Microprocessors and Microsystems 77 (2020) 103200

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

An early-stage statement-level metric for energy characterization of embedded processors

Vittoriano Muttillo*, Paolo Giammatteo, Vincenzo Stoico, Luigi Pomante

Università degli Studi dell'Aquila, Center of Excellence DEWS, Italy

ARTICLE INFO

Article history:
Received 8 November 2019
Revised 16 May 2020
Accepted 2 July 2020
Available online 8 July 2020

Keywords:
Embedded processor
Energy consumption
Profiling
Benchmarking
Metrics

ABSTRACT

This work presents an early stage statement-level metric for energy characterization of embedded processors. Definition and the framework for metric evaluation are provided. In particular, such a metric is based on an existing assembly-level analysis and some profiling activities performed on a given C benchmark, and it is related to the average energy consumption of a generic C statement, for a given target processor. Its evaluation is performed with a one-time effort and, once available, it can be used to rapidly estimate the energy consumption of a given C function for all the considered processors. Two reference embedded processors are then considered in order to show an example of usage of the proposed metric and framework.

1. Introduction

Energy consumption is one of the most critical design issues in the embedded systems domain. In particular, the need to guarantee even longer life for all battery-powered devices is one of the main problems that affect the design activities. Indeed, the choices made by designers at the system-level of abstraction, can drastically influence the final system energy consumption, since different optimizations can be considered in the whole *Electronic System Level* (ESL) design flow [1]. Therefore, different energy consumption models can be taken into account in order to estimate the energy consumption of the final system implementation. Such models can be related to processors, *Application Specific Integrated Circuit* (ASIC), memories, and the interconnections among them. Moreover, the models can be at different levels of abstraction and granularity, mainly depending on the required estimation accuracy.

Since this work focuses on embedded processors, Fig. 1 shows the typical abstraction levels involved in a classical ESL design flow for embedded processors [2]. The first abstraction level, called *Functional*, catches very few non-functional static processor features, as average *Clock cycles Per Instruction* (CPI), static power dissipation, etc. The *Architectural/ISS* abstraction level involves the knowledge of the *Instruction Set Architecture* (ISA) and it is normally supported by a so-called *Instruction Set Simulator* (ISS) to perform several kinds of dynamic analysis. The *Pipeline-accurate Architectural/ISS* abstraction level adds details about the behavior of the pipeline into the simulator, providing a more refined processor model. Finally, the *Cycle-accurate Micro-Architectural* abstraction level introduces further details about the processor architecture in terms of *Control Unit* and *Data Path*, allowing a cycle-accurate analysis of the final implementation. The following work is located between the first two levels of the design flow. Here, a *statement-level* metric called J4CS (*Joule for C Statement*) is proposed, in order to estimate the average energy consumption associated to the execution of a generic C statement by means of a given target processor (i.e., a hybrid functional/instruction level approach). However, two main issues must be firstly clarified. The first one is related to the definition of "generic C statement". This work exploits the same approach presented in [3], where it has been defined, by adopting an empirical approach. In particular, it refers to the way a common profiling tool as *GCov* [4] performs C statements identification and counting, when profiling their execution. The second issue is related to the fact that J4CS is influenced also by the used C compiler (and the adopted optimization options) since the optimizations performed by a compiler can lead to different energy consumption value. Furthermore, there are several ways to manage such an influence. One is to explicitate the used compiler, possibly giving rise to a J4CS for each processor/compiler pairs. Another one is to consider the average of the results obtained by using the most diffused C compilers. In such a case, the issue can be managed through a statistical characterization of J4CS, considering

* Corresponding author.
E-mail addresses: vittoriano.muttillo@univaq.it (V. Muttillo), paolo.giammatteo@univaq.it (P. Giammatteo), vincenzo.stoico@student.univaq.it (V. Stoico), luigi.pomante@univaq.it (L. Pomante).

https://doi.org/10.1016/j.micpro.2020.103200

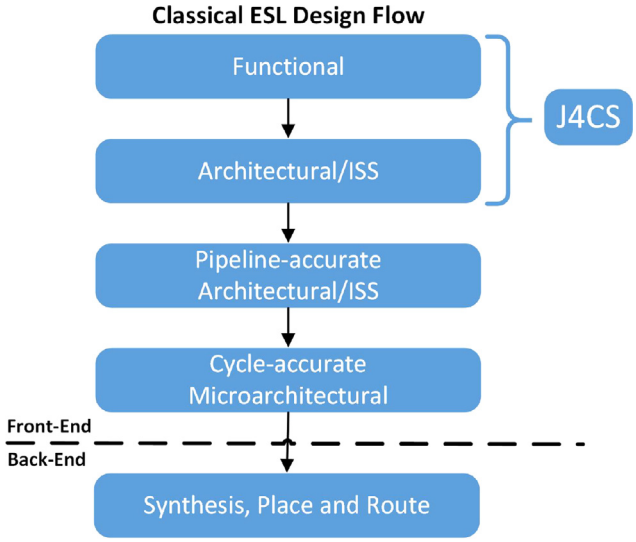


Fig. 1. Classical ESL design flow for embedded processors.

both different compilers and different compiler optimization options. This can be obtained by evaluating a set of values related to *Min, Max, Average, Standard Deviation* and by also trying to identify an associated statistical distribution. In such a context, this work intends to explore the statistical characterization approach by providing a metric useful to estimate, at an early-stage design phase, the energy consumption related to the execution of SW on a target embedded processor, so characterizing the processor itself. J4CS is suitable for very fast estimation, comparison and selection activities. J4CS evaluation considers the assembly-level analysis presented in [5], as explained in Section 3, and exploits the framework developed in [3]. The obtained value can be assigned to each statement of a given C function and exploited, with a host-based source-level profiling, in order to estimate the total amount of energy consumed when the C function with specific inputs is executed on the target embedded processor. According to this scenario, this work intends to extend the ideas addressed in [6] by providing in addition:

- a comparison of related works, with a focus on platforms and accuracy;
- more detailed aspects about the proposed J4CS evaluation framework;
- the addition of the Intel CISC 8051 micro-controller and the introduction of new target boards;
- a statistical analysis considering different data types and the correlation between assembly instructions and C statements;
- error estimation associated with a validation benchmark, and the exploitation of the *Affinity* [7] metric value to increase J4CS accuracy;

The remainder of the paper is organized as follows: Section 2 describes some relevant works related to the power/energy consumption estimation and evaluation problem. Section 3 formally defines the proposed J4CS metric. Section 4 presents the J4CS metric evaluation framework applied to two reference embedded processors. Section 5 shows how the obtained values can be used in order to estimate and compare the energy consumption associated with the execution of a given C function on a specific board. Finally, Section 6 closes the paper with conclusions and future works description.

