing modes only. This assumption is supported by the timing figures experimentally obtained on many microprocessors: the execution time of the expanded instruction is always an overestimate of the actual one. Based on this evidence, it is possible to translate an arbitrary source code into an assembly program using simple addressing mode instructions only, always resulting in a solution overestimating the actual execution time. A significant achievement of using a limited subset of instructions is the generalization capability since the basic operations that can be executed are roughly the same over a wide range of general–purpose processors. This concept can be formalized as follows.

Let P_h be a generic microprocessor and IS_h its instruction–set. Let then $\mathcal{P} = \{P_1, P_2, \ldots, P_p\}$ be a set of $p = |\mathcal{P}|$ processors supporting instructions with the same maximum number of operands (typically one, two or three). The generic instruction set IS_h can be partitioned into a fixed number c of predefined *Instruction Classes* $IC_{h,k}$ performing similar operations, such as data transfer, load/store, branch, etc. The instruction classes must satisfy the following relations:

$$\begin{cases} IS_h = \bigcup_{k=1}^{c} IC_{h,k} \\ IC_{h,k_1} \cap IC_{h,k_2} = \emptyset \qquad \forall k_1 \neq k_2 \in [1; c] \end{cases} \tag{1}$$

Instruction sets of different processors may significantly differ: for this reason a specific processor may have one or more empty instruction classes. Two instructions $I_1 \in IS_1$ and $I_2 \in IS_2$ belonging to different instruction sets are said to be *compatible* if and only if:

$$\exists k \mid (I_1 \in IC_{1,k}) \wedge (I_2 \in IC_{2,k}) \tag{2}$$

Considering all the p processors in $\mathcal{P}$ and their instruction–sets IS_h, $\bar{c}$ compatible instruction classes can be defined according to the following relation:

$$CIC_k = \begin{cases} \emptyset & \exists h \mid IC_{h,k} = \emptyset \\ \bigcup_{h=1}^{p} IC_{h,k} & \text{otherwise} \end{cases} \tag{3}$$

These new instruction classes collect all the instructions of different processors that are compatible in the sense that all the instructions in the same class perform equivalent operations. The union of all CIC_k classes can be thought of as a generic instruction set denoted as KIS or *Kernel Instruction Set*.

$$KIS = \bigcup_{k=1,\, CIC_k \neq \emptyset}^{\bar{c}} CIC_k \tag{4}$$

Let $CIC_k = \{I_{k,1}, \ldots, I_{k,N_k}\}$ be the k-th compatible instruction class and N_k its cardinality. An instructions $I_{k,L}$ can be identified in each compatible instruction class CIC_k such that its execution time, expressed in terms of CPI (Clock–cycles Per Instruction) is minimum. The instruction $I_{k,L}$ is the lower execution time bound for the k-th instruction class. Consider now a generic instruction I executed in $cpi(I)$ clock cycles. If I belongs to the k-th compatible instruction class, i.e. $I \in CIC_k$ then a lower–bound on its execution time is $cpi(I_{k,L})$. If I belongs to none of the compatible instruction classes, then there exist no single instruction in the compatible instruction set that can perform the same operation. Its functionality must thus be obtained by combining more than one instruction in KIS, as clarified by the example in Figure 5. The lower bound for execution time of instruction $I \in KIS$ can thus be defined introducing the function:

$$cpi_{min}(I) = cpi(I_{L,k}) \mid I \in CIC_k \tag{5}$$

By using the instructions in KIS, it is thus possible to generalize the translation templates over multiple instruction sets and to account for the behavior of different compilers.

3.3 Assembly language reference model

This scheme, though, still does not account for the architectural differences between microprocessors. It is well known, in fact, that the result of the compilation strongly depends on the available number of general–purpose registers. To properly account for such differences, an *ideal* architecture with an infinite number of general–purpose registers has been assumed as reference. This assumption greatly simplifies the construction of translation models, improves generality and allows an accurate estimate of timing and energy figures. A detailed semantic and experimental analysis of a wide range of instruction sets has led to the definition of the compatible instruction classes listed in Table I, along with a brief description of the semantics of the instructions they represent. The name of the class can be also

Class	Semantics
alul	Light ALU operations
aluh	Heavy ALU operations
mvld	Load from 'memory'
jump	Branches

Class	Semantics
cmpl	Light compare operations
cmph	Heavy compare operations
mvst	Store to 'memory'
call	Subroutine call/return

Table I. Compatible instruction classes composing the KIS

assumed to be the representative instruction of the class itself. For this reason, and for the sake of conciseness, the term *instructions*, such as *alul* or *jump*, will be used as a shorthand for the set of all actual instructions belonging to the corresponding compatible instruction class. The two classes *mvld* and *mvst* are worth noting since they refer to memory access. In the ideal reference architecture the memory is non present since all data is supposed to reside in registers. Nevertheless, the C language provides indirect memory access by means of pointers, i.e. variables whose value is the memory address of another variable. This concept has been captured in the kernel instruction set by assuming that a register can contain the number of a different register and the two instructions *mvld* and *mvst* provide indirect register

access. As an example, suppose that register `R3` contains the value 5 and register `R5` contains the value 10. In this case the instruction `mvld R3,R0` will copy the value 10 into the destination register `R0`.

Based on the kernel instruction set just defined, it is possible to identify either a translation template or a translation algorithm that transforms an atom into the sequence of kernel instructions necessary to its implement.

4. NOTATIONS

4.1 Code notations

The symbols summarized in Table II and described in the following are used to formally represent the source code and the set of data it operates on. According to the grammar Γ and to definition (1), a generic source code can be represented as a list of couples describing the atoms and the hierarchical relations between them.

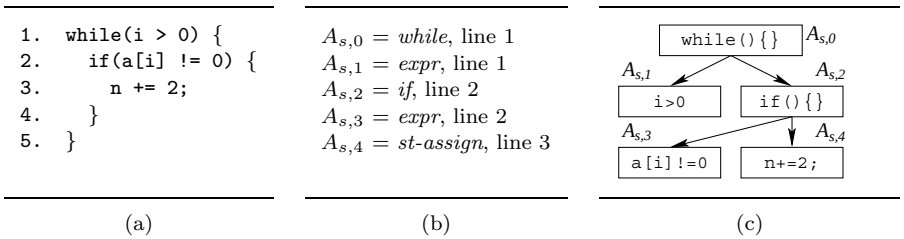
Notation	Meaning
s	source code index
i	atom index
j	data set index
m	kernel instruction index
C_s	s-th source code
$A_{s,i}$	i-th atom of the s-th source code
$I_{s,i,m}$	m-th kernel instruction of atom $A_{s,i}$
L_s	number of atoms of the s-th source code
$M_{s,i}$	number of kernel instructions for atom $A_{s,i}$
$D_{s,j}$	j-th data set for the s-th source code

Table II. Notation for source code

The generic source code C_s is a tree of atoms and can be completely defined by an ordered list of $L_s = |C_s|$ couples $(A_{s,i}; \overline{A}_{s,i})$ where $A_{s,i}$ is the actual atom while $\overline{A}_{s,i}$ is a reference to the parent atom of $A_{s,i}$ in the tree. A generic source code is thus formally represented as:

$$C_s = \{(A_{s,1}; \overline{A}_{s,1}),\ (A_{s,2}; \overline{A}_{s,2}),\ \ldots,\ (A_{s,L_s}; \overline{A}_{s,L_s})\} \tag{6}$$

For the sake of clarity, let us consider a simple example. Figure 6(a) shows a brief portion of C code and the identified atoms are listed in Figure 6(b). Finally Figure 6(c) shows a graphical representation of the corresponding tree.