2. Related works

Different abstraction levels can be considered in order to describe a processor and its behavior. Accordingly, several energy estimations can be performed [8], as previously stated. Starting from lower-levels of abstraction, such as gate or *Register-Transfer* (RT) ones [9], a lot of works consider the problem of estimate power/energy consumption by using time-consuming simulators [10–12]. Other works start from an accurate modeling of the target ISS [13,14], but this still requires a considerable time both for the modeling and the simulation activities.

Other studies present energy estimation approaches at (assembly) instructions level. These energy consumption estimation techniques usually consider either ISS or low-level assembly code analysis in order to obtain power characterization of the application, at the expense of a higher consumption of estimation time. Instruction-level energy estimation can be divided into two types: (1) measurement-based techniques that do not consider processors architectural details, while trying to extract an average energy cost per instruction [15,16]; (2) pipeline and accurate architectural analysis that take into account pipeline stages and complex architectural details [17–19]. The same problems arise in approaches that involve the introduction of some kind of *Virtual Instruction Set* (e.g., [20]), but that still require some explicit detailed knowledge of the target processors architecture.

Several works analyze the energy consumption of processor functional units, where the total energy consumption of a target processor is obtained by the sum of energy dissipation of these functional components [21]. Also for the so called *Functional-Level Power Analysis* (FLPA) methods, the energy model is derived by means of simulation or on-target measurements [22,23]. In order to bring the gap between instruction and functional level power estimation, Blume et al. [24] and Brandolese et al. [25] has presented a hybrid approach that combines these two methods to estimate energy consumption.

Other works try to rise the abstraction level, by going towards the system level one. This is often done by directly considering source-code [5], but also this kind of analysis can involve different time-consuming activities strictly related to the need of taking into account the peculiarities of the considered target processors. With respect to the source-code analysis, the works in [26,27] present a statement-level timing estimation that is used to evaluate power/energy metrics directly on the base of timing executions and profiling activities. A work that tries to fill up the gap between the reduced simulation time of high-level dynamic analysis of source-code, with the accuracy of low-level dynamic analysis, is [28], that introduces an intermediate pseudo instruction set for analyzing applications and HW architectures, using an approach still similar to [20]. Finally, a statement-level energy estimation based on GCC has been proposed in [29], where an higher absolute error was measured due to the simple and basic method used for the measurements. Table 1 presents a comparison of the considered works according to several features [30].

In such a context, the work presented in this paper is close to the work presented in [5]. The main difference is that the purpose of this work is to reduce the time needed for estimation activities by means of a strategy that allows to quickly evaluate and select processors in an early-stage analysis. In fact, the estimation of the energy consumed during SW execution is very fast since it is based only on a host-based source-level profiling performed one-shot independently from the number of target processors considered. More detailed ISA-related analysis, if needed, can be then performed by focusing only on the selected processors.

3. Metric definition

The transistor-level power consumption of a microprocessor [14] during the execution of a given program can be evaluated as:

Table 1
Comparison of considered power estimation works.

Work	Year	Target	Simulator and/or Estimation Model	Accuracy	Abstraction Level	Benchmark
[10]	1999	ARM710a	ARMulator	5%	Cycle Level	Dhrystone
[11]	2000	VLIW processor	SimplePower	15%	Cycle Level	Custom
[12]	2002	ARM920	Armulator	N.A.	Cycle Level	MPEG4
[13]	1994	Intel 486DX2-S SPARClite 934	Custom	N.A.	Cycle Level	Custom
[14]	2001	StrongARM SA-1100 Hitachi SH-4	JouleTrack	$2\% \leq acc \leq 8\%$	Cycle Level	Custom
[15]	2001	ARM7TDMI	Regression Model	$2\% \leq acc \leq 6\%$	Instruction Level	Custom
[16]	2002	ARM7TDMI	Regression Model	$2\% \leq acc \leq 6\%$	Instruction Level	Custom
[17]	2000	VLIW Core	Math Model	$3\% \leq acc \leq 16\%$	Instruction Level	Custom
[18]	2005	ARM7TDMI	Math Model	$acc \leq 5\%$	Instruction Level	Custom
[19]	2009	LEON3	Custom Framework	N.A.	Instruction Level	Custom
[20]	2000	Motorola MC68000 ARM7TDMI	TOSCA OCCAM2	$1\% \leq acc \leq 20\%$	Instruction Level	ILC 16
[24]	2007	ARM926EJ-S C55x DSP	ARMulator XDS510PP Plus	$4\% \leq acc \leq 9\%$	Instr./Func. Level	Digital Signal Processing Tasks
[25]	2002	Intel i960JF Intel i960HD SPARClite MB86934 ARM7TDMI	Instruction Characterization	$acc \leq 9\%$	Instr./Func. Level	Custom
[21]	2002	TMS320C6201	Math Model	$acc \leq 4.2\%$	Functional Level	Digital Signal Processing Algorithms
[22]	2007	TMS320C6416	Functional Model	$acc \leq 10\%$	Functional Level	Custom
[23]	2005	TMS320C6416	Functional Model	$acc \leq 9\%$	Functional Level	FIR filter
[5]	2007	ARM9TDMI ARM TRM	SystemC Sim.	$acc \leq 11\%$	System Level	Custom
[26]	2001	N.A.	Math Model	$acc \leq 11\%$	System Level	Custom
[27]	2010	ReISC III	Math Model	$acc \leq 6\%$	System Level	WCET suite
[28]	2002	Intel i486	Formal Model	$acc \leq 5\%$	System Level	Custom
[29]	2016	Tiva TM4C123G	Instruction Characterization	$acc \leq 30\%$	System Level	Custom
This Work	2019	RTAX UT699 AT89C51	TSIM, Dalton	$1\% \leq acc \leq 15\%$	System Level	Custom