(a) (b) (c)

Fig. 6. Source code, atoms and syntax tree

The list capturing the structure of this portion of code is thus:

$$L_s = \{(A_{s,0}, root),\ (A_{s,1}, A_{s,0}),\ (A_{s,2}, A_{s,0}),\ (A_{s,3}, A_{s,2}),\ (A_{s,4}, A_{s,2})\} \quad (7)$$

where the first couple indicates that the atom $A_{s,0}$ is the root atom.

Atoms, in turn, are represented as a list of kernel instructions, according to some set of suitable *translation templates*. An atom $A_{s,i}$ is thus modeled as as:

$$A_{s,i} \Leftrightarrow \{I_{s,i,1},\ I_{s,i,2},\ ,\ldots,\ I_{s,i,M_{s,i}},\ \} \qquad \text{with} \quad I_{s,i,m} \in KIS \quad (8)$$

Finally, the symbol $D_{s,j}$, far from giving details on the type, size and value of data, is used to refer to a specific run of the source code. Data is made dependent both on the spanning index j and the source code index s since a set of data must be compatible with the specific source code.

4.2 Profiling notation

When a source code C_s is run with data $D_{s,j}$ each atom, observed as a whole at source level, is executed a certain number of times. The symbols summarized in Table III are used to formally express profiling information.

Notation	Meaning
$n()$	function returning the number of executions
$N_{s,i,j}$	number of executions of the atom $A_{s,i}$ with data $D_{s,j}$
$N_{s,j}$	number of execution of all atoms of C_s with data $D_{s,j}$

Table III. Notation for source–level profiling

Since the function $n()$ returns a dynamic measure, it must always depend on a set of data. According to the above definitions, the following relations hold:

$$N_{s,i,j} = n(A_{s,i}, D_{s,j}) \quad (9)$$

$$N_{s,j} = n(C_s, D_{s,j}) = \sum_{i=1}^{L_s} n(A_{s,i}, D_{s,j}) = \sum_{i=1}^{L_s} N_{s,i,j} \quad (10)$$

The count returned by the function $n()$ has no explicit relation with the actual execution time or with the number of clock cycles.

4.3 Timing notation

In order to account for a real measure of the execution time, the functions and symbols of Table IV have been introduced.

Notation	Meaning
$t()$	returns the actual execution time of its argument
$\overline{t}()$	returns the reference execution time of its argument
$\hat{t}()$	returns the estimated execution time of its argument
$T_{s,j}$	actual execution time of source code C_s with data $D_{s,j}$
$\overline{T}_{s,j}$	reference execution time of source code C_s with data $D_{s,j}$
$\hat{T}_{s,j}$	estimated execution time of source code C_s with data $D_{s,j}$

Table IV. Notation for timing

The measure of unit of time, in this context, is expressed in terms of CPI in order to be independent of the operating frequency of a microprocessor. In the construction of the mathematical form the upper–case, shorthand forms will be used more often. Their meaning is clarified by the three following simple relations:

$$T_{s,j} = t(C_s, D_{s,j}) \tag{11}$$

$$\overline{T}_{s,j} = \overline{t}(C_s, D_{s,j}) \tag{12}$$

$$\hat{T}_{s,j} = \hat{t}(C_s, D_{s,j}) \tag{13}$$

The total actual execution time $T_{s,j}$ of the source code C_s with data $D_{s,j}$ can be expressed as the sum of the execution time of each atom $A_{s,i}$, counted $N_{s,i,j}$ times, that is:

$$T_{s,j} = \sum_{i=1}^{L_s} t(A_{s,i}, D_{s,j}) \cdot n(A_{s,i}, D_{s,j}) = \sum_{i=1}^{L_s} t(A_{s,i}, D_{s,j}) \cdot N_{s,i,j} \tag{14}$$

The reference and estimated timing can also be expressed with similar relations, but their discussion is deferred until the next Section.

5. FORMAL MODELS

5.1 Execution time general model

In this paragraph, a general mathematical framework suitable to describe the execution time of a software, starting from its high–level source description is presented.

As indicated in the concluding part of the previous Section, the execution time of a source code C_s with input data $D_{s,j}$ can be expressed as:

$$T_{s,j} = \sum_{i=1}^{L_s} N_{s,i,j} \cdot t(A_{s,i}, D_{s,j}) \tag{15}$$

where the dependence of the real execution time of an atom is explicitly indicated by the second argument of the function $t()$.

Equation (14) explicitly shows the dependence of the execution time on the specific input data and defines what is called *actual time* throughout the rest of the paper. To afford the complexity of the problem it is useful to start by considering ideal conditions delineated by the following assumptions:

—the target architecture has an unlimited number registers and all the variables of the code are allocated to a fixed register;

—the initial value of the variables is pre–loaded in the corresponding register and thus no explicit initialization is required;

—the execution time of a KIS instruction is approximated with its lower bound.

—inter–atom compiler optimizations are neglected;

—intra–atom compiler optimizations are neglected;

The first three items of this list lead to an underestimate of the real execution time while the last two tend to produce an overestimate. Let $\overline{t}()$ denote the function returning the execution time in these ideal conditions, referred to, in the following,

as *reference time*. Noting that the reference execution time of a kernel assembly instruction $\bar{t}(I_{s,i,m})$ is equal to $cpi_{min}(I_{s,i,m})$, the following equation holds:

$$\overline{T}_{s,j} = \sum_{i=1}^{L_s} \left[N_{s,i,j} \cdot \sum_{m=1}^{M_{s,i}} cpi_{min}(I_{s,i,m}) \right] \tag{16}$$

The aim of the model is to determine a function $\hat{t}()$ returning an estimate of the actual execution time such that the estimation error is minimized. The relation that express the estimated timing of a source code is similar to equation (14) for the actual timing:

$$\hat{T}_{s,j} = \sum_{i=1}^{L_s} N_{s,i,j} \cdot \hat{t}(A_{s,i}, C_s) \tag{17}$$