$$P_{tot} = P_{dyn} + P_{stat} = P_{switching} + P_{short-circuit} + P_{stat} = \alpha \cdot C_L \cdot V_{dd}^2 \cdot f + I_{sc} \cdot V_{dd} + V_{dd} \cdot I_{leak} \quad (1)$$

where P_{tot} is the total power consumption made up of dynamic and static power contributions, α is the node transition activity factor (normally, $\alpha = \frac{1}{2}$), C_L is the average switched capacitance per clock cycle during the execution of the program, V_{dd} is the supply voltage, and f is the clock frequency. $P_{short-circuit}$ is related to the direct-path short circuit current I_{sc}, which arises when both the NMOS and PMOS transistors are simultaneously active, conducting current directly from supply to ground. P_{static} is related to the leakage current I_{leak}, i.e., the current that flows through the circuit to ground. The works [17,31] show that the switching activity represents 90% of the total power consumption, so estimations mainly focus on it.

Considering the execution time associated to a given SW program (Δt), it is possible to evaluate the total energy consumption as:

$$E_{tot} = P_{tot} \cdot \Delta t = \alpha \cdot C_{tot} \cdot V_{dd}^2 + I_{sc} \cdot V_{dd} \cdot \Delta t + V_{dd} \cdot I_{leak} \cdot \Delta t \quad (2)$$

where C_{tot} is the total switched capacitance. Changing the clock frequency (and so decreasing/increasing the program execution time) doesn't change C_{tot} [14] and so the energy consumption decrease/increase linearly with the scaled frequency with the slope proportional to the amount of leakage. Eq. (1) and Eq. (2) define the power processor model at circuit level.

Then, by considering the assembly-level software model presented in [32], it is possible to define the energy consumption associated with a given program also as:

$$\bar{E} = \sum_i (B_i \times N_i) + \sum_{i,j} (O_{i,j} \times N_{i,j}) + \sum_k (E_k) \quad (3)$$

where B_i is the base cost (i.e., the energy cost associated to the execution of a specific assembly instruction), N_i is the number of times a specific assembly instruction has been executed, $O_{i,j}$ represents the circuit state overhead (i.e., the energy overhead due to the execution of two separated sequential instructions), and E_k is the energy contribution related to other inter-instruction effects (i.e. stalls or cache misses) [13].

The higher the abstraction level, the lower the estimation accuracy, but also the lower the timing simulation activities needed for power estimation. Furthermore, at instruction level, considering the average power consumed by a microprocessor while running a program, it is possible to simplify Eq. (2) and Eq. (3) by considering $\bar{P} = \bar{I} \times V_{dd}$, where $\bar{I}$ is the average current and V_{dd} the voltage supply. So, the average energy consumed by a program can be expressed by: $\bar{E} = \bar{P} \times N \times \tau$, where N is the number of program clock cycles and τ is the clock period [32].

Thus, while taking into account the formulas described above, the approach proposed in this work exploits some benchmark activities on a specific set of C functions in order to evaluate a metric related to the average energy consumption per C statement, as described below, to estimate a statistical interval of energy consumption.

3.1. Main assumptions

As described in [5], many embedded microprocessors have a statistical property of constant energy consumption for each executed assembly instruction. So, the proposed idea is to apply the same approach to a higher abstraction level (i.e., statement-level) by characterizing the energy cost (e.g., probabilistic distribution in terms of distribution parameters) associated to the execution of a C statement. This is achieved by performing several simulations in

order to consider a meaningful number of execution paths depending on different inputs.

In order to perform statistical analysis, some assumptions must be made:

- if the program has a huge amount of lines of code (LOC), then the energy consumed for each instruction can be considered constant without great loss of accuracy [14];
- each statement contributes with the same weight for the evaluation of the total energy consumption, since the analysis is given from a statistical point of view, while the evaluation does not consider the influence of operators and variables on the total energy cost;
- an average number of assembly instructions per C statement is considered, since the number of instructions can vary depending on several factors (i.e., memory accesses, addressing modes, register file size, branch mis-predictions, etc.);
- an average number of clock cycles for a pipeline stage divided by the processor efficiency Φ is considered in order to evaluate the mean energy consumed by each executed assembly instruction, as presented in [5].

By these assumptions, it is possible to define the following energy model.

3.2. Proposed energy model

The approach proposed in this work performs statement-level energy estimation using statistical analysis and approximate predictions.

Definition 3.1. $Z = \{z_i \mid i = 1, 2, \ldots, n\}$ is a set of n software functions, $B = \{b_{i,k} \mid i = 1, 2, \ldots, n \ \wedge \ k = 1, 2, \ldots, t\}$ is a set of t randomly generated inputs for each function z_i, and $P = \{p_j \mid j = 1, 2, \ldots, m\}$ is a set of m processing units (i.e., processors that are able to execute the considered software functions).

Definition 3.2. *Total Energy Consumption for a Generic Software Function*:

Let $CS(z_i, b_{i,k})$ the number of statements executed for a generic software function z_i with input $b_{i,k}$ (evaluated by considering a statement-level execution trace representing the sequence of the executed statements), then the total energy consumed $E_T(p_j, z_i, b_{i,k})$ to execute the whole function z_i on processor p_j with input $b_{i,k}$ is:

$$E_T(p_j, z_i, b_{i,k}) = \sum_{s=1}^{CS(z_i,b_{i,k})} E_s(p_j, z_i, b_{i,k}) \tag{4}$$

where $E_s(p_j, z_i, b_{i,k})$ is the amount of energy consumption related to the statement s in the execution trace.

The energy consumption is different each time a generic statement s inside the function z_i is executed, because, depending on involved data location (i.e., memory, cache or register), data type size (i.e., 8, 16, 32, 64 bit), and statement complexity, the number of needed assembly instructions is different.

If $I_s(p_j, z_i, b_{i,k})$ is the number of assembly instruction required to execute statement s of function z_i on processor p_j with input $b_{i,k}$ (evaluated by considering an assembly-level execution trace representing the sequence of the executed instructions), and $E'_{l,s}(p_j, z_i, b_{i,k})$ is the corresponding energy consumed by each assembly instruction l of statement s, then:

$$E_s(p_j, z_i, b_{i,k}) = \sum_{l=1}^{I_s(p_j,z_i,b_{i,k})} E'_{l,s}(p_j, z_i, b_{i,k}) \tag{5}$$

$$E_T(p_j, z_i, b_{i,k}) = \sum_{s=1}^{CS(z_i,b_{i,k})} \sum_{l=1}^{I_s(p_j,z_i,b_{i,k})} E'_{l,s}(p_j, z_i, b_{i,k}) \tag{6}$$

Simplify Eq. (6), we can consider the mean energy consumption value $\overline{E}'_s(p_j, z_i, b_{i,k})$ for assembly instruction in the execution trace belonging to statement s inside the function z_i executed on processor p_j with input $b_{i,k}$ in this manner:

$$\overline{E}'_s(p_j, z_i, b_{i,k}) = \frac{1}{I_s(p_j, z_i, b_{i,k})} \cdot \sum_{l=1}^{I_s(p_j,z_i,b_{i,k})} E'_{l,s}(p_j, z_i, b_{i,k}) \tag{7}$$

$$\begin{aligned} E_s(p_j, z_i, b_{i,k}) &= \sum_{l=1}^{I_s(p_j,z_i,b_{i,k})} E'_{l,s}(p_j, z_i, b_{i,k}) \\ &= I_s(p_j, z_i, b_{i,k}) \cdot \overline{E}'_s(p_j, z_i, b_{i,k}) \end{aligned} \tag{8}$$

We can re-write Eq. (6) in this way:

$$E_T(p_j, z_i, b_{i,k}) = \sum_{s=1}^{CS(z_i,b_{i,k})} I_s(p_j, z_i, b_{i,k}) \cdot \overline{E}'_s(p_j, z_i, b_{i,k}) \tag{9}$$

When the executed code of function z_i is longer enough, the mean energy consumption $\overline{E}'_s(p_j, z_i, b_{i,k})$ per assembly instruction on statement s can be considered constant among different statements without great loss of accuracy [14]. Under this assumption, at system-level we have:

$$\left\{ \begin{array}{l} \forall l \in \{1, 2, \ldots, I_s(p_j, z_i, b_{i,k})\} \\ \forall s \in \{1, 2, \ldots, CS(z_i, b_{i,k})\} \end{array} \right\} \Rightarrow \ \overline{E}'_s(p_j, z_i, b_{i,k}) \cong \overline{E}'(p_j) \tag{10}$$

$$E_T(p_j, z_i, b_{i,k}) = \sum_{s=1}^{CS(z_i,b_{i,k})} I_s(p_j, z_i, b_{i,k}) \cdot \overline{E}'(p_j) \tag{11}$$

where $\overline{E}'(p_j)$ is the approximate average assembly instruction energy consumption on processor p_j defined as follow [5]:

Definition 3.3. *Average Assembly Instruction Energy*:

The average energy consumption associated to a generic assembly instruction executed on a processor p_j can be defined as:

$$\bar{E}'(p_j) = \frac{\overline{MPC}(p_j)}{\phi(p_j) \cdot f(p_j)} \tag{12}$$

where $\overline{MPC}(p_j)$ is the *Mean Power Consumption*, $f(p_j)$ is the *Frequency*, and $\phi(p_j)$ is the *Power Efficiency* associated to processor p_j, while $\phi(p_j)$ is related to the MIPS parameter, normally provided on processors data-sheets [33].

Simplify Eq. (11) using the average number of assembly instruction executed, we have:

$$\bar{I}'(p_j, z_i, b_{i,k}) = \frac{1}{CS(z_i, b_{i,k})} \cdot \sum_{i=1}^{CS(z_i,b_{i,k})} I_s(p_j, z_i, b_{i,k}) \tag{13}$$

$$I(p_j, z_i, b_{i,k}) = \sum_{s=1}^{CS(z_i,b_{i,k})} I_s(p_j, z_i, b_{i,k}) = \bar{I}'(p_j, z_i, b_{i,k}) \cdot CS(z_i, b_{i,k}) \tag{14}$$

where $\bar{I}'(p_j, z_i, b_{i,k})$ is the mean number of assembly instructions executed for each generic statement C belonging on software function z_i on the processor p_j with input $b_{i,k}$, while $I(p_j, z_i, b_{i,k})$ is the total number of assembly instruction executed. Finally, we can define the total energy consumption of a software function z_i running on processor p_j with input $b_{i,k}$ as follow:

$$E_T(p_j, z_i, b_{i,k}) = \overline{E}'(p_j) \cdot I(p_j, z_i, b_{i,k}) \tag{15}$$

$$E_T(p_j, z_i, b_{i,k}) = \overline{E}'(p_j) \cdot \bar{I}'(p_j, z_i, b_{i,k}) \cdot CS(z_i, b_{i,k}) \tag{16}$$

The evaluation of $I(p_j, z_i, b_{i,k})$ in Eq. (15) is time consuming, since it is needed to evaluate this value each time using an ISS execution, while $CS(z_i, b_{i,k})$ in Eq. (16) considers the use of static profiling tools on host environment and does not need to have the real or emulated target processor. Furthermore, the $\bar{I}'(p_j, z_i, b_{i,k})$ value can be evaluated using a statistical approach, as presented in the next section.

3.3. J4CS Metric

Starting from Eq. (16), we need information about the average number of assembly instruction executed $\bar{I}'(p_j, z_i, b_{i,k})$ for each generic statement C belonging to function z_i executed on a processor p_j with input $b_{i,k}$. This feature can be considered as a fixed distribution evaluated on a selected function set (as much as possible characterizing all the possible functionalities used to realize embedded software application) and re-used to evaluate energy consumption associated to a generic C function executed on a processor p_j. For this, it is possible to exploit two profiling activities on a specific selected benchmark:

- by means of an ISS to find the number of executed assembly instructions $I(p_j, z_i, b_{i,k})$;
- by means of *GCov* [3] on a host-based compilation it is possible to find the number of executed C statements $CS(z_i, b_{i,k})$;

Then, it is possible to define a statement-level energy consumption metric as follows below.