It is worth noting that the function $\hat{t}$ does not depend on the actual data fed as input to the source code: this dependence is completely accounted for in the execution count coefficient $N_{s,i,j}$. This is crucial in order to allow an a–priori characterization of the atoms. Nevertheless, an explicit dependence on the source code C_s, considered as a whole, is present: this point is clarified later on. The error to minimize over a number of source codes and input data sets is thus:

$$\epsilon^2_{s,j} = \left(T_{s,j} - \hat{T}_{s,j} \right)^2 \tag{18}$$

The basic idea behind the model is that the estimated timing of each atom can be expressed as the sum of two contribution: the reference timing, that accounts for all the *deterministic* aspects in ideal conditions, and a statistical deviation that depends on complex factors such as the structure of the source code, the characteristics of the compiler, the actual architecture etc. This idea is be formally expressed by the following relation:

$$\hat{t}(A_{s,i}, C_s) = \overline{t}(A_{s,j}) + \delta(C_s, A_{s,i}) \tag{19}$$

where the form of the function $\delta()$ can be arbitrarily defined. An analysis of some preliminary timing measurement results, suggests that δ should depend on the specific atom $A_{s,i}$ as well as on the source code considered as a whole. For the sake of generality, it might be useful and interesting to consider a dependence not only on the atom $A_{s,j}$ but rather on a range of adjacent atoms, thus:

$$\delta = \delta(A_{s,i-k_1}, \ldots, A_{s,i}, \ldots A_{s,i+k_2}, C_s) \tag{20}$$

where k_1 and k_2 determine the extension of the range around $A_{s,i}$. The dependence on the source code and the range of atoms defined thus far is only symbolic, since atoms and source code are neither numbers nor functions, and does not specify the explicit mathematical form. To refine the expression of the function it is necessary to identify some measures to be performed statically on an arbitrary range of atoms and on the entire source code. As examples consider measurements such as the number of consecutive sequential statements, the maximum nesting level of the whole source, the number of variable used and so on. For the sake of conciseness, let $\mathbf{q}()$ and $\mathbf{Q}()$ be two vector functions operating on a range of atoms and on the

tree representing the entire source code, respectively. Formally:

$$\mathbf{q} = \mathbf{q}(A_{s,i-k_1}, \ldots, A_{s,i}, \ldots A_{s,i+k_2}) \tag{21}$$

$$\mathbf{Q} = \mathbf{Q}(C_s) \tag{22}$$

According to the definitions and hypotheses discussed thus far, and combining equations (17), (19), (20), (21) and (22) the estimated time can be expressed as:

$$\hat{T}_{s,j} = \sum_{i=1}^{L_s} N_{s,i,j} \cdot \left[\overline{t}(A_{s,i}) + \delta(\mathbf{q}(A_{s,i-k_1}, \ldots, A_{s,i+k_2}), \mathbf{Q}(C_s))\right] \tag{23}$$

Equation 23 expresses a very general and flexible model but involves a number of scalar, vectorial and functional unknowns that render it almost mathematically untreatable. The next paragraph proposes some simplifications that, though limiting the generality, lead to an affordable mathematical problem.

5.2 Execution time simplified model

At this point it is useful to summarize how the different components of the model can be determined.

what, in the model, is to be considered known and what is unknown. The following components are known:

(1) The reference time of each atom $\overline{t}(A_{s,j})$. The value can be derived by using the analytical models of the atoms, described in Section 6, combined with the timings of KIS instructions.

(2) The execution count of atoms $N_{s,i,j}$. These values can be derived from source–level profiling of the code. A possible solution to this problem has been implemented by instrumenting, compiling and running the original source code on a generic host platform.

(3) The mathematical form of the statistical correction function $\delta(\cdot)$ with respect to the vector functions $\mathbf{q}$ and $\mathbf{Q}$. A possible formulation is proposed in this Section.

(4) The meaning of the vector functions $\mathbf{q}$ and $\mathbf{Q}$. They should be applied to lists of atoms, and return numeric values corresponding to some sort of *measure* on the source code. Though it is possible to suggest a number of such measurement functions, it is all but straightforward to determine whether the adopted functions are meaningful for the problem or not. In the present work, both functions have been assumed constant.

A simple yet versatile form for the function $\delta(\cdot)$ is that of a multilinear dependence on the two vector functions: this leads to a significant mathematical simplification with an acceptable loss of generality. This assumption leads to the expression:

$$\delta = \mathbf{a} \times \mathbf{q}(A_{s,i-k_1}, \ldots, A_{s,i+k_2}) + \mathbf{b} \times \mathbf{Q}(C_s) + c \tag{24}$$

that, expanding the vectors and denoting with n_q and n_Q the number of elements of the two vectors $\mathbf{q}$ and $\mathbf{Q}$, respectively, becomes:

$$\delta = [\, a_1 \ \cdots \ a_{n_q} \,] \times \begin{bmatrix} q_1 \\ \vdots \\ q_{n_q} \end{bmatrix} + [\, b_1 \ \cdots \ b_{n_q} \,] \times \begin{bmatrix} Q_1 \\ \vdots \\ Q_{n_q} \end{bmatrix} + c \tag{25}$$

In this equation $a_1, \ldots, b_1, \ldots$ and c are the parameters of the model and their values have to be determined statistically in order to minimize the square error, while $q_1, \ldots$ and $Q_1, \ldots$ are the results of the measures performed on ranges of atoms and the entire source code, respectively.

5.3 Statistical model characterization

Consider now a source code C_s and a corresponding set of data $D_{s,j}$. According to equations 24 and 23 and noting that the reference times $\overline{t}(A_{s,i})$ and the vectors $\mathbf{q}$ and $\mathbf{Q}$ are known, the estimated time can be expressed as:

$$\hat{T}_{s,j} = \sum_{i=1}^{L_s} N_{s,i,j} \cdot \left[\overline{t}(A_{s,i}) + \mathbf{a} \times \mathbf{q} + \mathbf{b} \times \mathbf{Q} + c\right] \tag{26}$$

Distributing the summation an noting that $\mathbf{a}$ and $\mathbf{b}$ are intended to be independent of the atom, this relation can be rewritten as:

$$\hat{T}_{s,j} = \overline{T}_{s,j} + \mathbf{a} \times \mathbf{q}_{tot,j} + \mathbf{b} \times \mathbf{Q}_{tot,j} + c \cdot N_{s,j} \tag{27}$$

where:

$$\overline{T}_{s,j} = \sum_{i=1}^{L_s} N_{s,i,j} \cdot \overline{t}(A_{s,i}) \tag{28}$$

$$\mathbf{q}_{tot,j} = \sum_{i=1}^{L_s} N_{s,i,j} \cdot \mathbf{q} \tag{29}$$

$$\mathbf{Q}_{tot,j} = \sum_{i=1}^{L_s} N_{s,i,j} \cdot \mathbf{Q} \tag{30}$$

$$N_{s,j} = \sum_{i=1}^{L_s} N_{s,i,j} \tag{31}$$

Let now C_s be fixed and let the data set $D_{s,j}$ vary with the index $j = [1, \ldots, n_d]$. For each data set, the actual timing is given by $T_{s,j}$ while the estimated timing can be derived by using the above relations. With these data it is possible to build the linear system:

$$\begin{bmatrix} T_{s,1} \\ \vdots \\ T_{s,n_d} \end{bmatrix} = \begin{bmatrix} \overline{T}_{s,1} \\ \vdots \\ \overline{T}_{s,n_d} \end{bmatrix} + \begin{bmatrix} \mathbf{q}_{tot,1}^T & \mathbf{Q}_{tot,1}^T & N_{s,1} \\ \vdots & \vdots & \vdots \\ \mathbf{q}_{tot,n_d}^T & \mathbf{Q}_{tot,n_d}^T & N_{s,n_d} \end{bmatrix} \times \begin{bmatrix} \mathbf{a}^T \\ \mathbf{b}^T \\ c \end{bmatrix} \tag{32}$$

or, in a more compact form:

$$\mathbf{T}_s = \overline{\mathbf{T}}_s + \mathbf{A} \times \mathbf{X} \tag{33}$$

which becomes the well known linear form:

$$\mathbf{Y} = \mathbf{A} \times \mathbf{Y} \tag{34}$$

where the vector $\mathbf{Y}$ has been defined as:

$$\mathbf{Y} = \mathbf{T}_s - \overline{\mathbf{T}}_s \tag{35}$$

In equations (34) and (35), $\mathbf{Y}$ is a $n_d \times 1$ column vector, $\mathbf{A}$ is a $n_d \times (n_q + n_Q + 1)$ matrix and $\mathbf{X}$ is a $(n_q + n_Q + 1) \times 1$ column vector. Since the matrix $\mathbf{A}$ is not square and the experimental setup is such that $n_d \gg (n_q + n_Q + 1)$, the linear system can only be solved in a statistical sense. A simple and well known statistical estimator is the least square method that leads to the solution:

$$\mathbf{X} = \left(\mathbf{A}^T \times \mathbf{A}\right)^{-1} \times \mathbf{A}^T \times \mathbf{Y} \tag{36}$$

allowing to derive the parameters of the simplified linear model for any number of metric functions $\mathbf{q}$ and $\mathbf{Q}$.

5.4 Energy consumption model

The models presented thus far allow estimating the execution time of a program by analyzing and decomposing its source code into kernel assembly instructions. Such approach is viable thanks to the possibility of characterizing each KIS instruction with a reference execution time. To obtain an estimate of the energy absorbed by the execution of a program with a certain set of input data, each KIS instruction must also be associated with a *reference energy* absorption. To maintain the desired level of abstraction with respect to the target processor, energy characterization must be parametric.

Different approaches for the energy characterization of processors at assembly–level have been proposed in literature. The models in [Tiwari et al. 1996; Macii et al. 1998; Russell and Jacome 1998] are particularly interesting and can be easily adopted in this context. All these methodologies, though, do not provide sufficient generalization capabilities, resulting thus in contrast with one of the main goals of this paper.

The approach proposed in [C.Brandolese et al. 2000], on the other hand, abstracts from the architectural level by determining a set of *functionalities* and by decomposing the computational activity of each instruction in terms of these functionalities. The model shows good generalization properties and provides a *static* estimation of the energy consumption of single instructions. According to [C.Brandolese et al. 2000], the energy dissipation e_m of an instruction $I_{s,i,m}$ is given by as:

$$e_m = \sum_{j=0}^{5} e_{m,j} = V_{dd} \cdot \tau \cdot \sum_{j=0}^{5} if_j \cdot a_{m,j} \tag{37}$$

where if_j is the average current associated with the j-th functionality, V_{dd} is the power supply voltage, τ is the clock period and $a_{m,j}$ is a coefficient expressing the execution time spent by instruction the $I_{s,i,m}$ in the j-th functionality. It is worth noting that the energy contributions if_j are the parameters that make the model generic with respect to the target architecture. The coefficients $a_{m,j}$, on the other hand, are used to model the functional and timing behavior of each KIS instruction and satisfy the following relation:

$$\sum_{j=0}^{5} a_{m,j} = \bar{t}(I_{s,i,m}) \tag{38}$$

According to this model the energy absorbed by each instruction is computed as the weighted sum of the contributions related to the different functionalities.

6. ANALYSIS OF THE C LANGUAGE

6.1 Data types and variable access

Data types and variables are a critical aspect of the C language. Defining a clear and consistent model for the cost of variable access is thus of paramount importance. To this purpose, it is necessary to descend in further detail into the adopted memory model. The C language provides four classes of data types: scalars, pointers, arrays and structures (or unions). For each of these types a storage model has been defined paying particular attention to the mutual consistence.

6.1.1 *Scalars.* A scalar variable is stored in a single register, either integer or floating–point. In this context, the types `char`, `int` and all `short`, `long`, `long long`, `signed` and `unsigned` variants are considered integer while the types `float` and `double` and their variants are considered floating–point. The access cost is thus null since ALU instructions can access them directly.

6.1.2 *Pointers.* A pointer is stored in a single register and there is no difference between pointer to different data types. Accessing a pointer itself has thus a null cost while dereferentiation (operator `*`) is achieved by means of *mvld* or *mvst* instructions. Address extraction (operator `&`) involves copying the number of a register into a different register and this is performed by an *alul* instruction.

6.1.3 *Arrays.* Array elements are organized in a bank of consecutive registers. An additional register points to the base of the bank. This model closely resembles the memory model actually adopted by the C language. Accessing an element involves, thus, the computation of the target register, base plus (*alul*) offset, and an indirection operation. It is worth noting that the array access model and the pointer access model must be consistent. The translations of two semantically equivalent atoms such as `a[5]` and `*(a+5)` are, in fact, identical: both involve *alul* and *mvld* or *mvst* instructions.

6.1.4 *Structures and unions.* Similarly to arrays, structures are stored in bank of adjacent registers plus a register that serves as a pointer to the base of the bank. This organization is mainly motivated by an accurate analysis of the actual assembly code generated by different compilers for different architectures. The dot (`.`) member access operator has thus the same translation of an array access, while the indirect member access operator (`->`) requires an additional *mvld* instruction to dereference the pointer to the structure. This scheme is consistent with respect to the semantic equivalence of atoms such as `s->x` and `(*s).x`.