Definition 3.4. *J4CS (Joule for C Statements)*:
Let $Z = \{z_i \mid i = 1, 2, \ldots, n\}$ a benchmark set of n reference leaf C functions (i.e., no other internal nested function calls). The J4CS metric is the ratio between the number of assembly instructions executed by the target processor p_j running the functions z_i, and the number of executed C statements multiplied by the average energy of an assembly instruction execution $\bar{E}'(p_j)$, i.e.:

$$J4CS(p_j) = \left\{ \bar{E}'(p_j) \cdot \bar{I}'(p_j, z_i, b_{i,k}) = \bar{E}'(p_j) \cdot \frac{I(p_j, z_i, b_{i,k})}{CS(z_i, b_{i,k})}, \forall z_i \in Z, b_{i,k} \in B \right\} \quad (17)$$

where $I(p_j, z_i, b_{i,k})$ is the number of assembly instructions executed by the target processor p_j to execute the function z_i with inputs $b_{i,k}$, and the $CS(z_i, b_{i,k})$ is the number of executed C statements for the function z_i with inputs $b_{i,k}$, evaluated with static host profiling. The $J4CS(p_j)$ is a frequency distribution of assembly instructions, C statements and average power consumption, and it is represented by meaningful statistical values (e.g., min, arithmetic mean, standard deviation, median, max, quartiles, percentiles).

The reference approach to evaluate J4CS metric is shown in Fig. 2. The first step is the function selection, where all the reference functions, one at a time, are extracted from the benchmark. The next step involves the input generation, where a fixed random number of inputs for each considered function have been generated. Two parallel path evaluates the assembly instructions needed to execute the functions and the "host-based" executed C statements (i.e., they depends only on inputs and functions, not on target processing unit). Finally, it is possible to evaluate J4CS using Eq. (17) with an iterative approach.

As said in the introduction, a first clarification is due with respect to the concept of generic C statement. It could be generally intended as something that ends with a semicolon (other views are possible too, e.g. Table 6.1 in [34]) but, to avoid ambiguity, this work adopts an empirical approach: it refers to the way a common profiling tool as *GCov* [4] performs the C statements identification when profiling their execution. Another clarification is related to the fact that such a metric will be for sure influenced by the used compiler. Some ways to manage this issue could be: (1) to specify also the used compiler (possibly giving rise to a set of J4CS for each processor/compiler pair); (2) to report the average of the results obtained by using the most diffused compilers; (3) to report only the results related to the most diffused one. At this point, it is quite clear that J4CS, as defined above, will be influenced by several factors and that a J4CS-based estimation will be probably affected by relevant errors. However, these can be still acceptable by keeping in mind the following aspects: it is a straightforward way to have an off-the-shelf metric (i.e., by defining a standard benchmark it would possible to report its value on a processor datasheet like normally done for the MIPS metric); it can be applied to several SW processor technologies; it is intended to be used for very early performance analysis in SW domain. Anyway, as described in the next section, J4CS can be also characterized by a set of values related to *Min, Max, Average*, and *Standard Deviation* (i.e., by statistical distribution parameters). In this way, it is possible to perform different analysis depending on the final goal.

4. Evaluation of J4CS

4.1. Generic framework

In order to evaluate the metric for a given target processor, it is needed, at least, to: define a set of relevant C functions to be used as benchmark (ideally a standard benchmark to be used for all the processors) [3]; identify a way to stimulate (i.e., to execute) it by means of relevant input data sets; identify a tool to perform statement-level profiling in order to count the number of executed C statements for each input; identify tools to compile the C function for the target processor and to simulate its execution in order to obtain total number of executed assembly instructions, as shown in Fig. 3.

It is worth noting that only the last step must be applied for each different processors that have to be characterized. Indeed, the others are independent. Moreover, J4CS is an one-time effort since this metric, once evaluated, is available "for free" for any estimation activity. So, to support J4CS evaluation, a proper framework [3] has been adapted. Additionally, such a framework is also able to evaluate statistics on the metric itself.

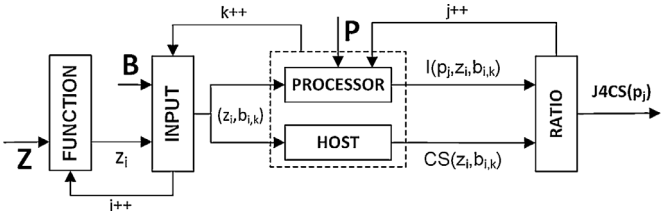


Fig. 2. J4CS evaluation approach.

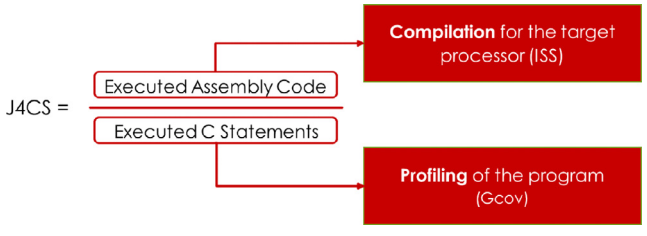


Fig. 3. J4CS evaluation methodology.

The overall flow of the J4CS evaluation can be summarized into three phases.

- Energy Data Acquisition: collect energy information useful to evaluate J4CS metrics (i.e., assembly line executed, C statement executed).
- Energy Model Characterization: calculate the parameter related to energy consumption [5]
- Energy Estimation: execute profiling and substitute this results into the energy model to calculate the total energy consumption.

The advantage of this approach relies on a statistical analysis able to extract as-much-as-possible energy/power pattern behavior, without an exhaustive instruction-level energy estimation for software that often need to perform deep architectural software analysis (i.e., control-flow analysis, loop bound analysis, path analysis).

It is possible, for instance, to consider the real number of executed assembly line, and correlate them with the number of executed C statement. The higher the correlation between these two parameters, the higher the accuracy in power consumption estimation inside a specific fixed interval. The whole framework is shown in Fig. 4. The following paragraphs describe the main features of this generic framework, meanwhile processor specific features are described later.