Unions are treated differently. A union, in fact, contains a single datum whose dimension depends on the field that is accessed. All fields of the union are stored in the same, shared, bank of registers whose base is fixed and known at compile–time. A member of a union, accessed by means of the dot operator, is thus a simple variable that can be used directly by all instructions. For the same reason, the indirect member access operator is translated according to the template adopted for simple pointers.

6.1.5 *Data types and variables cost models.* The translation templates described above lead to the models summarized in Table V. The costs are reported for the two cases in which the variable is either read (use) or written (definition).

Data type	Operator	Cost	
		Use	**Definition**
Scalar	n/a	0	0
Pointer	*	$\bar{t}(mvld)$	$\bar{t}(mvst)$
	&	$\bar{t}(alul)$	$\bar{t}(alul)$
Array	[]	$\bar{t}(alul) + \bar{t}(mvld)$	$\bar{t}(alul) + \bar{t}(mvst)$
Structure	.	$\bar{t}(alul) + \bar{t}(mvld)$	$\bar{t}(alul) + \bar{t}(mvst)$
	->	$\bar{t}(alul) + 2 \cdot \bar{t}(mvld)$	$\bar{t}(alul) + \bar{t}(mvld) + \bar{t}(mvst)$
Union	.	0	0
	->	$\bar{t}(mvld)$	$\bar{t}(mvst)$

Table V. Data types and variable access costs

6.2 Expressions

Expressions constitute the most important and complex portion of almost all programming languages. Their structure is hierarchical and involves many different types of operators and variables. To dominate this complexity, a taxonomy of the operators has been performed, resulting in three classes: *arithmetic operators* (+, -, /, bitwise operations etc.), *relational operators* (>, <, ==, etc.) and *logical operators* (!, &&, ||). According to these classes, three subtypes of expressions, referred to as *pure expression*, can be introduced, each containing only operators of one class. The analysis of pure expressions is simpler than that of generic expressions and constitutes the starting point of the complete expression model.

6.2.1 *Pure arithmetic expressions.* A pure arithmetic expressions is defined by the production of Figure 7:

arith-expr	→	*var*
	|	*arith-expr binary-arith-op arith-expr*
	|	*unary-arith-op arith-expr*

Fig. 7. Production for pure arithmetic expressions

where *binary-arith-op* and *unary-arith-op* are terminals standing for all arithmetic operators. The translation, and thus the cost, of the operators is easily determined by recalling the definitions of the compatible instruction classes and is summarized in Table VI.

Operator	Operand	Cost
+ (unary)	both	0
-, ~ (unary)	integer	$\bar{t}(alul)$
- (unary)	foating point	$\bar{t}(alul)$
+, -, &, |, ^, <<, <<	integer	$\bar{t}(alul)$
%	integer	$\bar{t}(aluh)$
+, -	floating point	$\bar{t}(aluh)$
*, /	both	$\bar{t}(aluh)$
cast	both	$\bar{t}(aluh)$

Table VI. Arithmetic operators costs

To derive the overall cost of a pure arithmetic expression, let op_i denote the operators and var_j the variables involved. The cost is then given by the expression:

$$\bar{t}(\textit{arith-expr}) = \sum_i \bar{t}(op_i) + \sum_j \bar{t}(var_j) \tag{39}$$

where $\bar{t}(var_j)$ accounts for the access cost of the j-th variable.

6.2.2 *Pure relational expressions.* Pure relational expressions are very simple since the concatenation of relational operators, generally, makes no sense. The grammar reported in Figure 8 defines this type of expressions.

rel-expr	→	*var rel-op var*

Fig. 8. Production for pure relational expressions

The cost model must account for the two possible uses of these expressions. On one hand they can be used as a condition in a selection or looping statement, while, on the other hand, can be used as subexpressions in an arithmetic or logic expression. In the latter case, the evaluation, and thus the translation, involves not only a comparison but also a jump to an appropriate statement devoted to assign the values 0 or 1 to a variable. The cost of the relational operators is summarized in Table VII.

Operator	Operand	Cost
<, <=, >, >=, ==, !=	integer	$\bar{t}(cmpl) + \bar{t}(jump)$
<, <=, >, >=, ==, !=	floating point	$\bar{t}(cmph) + \bar{t}(jump)$

Table VII. Relational operators costs

Recalling the symbols used for the arithmetic expression model, the cost of relational expressions is given by the relation:

$$\bar{t}(\textit{rel-expr}) = \sum_i \bar{t}(op_i) + \bar{t}(var_1) + \bar{t}(var_2) \tag{40}$$

where op_i indicates the relational operator and possible cast operators.

6.2.3 *Pure logic expressions.* Pure logic expressions are defined by the production reported in Figure 9.

logic-expr	→	*var*
	|	*logic-expr binary-logic-op logic-expr*
	|	*unary-logic-op logic-expr*

Fig. 9. Production for pure logic expressions

This class of expressions poses two critical problems. For efficiency reasons, in fact, many compilers translate a complex expression by introducing shortcuts. This implies that some portions of the expression might not be evaluated at all. This

aspect is accounted for by a suitable profiling mechanism, omitted here for the sake of conciseness. The second problem is related to the possibility of combining in the same expression variables or arithmetic expressions with relational expressions. To clarify this point let consider the two sample expressions of Figure 10 and their translations in assembly (on a Sparc machine).

```
(a>b) && (c>d)
        mov   0, %o4
        cmp   %o0, %o1
        ble   .LL3
        cmp   %o2, %o3
        ble   .LL3
        mov   1, %o4
.LL3    nop
```

(a)

```
x && y
        mov   0, %o4
        cmp   %o0, 0
        be    .LL3
        cmp   %o1, 0
        be    .LL3
        mov   1, %o4
.LL3    nop
```

(b)

Fig. 10. Translation of logic expression

In the code of Figure 10(a) the compare and jump instructions come from the translation template of the the two relational subexpressions `a>b` and `c>d`. In this case the cost of the logic operator `&&` is null. In the code of Figure 10(b), on the other hand, the same instructions can only be associated to the `&&` operator since the accesses to the variables `x` and `y` have zero cost. A statistical analysis of approximately 500,000 lines of C source code has shown that logic expressions of the form of Figure 10(b) account for less than the 1.5% of all logic expressions. The adopted model refers, thus, to the the case of 10(a) and the cost of logic operators is assumed to be always 0. According to this hypothesis, the overall cost of a logic expression is given by the relation:

$$\bar{t}(\textit{logic-expr}) = \sum_j \bar{t}(var_j) \tag{41}$$

6.2.4 *Generic expressions cost model.* Based on the models described in the three preceding paragraphs, a complete scheme for generic expressions translation can be built. The productions of Figure 11 summarize their grammar.

operator	→	*rel-op* | *binary-arith-op* | *unary-arith-op* | *logic-op*
expr	→	*rel-expr* | *arith-expr* | *logic-expr* | *expr operator expr*

Fig. 11. Productions for generic expressions

To correctly evaluate the overall cost, though, the order in which subexpressions are evaluated is important. In particular the syntax tree of the expression must be visited depth–first and whenever a subexpression is completely analyzed, it must be substituted with a placeholder representing a temporary variable of the appropriate type. An example of expression cost calculation is reported in Figure 12, where all the necessary substitution steps are explicitly shown.

The leftmost tree contains all three types of expression. After the first reduction, leading to the temporary placeholder `t1`, the tree only contains relational and logic

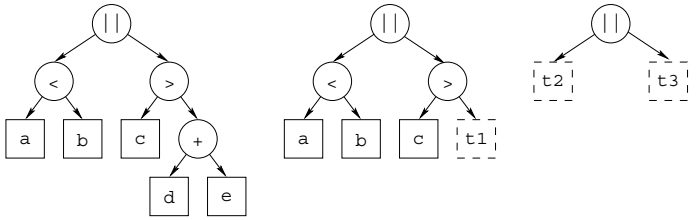


Fig. 12. Generic expression visit example

expressions. The reduction of the two relational expression into the temporary placeholders `t2` and `t3`, leads a new tree representing a pure logical expression. All the expressions that have been reduced at each step of this process are *pure*.

6.3 Assignment statements

Expressions are mainly used as *r–value* in assignment statements. Nevertheless, assignments may also be used to copy a value from a variable to a different variable, i.e. from a register or a bank of registers to a different one. In the case of an expression whose result is assigned to a variable, the assignment operator `=` has a null cost. In fact, the compiler can always arrange the translation in such a way that the operator at the root of the syntax tree of the expression assigns its result directly to the variable appearing at the left–hand side of the assignment.

An explicit copy is, on the contrary, necessary whenever the right–hand side of the assignment is a variable. In this case, the copy requires a certain number of *alul*, *mvld* or *mvst* instructions. The choice of *mvld* or *mvst* instructions rather *alul* instructions depends on the number of indirections necessary to access the two variables involved. Table VIII summarizes the cost models adopted. In this table, the number of indirections indicates the number of chained dereferentiations necessary to access a variable and the function $r(\cdot)$ indicates the number of registers necessary to store a variable.

Number of indirections		**Cost**
l–value	**r–value**	
0	0	$r(var) \cdot \bar{t}(alul)$
0	> 0	$[r(var) - 1] \cdot \bar{t}(mvld)$
> 0	0	$[r(var) - 1] \cdot \bar{t}(mvst)$
> 0	> 0	$[r(var) - 1] \cdot [\bar{t}(mvld) + \bar{t}(mvst)]$

Table VIII. Cost of the assignment atom

It is worth noting that the term -1 in the factors $r(var) - 1$ is motivated by the fact that one of the *mvld* or *mvst* instruction is already accounted for in the cost model of variable access and must not thus be considered again when evaluating the assignment itself.

The C language also provides special assignment operators such as `+=` or `-=`. In these cases, an assignment can always be expanded in its explicit form, and evaluated afterward. For this reason, no specific model has been introduced for such operators.

6.4 Selection statements

The selection statements of the C language are the `if-then-else`, the `switch-case` and the ternary operator `?:`. For the sake of conciseness this paper only reports the model of the first construct, which, by the way, can be used to implement the others. Similarly to logical expressions, the condition in an `if` atom can either be an arithmetic expression or a logic or relational expression. In the former case, an implicit comparison with zero is assumed. Whenever the condition is a relational or logic expression, the compare and jump instructions (necessary to implement the selection statement) are already accounted for during expression evaluation. In the case of an arithmetic expression or a variable, conversely, the comparison and the jump need to be explicitly associated to the `if` statement.

The `else`–branch implies the presence of an additional jump at the end of the `then`–branch code to skip the `else`–branch code. Since it is not possible to estimate at compile–time the result of the evaluation of the condition, the cost of the additional jump can be conveniently weighted by a coefficient accounting for the probability of the execution of the `then`–branch. The resulting cost model is reported in Table IX, where p_{true} is the probability that the condition evaluates to true and $p_{true} \equiv 0$ whenever an `else`–branch is missing.

Expression class	**Expression type**	**Cost**
arithmetic	integer	$\bar{t}(cmpl) + (1 + p_{true}) \cdot \bar{t}(jump)$
arithmetic	floating point	$\bar{t}(cmph) + (1 + p_{true}) \cdot \bar{t}(jump)$
logic, relational	both	$p_{true} \cdot \bar{t}(jump)$

Table IX. Cost of the `if-then-else` atom

The models of other selection statements are built according to a similar procedure and a detailed description can be found in [Dadomo 2002].

6.5 Loops

The C language provides three loop constructs: `for`, `while` and `do-while`. In the following, only the model of the `while` construct is reported. This atom requires the evaluation of a condition at the beginning of the loop and a jump from the end of the loop body back to the beginning. The discussion on the condition done for the `if-then-else` atom also apply to this case. The only difference lays in the fact that the jump back is executed at all iteration but the last one. Since most loops iterate a high number of times, the cost of the missing jump at the last iteration can be safely neglected. The resulting cost model is summarized in Table X.

Expression class	**Expression type**	**Cost**
arithmetic	integer	$\bar{t}(cmpl) + 2 \cdot \bar{t}(jump)$
arithmetic	floating point	$\bar{t}(cmph) + 2 \cdot \bar{t}(jump)$
logic, relational	both	$\bar{t}(jump)$

Table X. Cost of the `while` atom

Again, the other loop statements are described in full detail in [Dadomo 2002].

6.6 Function calls

Function calls modeling requires considering two aspects: jumping from the caller to the callee and back, and passing the actual parameters to the function. Jumps are always implemented by means of dedicated assembly instructions, belonging to the *call* class of the kernel instruction set and can thus be easily modeled. The problem of parameter passing, on the other hand, is more complex and in fact requires the notion of *stack*. In the ideal reference architecture, the stack is a variable–sized register bank plus an additional register playing the role of stack pointer. According to this assumption, a *push* operation implies copying as many registers as required by the specific datum considered and consequently updating the stack pointer register. With the exception of the direction of the copy, a *pop* operation involves exactly the same steps.

A function call is thus translated into a sequence of push operations to copy the formal parameters onto the stack and a jump to the beginning of the function code. It is worth noting that while executing the code of the function, all parameters must be popped from the stack into local variables. The cost related with the sequence of pop operations cannot be associated to an atom of the called function since it is implicit in its prototype definition. Nevertheless, a call to a function necessarily involves both pushing and popping all parameters, and for this reason, both costs are associated to the *function-call* atom in the caller function.