4.1.1. Reference benchmark

A simple benchmark composed of 14 well-known functions (i.e., C leaf functions z_i in Eq. (17)) has been realized. The functions of the benchmark are the following ones:

- *Quicksort*: the sorting algorithm that follows the divide et impera approach. The algorithm recursively divides the input array until many small 1-length arrays was obtained. An array and its length have been passed as parameters.
- *Mergesort*: a sorting algorithm that follows the divide et impera approach. The array is recursively split until the sub-lists are composed by a single element. Then, two adjacent sub-lists are compared and merged sorting the inner elements.
- *Matrix Multiplication*: an algorithm that multiplies rows by columns of two-dimensional array.
- *Kruskal*: it is used to find the minimum spanning tree of a non-oriented graph that does not contains negative edges. It is a greedy algorithm and, in this case, the greedy choice consists in taking always the edge with minimum cost between among those available.
- *Floyd-Warshall*: it calculates the distances between all pairs of vertices of a weighed graph with no negative loops. The costs of the edges may be negative values as long as these are not part of a negative loop.
- *Dijkstra*: it calculates the minimum paths from a starting node x towards all nodes accessible by x. The graph must be oriented, can contain loops and must have edges with positive costs. This algorithm uses the concept of relaxation in order to obtain distances.
- *Breadth First Search and Depth First Search*: two algorithms for traversing a graph. In the first function, the nodes that must be visited are inserted in a queue, while in the second one in a stack.
- *Banker's Algorithm*: it is used in the operating systems to avoid deadlock situations during the allocation of resources to a process.
- *A**: a graph-searching algorithm that identifies a path from an initial node x to a final node y. It is similar to the Dijkstra algorithm that for each node takes into account all possible directions and then chooses the one with lower cost. Instead, A* avoids to visit all edges connected to a node using a heuristic function that estimates the cost to the destination node.
- *Bubble Sort*: Sorting algorithm that repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order.
- *Selection Sort*: it divides the input list into two parts, the subset of items already sorted, and the subset of items remaining to be sorted that occupy the rest of the array.
- *Insertion Sort*: it builds the final sorted array one item at a time.
- *Greatest Common Divisor (GCD)*: the classical greatest common divisor algorithm.

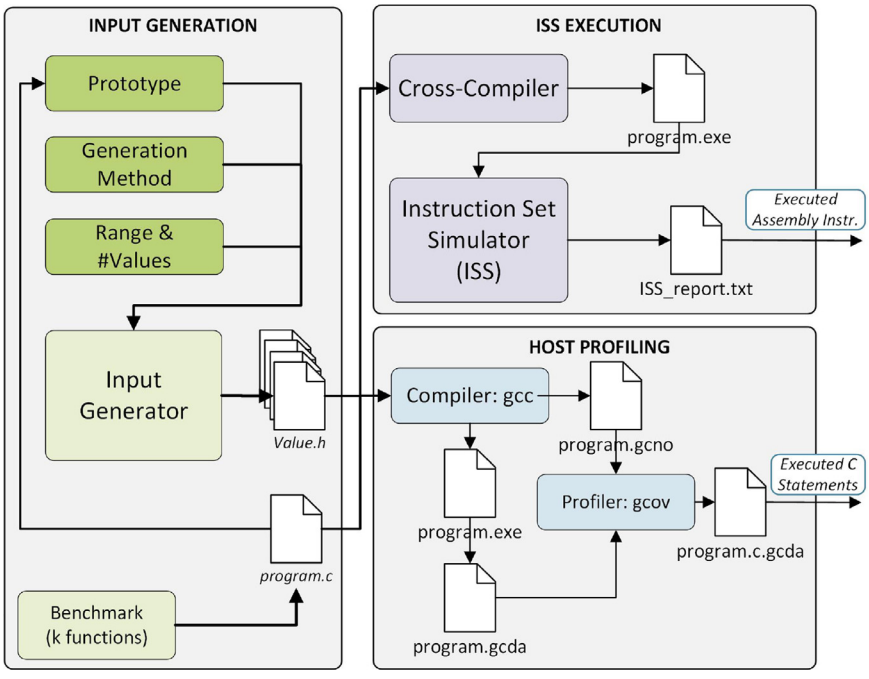


Fig. 4. J4CS evaluation framework.

Table 2
Benchmark functions characterization.

Functions	Decision Point	Global Variables	Loop	GOTO	Assignment	Exit Point	Total Operators	Distinct Operators	Total Operands	Distinct Operands
A*	19	13	7	3	39	10	372	41	205	34
Banker's Algorithm	6	12	4	1	20	4	247	33	119	44
Bellman Ford	14	7	7	2	28	5	352	34	168	52
Binary Search	6	6	3	1	19	4	195	35	75	25
Bubble Sort	4	5	2	0	17	4	243	29	105	57
Dijkstra	15	12	6	2	35	6	331	39	161	35
Floyd-Warshall	4	5	3	0	11	3	266	26	140	88
GCD	6	6	2	1	13	4	164	25	55	21
Insertion sort	3	7	2	0	12	3	164	30	59	20
Kruskal	23	14	12	1	46	11	394	34	194	38
Matrix Multiplication	4	6	3	0	16	3	207	33	80	35
Mergesort	10	6	6	0	41	5	271	36	133	45
Quicksort	6	5	4	0	27	5	238	36	118	33
Selection Sort	3	7	2	0	14	4	163	28	62	22

- *Binary Search*: it finds the position of a target value within a sorted array.
- *Bellman Ford*: it computes shortest paths from a single source vertex to all of the other vertices in a weighted graph.

The source-code is available on [35]. Tables 2 and 3 summarizes the main features of the whole benchmark. Functions are characterized by decision points (i.e., control flow statements), loops, GOTO (i.e., unconditional jumps), variables assignments, functions exit points, operators and operands (total and distinct). Moreover, all the functions are leaf ones. Functions source-code is characterized by source lines of code (SLOC), data types, inputs types (scalar or vector) and their values. SLOC do not include comment lines or empty lines. The scalar values range have been chosen related to 8051 internal memory size (128 KB), to prevent buffer overflows.

4.1.2. Inputs generation

A module that (semi)automatically generates inputs for the benchmark functions has been used in order to evaluate J4CS (i.e., $b_{i,k}$ in Eq. (17)). In particular, for each function they have been randomly generated 1000 input data sets. Moreover, for each function, different data types have been considered (i.e., *int8, int16, int32,* and *float*) to analyze the results with respect to the internal architecture of the considered processor. Each input data set is then stored in a header file to be included in the function at compile time.