A similar translation strategy can be adopted for the `return` statement. In this case the called function pushes the return value onto the stack and requires popping it after passing the control back to the caller.

The resulting cost models are summarized in Table XI, where $var_{in,j}$ are the actual parameters, var_{out} is the returned variable and the function $r(\cdot)$ returns the number of registers necessary to store a variable.

Statement	**Cost**
function call	$\bar{t}(call) + [\bar{t}(mvld) + \bar{t}(mvst)] \cdot \sum_j r(var_{in,j})$
function return	$\bar{t}(call) + [\bar{t}(mvld) + \bar{t}(mvst)] \cdot r(var_{out})$

Table XI. Cost of the function call/return

It is worth noting that copying the actual parameters—as well as the return value—from (or to) the origin registers, which are known at compile–time, to a different register bank whose position is determined at run–time by means of the stack pointer register, involves *mvld* and *mvst* instructions and not *alul* instructions.

7. EXPERIMENTAL RESULTS

To validate the entire methodology, different benchmarks have been analyzed and the obtained results compared with assembly–level data derived from processor–specific instruction set simulators. The platform on which validation has been carried out is based on the Intel i486 Embedded Processor and the GNU `gcc` compiler under Linux RedHat 7.2. To assess the accuracy of energy figures, an indirect approach has been followed, based on the assembly–level characterization of single instructions [Tiwari et al. 1996]. All the analysis steps outlined in Figure 1 have been implemented in a completely automated toolset. The next paragraphs give

the flavor of how the different steps of the methodology can be applied and report the results obtained considering a significant benchmark set.

7.1 Sample analysis flow

This paragraph briefly discusses the different phases of the analysis applied to a very small code portion. The main purpose is to clarify some details of the methodology and to illustrate how the theoretical formulation presented above applies to the practice. Let the starting point be the simple code of Figure 13(a). The decomposition phase leads to the 11 atoms listed in Figure 13(b). It is important to note that a reference to the source line number is maintained for each atom to enable backannotation of low–level figures up to the source code.

```
1.  i=n=0;
2.  while( i < 10 ) {
3.    if( i % 2 == 0 ) {
4.      n = n + i;
5.    } else {
6.      n = n * (i - 1);
7.    }
8.    i++;
9.  }
```

(a)

$A_{s,0}$	*st-assign*	`n=0`	line 1
$A_{s,1}$	*st-assign*	`i=n`	line 1
$A_{s,2}$	*while*	`while(){}`	line 2
$A_{s,3}$	*expr*	`i<10`	line 2
$A_{s,4}$	*if-then-else*	`if(){}`	line 3
$A_{s,5}$	*expr*	`i%2==0`	line 3
$A_{s,6}$	*st-assign*	`n=`	line 4
$A_{s,7}$	*expr*	`n+i`	line 4
$A_{s,8}$	*st-assign*	`n=`	line 6
$A_{s,9}$	*expr*	`n*(i-1)`	line 6
$A_{s,10}$	*expr*	`i++`	line 8

(b)

Fig. 13. Atom decomposition of a sample source code

The atoms in the second column of Figure 13(b) are then translated according to the schemes presented in Section 6 and lead to the KIS program of Figure 14.

$A_{s,0}$	*st-assign*	`n=0`	line 1	–
$A_{s,1}$	*st-assign*	`i=n`	line 1	–
$A_{s,2}$	*while*	`while(){}`	line 2	*jump*
$A_{s,3}$	*expr*	`i<10`	line 2	*cmpl*, *jump*
$A_{s,4}$	*if-then-else*	`if(){}`	line 3	$p_{true} \cdot jump$
$A_{s,5}$	*expr*	`i%2==0`	line 3	*aluh*, *cmpl*, *jump*
$A_{s,6}$	*st-assign*	`n=`	line 4	–
$A_{s,7}$	*expr*	`n+i`	line 4	*alul*
$A_{s,8}$	*st-assign*	`n=`	line 6	–
$A_{s,9}$	*expr*	`n*(i-1)`	line 6	*aluh*, *alul*
$A_{s,10}$	*expr*	`i++`	line 8	*alul*

Fig. 14. Translation of the sample source code into a KIS program

This simple code requires no input data and thus the profiling information can be derived simply by compiling an instrumented version of the source code and by running it on a generic host platform. The results of profiling refer to atoms, i.e. for each atom an execution count is collected. Thus far, the analysis procedure does not make explicit reference to any specific target architecture. At this

point, the different KIS instructions can be mapped to the selected processor simply by looking up their cost into a technology library. Table XII summarizes the data available in this final phase of the analysis flow, where the execution time is expressed in clock cycles and the energy is reported as average current in mA.

Atom	Line	*KIS*	Count	Time	Energy
$A_{s,0}$	1	–	1	0.000	0.000
$A_{s,1}$	1	–	1	0.000	0.000
$A_{s,2}$	2	*jump*	11	1.186	1.046
$A_{s,3}$	2	*cmpl*, *jump*	11	2.620	2.580
$A_{s,4}$	3	$p_{true} \cdot jump$	10	0.593	0.523
$A_{s,5}$	3	*aluh*, *cmpl*, *jump*	10	15.119	9.177
$A_{s,6}$	4	–	5	0.000	0.000
$A_{s,7}$	4	*alul*	5	1.434	0.681
$A_{s,8}$	6	–	5	0.000	0.000
$A_{s,9}$	6	*aluh*, *alul*	5	13.933	7.278
$A_{s,10}$	8	*alul*	10	1.434	0.681

Table XII. Translation of the sample source code into a KIS program

The total execution time is thus $T_{s,j} = 290.161$ clock cycles. To calculate the total energy let the clock frequency be $f = 33$MHz and the power supply voltage be $V_{dd} = 3.3$V. Under this operating conditions the total energy is $E_{s,j} = 19.982\mu$J, corresponding to an average power dissipation of $W_{s,j} = 2.086$W. It is worth noting that to derive an estimate for a different processor it is sufficient to replace the values of the last two columns of Table XII, which can be done effortlessly.

7.2 Results

The same methodology has been applied to more significant testbenches such as a 16-bit CRC encoder, a Base64 encoder, a prime factor decomposition algorithm used in cryptography and an ADPCM compression/decompression algorithm. Table XIII collects the results and allows comparing the estimated values with the actual ones. The average absolute error for both energy and timing is below 5%.

	Time (Clock cycles)			Energy (μJ)		
Program	**Real**	**Estimated**	**Error (%)**	**Real**	**Estimated**	**Error (%)**
`loop`	25256	25394	-0.54	1867	1888	-1.11
`factorial`	56940	61030	-6.70	4169	5137	-18.84
`mcd3`	32193	32094	+0.31	2379	2373	+0.27
`arith`	48294	49949	-3.31	3553	3678	-3.41
`crc16`	226093	246012	-8.10	17510	18539	-5.55
`base64enc`	162455	149211	+8.88	11501	11098	+3.63
`real2frac`	46735	48688	-4.01	3407	3533	-3.57
`pfactor`	84058	84158	-0.12	6264	6500	-3.63
`adpcm`	41384	37629	-9.07	3213	3275	+1.93
Average			**4.55**			**4.65**

Table XIII. Execution time and energy consumption results

These results refer to the simplest version of the mathematical model, namely the one that assumes the statistical correction function $\delta(\cdot) \equiv 0$. Nevertheless, preliminary results suggest that using $\delta(\cdot) = const$ would significantly compensate the errors that the current model exhibits. To prove this, consider the two plots, one for the execution time and one for the energy consumption, shown in Figure 15.

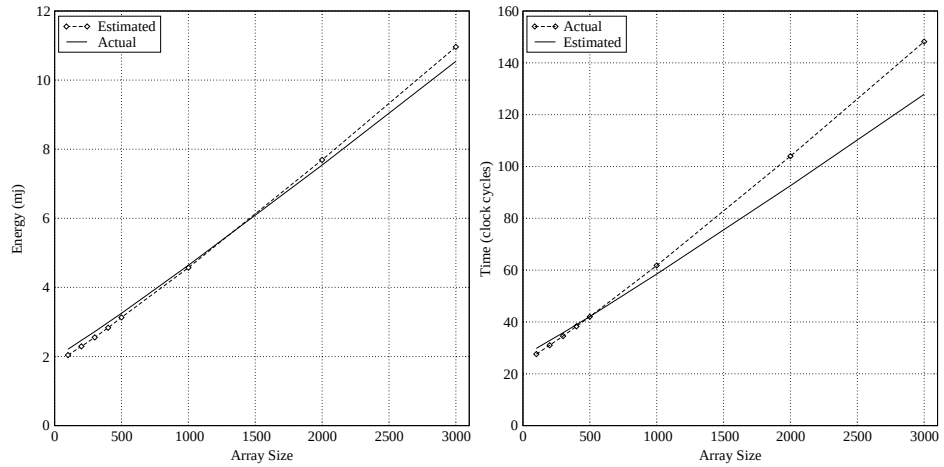


Fig. 15. Data–size dependence of estimation error

The reported measures, referring to the Base64 encoding algorithm, show that the gap between actual and estimated figures is proportional to the size of the data being encoded, which in turn, is strictly correlated to the number of iterations of the main loop. Recalling Equation (23) and noting that the statistical term $\delta(\cdot)$ is multiplied by the execution count $N_{s,i,j}$, it seems reasonable to suppose that a constant term would thus be sufficient to compensate such a deviation.

8. CONCLUSION

The goal of this paper has been to provide the designer a comprehensive and theoretically well–funded methodology to analyze both timing and energy characteristics of software for embedded applications. The key elements are the possibility to maintain the analysis at the same (source) abstraction level at which the designer is coding the application, while achieving an accuracy close to that of a lower–level profile–driven analysis of the compiled code. The paper presented a strategy to analyze C specifications, by identifying basic elements called *atoms* and *translation templates schemes*, in a manner that is fairly independent from the target microprocessor. All the different elements composing a program have been considered and properly modeled, including function calls. Retargetabiliy is another important achievement of the methodology, since specific information on timing and energy are stored in technology libraries allowing easy account for different microprocessors. The outcome is a system–level model allowing the designer to feed the program with stimuli and to obtain the estimates without performing any

additional time–consuming compilation or simulation. Experimental assessment of the methodology has been carried out by considering a benchmark set composed of code coming from real embedded applications. The average accuracy in predicting time and energy is within the error band of 5% with runtimes is the order of seconds (assuming the availability of technology libraries), thus enabling effective design space exploration. Current effort is devoted to refine the statistical correction term and to include additional models tailored to account for the contribution of black–box portions of the specification, such as precompiled libraries.

REFERENCES

Benini, L. and Micheli, G. D. 2000. System–level power optimization: techniques and tools. *ACM Transaction on Deasign Automation of Electronic Systems 5,* 2 (Apr.), 115–192.

C.Brandolese, W.Fornaciari, F.Salice, and D.Sciuto. 2000. An instruction–level functionality–based energy estimation model for 32-bits microprocessors. In *Proceedings of the IEEE Design Automation Conference*. IEEE, Los Angeles (CA), 346–351.

C.Brandolese, W.Fornaciari, F.Salice, and D.Sciuto. 2001. Source–level execution time estimation of c programs. In *Proceedings of the IEEE/ACM International Symposium on Hardware/Software Codesign*. IEEE, Los Angeles (DK), 98–103.

Dadomo, M. 2002. Estimation of the energy/timing characteristics of source–level c code. M.S. thesis, CEFRIEL Research Centre. Technical Report, N. 02002.

Giusto, P., Martin, G., and Harcourt, E. 2001. Reliable estimation of execution time of embedded software. In *Proceedings of the IEEE International Conference on Design, Automation and Test in Europe*. IEEE, Munich (D), 580–588.

Hsieh, C.-T. and Pedram, M. 1998. Microprocessor power estimation using profile–driven program synthesis. *Transaction on CAD 17,* 11 (Nov.), 1080–1089.

Landman, P. and Rabaey, J. 1993. Power estimation for high level synthesis. In *Proceedings of the IEEE EDAC-EUROASIC*. IEEE, Paris (F), 361–366.

Lazarescu, M. T., Bammi, J. R., Harcourt, E., Lavagno, L., and Lajolo, M. 2000. Compilation–based software performance estimation for system level design. In *Proceedings of the IEEE International High–Level Design Validation and Test Worksop*. IEEE, Berkeley (CA), 167–172.

Macii, E., Pedram, M., and Somenzi, F. 1998. High–level power modeling, estimation and optimization. *IEEE Transaction on CAD 17,* 11 (Nov.), 1061–1079.

Malik, S., Martonosi, M., and Li, Y. 1997. Static timing analysis of embedded software. In *Proceedings of the IEEE Design Automation Conference*. IEEE, Anaheim (CA), 147–152.

Mehta, H., Owens, R., and Irwin, M. 1996. Instruction level power profiling. In *Proceedings of the IEEE ICASSP*. IEEE, Atlanta (GA), 3326–3329.

Russell, J. T. and Jacome, M. F. 1998. Software power estimation and optimization for high performance, 32-bit embedded processors. In *Proceedings of the IEEE International Conference on Computer Design*. IEEE, Austin (TX), 328–333.

Tiwari, V., Malik, S., Wolfe, A., and Lee, M. T.-C. 1996. Instruction level power analysis and optimization of software. *Kluwer Academic Publisher Journal of VLSI Signal Processing 13,* 1-2, 223–233.