The module needs to know what variables the function requires. For this purpose, the user must define a prototype of the implemented function. This prototype contains the function name and the name and type of each input variable. The input generator parses the prototype file to find its name and to find out proper data for the function. For each variable, the user is asked to insert a values range (as shown in Table 3) and then the module will generate the number of values accordingly. With a function that requires more then one variable, it performs the Cartesian product of generated values. For each combination obtained, it creates a header file that contains the values of a single combination. Finally, the module creates a directory that contains every header file.

4.1.3. Profiling on the host architecture (Host Profiling)

After the inputs generation phase, a tool to count the number of executed C statements is needed (i.e., $CS(z_i, b_{i,k})$ in Eq. (17))). This value is obtained by performing a profiling of the benchmark functions by means of the *GCov* [4] profiler for each generated input. The functions have been compiled with GCC, using *-fprofile-arcs* and *-ftest-coverage* compilation flags. These flags tell the compiler to generate other information needed by *GCov* in order to make correct profiling. The first flag allows the generation of a *.gcda* file that has more information for each branch of the program, while the second one adds information to count the number of times a statement has been executed. The compilation will trigger the creation of a *.gcno* file and generate the corresponding *.gcda* file. To complete the task, the *GCov* command has been executed. The profiling will be done correctly only if the above-described files were generated and reachable.

The total number of executed C statements for each function is simply the sum of the single profiling numbers associated with each statement. It is worth noting that such profiling is performed one-time on the host platform since it is independent of the target processor.

4.1.4. Profiling on the target processor (ISS Execution)

The last data needed to calculate the J4CS metric is the number of assembly instructions executed by the target processor for each function and input set in the benchmark (i.e., $I(p_j, z_i, b_{i,k})$ in Eq. (17)). So, for each target processor there is the need for an *Instruction Set Simulator* (ISS).

4.2. Processor specific framework: two examples

J4CS has been evaluated by considering some specific processors (i.e., p_j in Eq. (17)). In this work, as done in [3], two processors have been analyzed: the Cobham Gaisler LEON3 micro-processor [36], and the Intel 8051 micro-controller [37].

4.2.1. LEON3 micro-processor

LEON3 [36] is a 32-bit synthesizable soft-processor that is compatible with SPARC V8 architecture: it has a seven-stage pipeline and Harvard architecture, uses separate instruction and data caches and supports multiprocessor configurations. It represents a soft-processor for aerospace applications. Cobham Gaisler offers TSIM System Emulator [38] as an accurate emulator of LEON3 processors. A free evaluation version of TSIM/LEON3 is available on Gobham website [38], but it does not support code coverage, configuration of caches, memories and so on. Anyway, it has been chosen as the reference ISS for first analysis since it provides the information needed to evaluate J4CS. The LEON3 version has a default simulated system clock of 50 MHz. The evaluation version of TSIM/LEON3 implements 2*4 KiB caches (not removable), with 16 bytes per line with Least-Recently-Used (LRU) replacement algorithm. It has 8 register windows, a RAM size of 4096 KiB and a ROM size of 2048 KiB. By default, TSIM/LEON3 emulates the FPU. Benchmark functions have been compiled, with the Bare-C Cross-Compiler (BCC) for LEON3 processors [39]. It is based on the GNU compiler tools and the New-lib standalone

Table 3
Source-level characteristics.

Functions	SLOC	Data Type	Scalar Inputs	Range Scalar Values	Array Inputs	Range Array Values
A*	105	int8	s,	s ∈ [2, 9], g ∈ [0, 8]	a[s][s]	[−128,127]
		int	g	s ∈ [2, 6], g ∈ [0, 5]		[−32768,32767]
		long		s ∈ [2, 3], g ∈ [0, 2]		[−2147483648,2147483647]
		float		s ∈ [2, 3], g ∈ [0, 2]		[−2147483648.0,2147483647.0]
Banker's Algorithm	46	int8	nr,	nr ∈ [1, 5], np ∈ [1, 5]	available[nr]	[0,255]
		int	np	nr ∈ [1, 3], np ∈ [1, 3]	allocated[np][nr]	[0,65535]
		long		nr ∈ [1, 2], np ∈ [1, 2]	max[np][nr]	[0,4294967295]
		float		nr ∈ [1, 2], np ∈ [1, 2]		[0.0,4294967295.0]
Bellman Ford	75	int8	s	s ∈ [2, 10]	a[s][s]	[−128,127]
		int		s ∈ [2, 7]		[−32768,32767]
		long		s ∈ [2, 4]		[−2147483648,2147483647]
		float		s ∈ [2, 4]		[−2147483648.0,2147483647.0]
Binary search	44	int8	n	n ∈ [2, 116]	a[n]	[−128,127]
		int		n ∈ [2, 54]		[−32768,32767]
		long		n ∈ [2, 23]		[−2147483648,2147483647]
		float		n ∈ [2, 23]		[−2147483648.0,2147483647.0]
Bubble Sort	35	int8	n	n ∈ [1, 116]	a[n]	[−128,127]
		int		n ∈ [1, 54]		[−32768,32767]
		long		n ∈ [1, 23]		[−2147483648,2147483647]
		float		n ∈ [1, 23]		[−2147483648.0,2147483647.0]
Dijkstra	82	int8	s	s ∈ [2, 9]	a[s][s]	[−128,127]
		int		s ∈ [2, 5]		[−32768,32767]
		long		s ∈ [2, 3]		[−2147483648,2147483647]
		float		s ∈ [2, 3]		[−2147483648.0,2147483647.0]
Floyd-Warshall	29	int8	s	s ∈ [1, 10]	a[s][s]	[0,255]
		int		s ∈ [1, 7]		[0,65535]
		long		s ∈ [1, 5]		[0,4294967295]
		float		s ∈ [1, 5]		[0.0,4294967295.0]
GCD	32	int8	n,	n ∈ [2, 120], m ∈ [2, 120]	-	-
		int	m	n ∈ [2, 32768], m ∈ [2, 32768]		
		long		n ∈ [2, 2147483647], m ∈ [2, 2147483647]		
		float		n ∈ [2, 2147483647], m ∈ [2, 2147483647]		
Insertion Sort	35	int8	n	n ∈ [2, 116]	a[n]	[−128,127]
		int		n ∈ [2, 54]		[−32768,32767]
		long		n ∈ [2, 23]		[−2147483648,2147483647]
		float		n ∈ [2, 23]		[−2147483648.0,2147483647.0]
Kruskal	129	int8	s	s ∈ [2, 10]	a[s][s]	[−128,127]
		int		s ∈ [2, 6]		[−32768,32767]
		long		s ∈ [2, 4]		[−2147483648,2147483647]
		float		s ∈ [2, 3]		[−2147483648.0,2147483647.0]
Matrix Multiplication	34	int8		[1,6]		[−128,127]
		int	rowA, colA	[1,4]	a[rowA][colA]	[−32768,32767]
		long	rowB, colB	[1,2]	b[rowB][colB]	[−2147483648,2147483647]
		float		[1,2]		[−2147483648.0,2147483647.0]
Mergesort	84	int8	n	n ∈ [1, 57]	a[n]	[−128,127]
		int		n ∈ [1, 25]		[−32768,32767]
		long		n ∈ [1, 11]		[−2147483648,2147483647]
		float		n ∈ [1, 6]		[−2147483648.0,2147483647.0]
Quicksort	55	int8	n	n ∈ [1, 36]	a[n]	[−128,127]
		int		n ∈ [1, 17]		[−32768,32767]
		long		n ∈ [1, 6]		[−2147483648,2147483647]
		float		n ∈ [1, 5]		[−2147483648.0,2147483647.0]
Selection Sort	29	int8	n	n ∈ [2, 116]	a[n]	[0,255]
		int		n ∈ [2, 54]		[0,65535]
		long		n ∈ [2, 23]		[0,4294967295]
		float		n ∈ [2, 23]		[0.0,4294967295.0]

C-library. BCC is composed by GNU GCC C/C++ compiler 4.4.2, GNU Binutils 2.19.51 and Newlib C-library 1.13.1. After the simulation, the framework is ready to calculate the metric and some statistics by considering all the inputs used to stimulate the functions.

4.2.2. LEON3 Statistical analysis

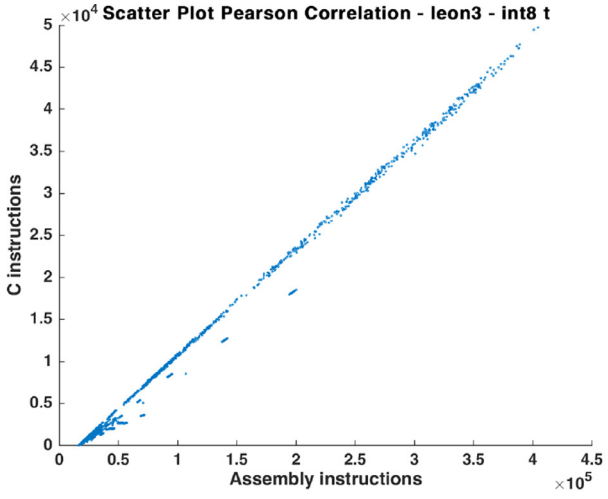
Table 4 shows the Pearson correlation and slope values between executed assembly instructions and executed C statements. It is possible to note that the correlation is close to 1, while the slope indicates that the estimation uncertainty depends on the number of data input bits. Furthermore, some specific functions are more sensitive to data types compared to other ones. These last results depend on functions implementation (number of branches, loop, and complex arithmetic operations).

Fig. 5 shows the scatter plot related to the Pearson correlation. The plots show a strict correlation between C statements and Assembly instructions, due to the Sparc-V8 RISC ISA architecture, as reported in [5]. Regarding to the other points (the ones under the principal linear regression line), the deviation depends on different data types and functions implementations that introduce different behaviors compared to the prominent distribution. These points contribute to the introduction of errors inside the energy estimation activity, and are dependent on the LEON3 processor internal micro-architecture (i.e., 32 bit architecture, pipeline, number of registers).

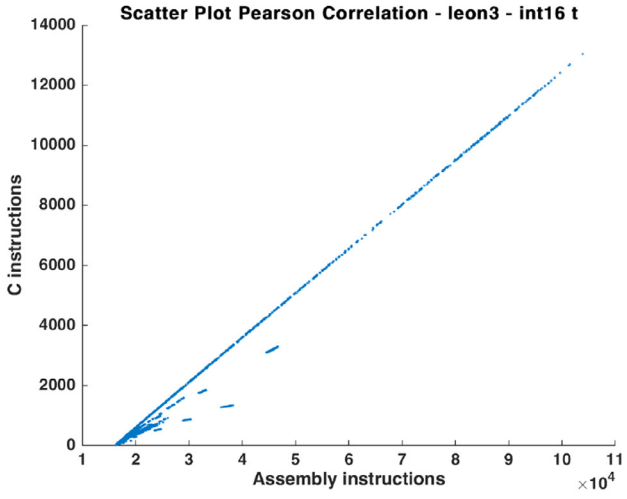
Table 4
LEON3 statistical analysis results.

	Data Type Corr.[1]					Data Type Slope[3]				
Function	int8	int16	int32	float	Corr. Tot.[2]	int8	int16	int32	float	Slope Tot.[4]
A*	0.997130176	0.995528883	0.997342958	0.999199314	0.997558728	0.0890	0.0854	0.0818	0.0698	0.0878
Banker's Algorithm	0.976985991	0.971856389	0.973122344	0.983442876	0.979447788	0.0730	0.0736	0.0774	0.0729	0.0738
Bellman Ford	0.996911126	0.994174207	0.999111052	0.999758625	0.993600677	0.0694	0.0943	0.1162	0.0733	0.0697
Binary Search	0.999249964	0.998686622	0.997520361	0.993549505	0.904750109	0.1545	0.2294	0.2226	0.1692	0.1529
Bubble Sort	0.999886591	0.99998281	0.999912836	0.999845827	0.999198011	0.1266	0.1484	0.1517	0.1128	0.1270
Dijkstra	0.999239588	0.998580215	0.99865452	0.999434198	0.998348213	0.0892	0.0947	0.0879	0.0798	0.0908
Floyd Warshall	0.997867119	0.999069696	0.994619228	0.997353347	0.997684756	0.0659	0.0605	0.0693	0.0642	0.0661
GCD	0.999986897	0.997729568	0.999092395	-	0.958592379	0.1534	0.2003	0.1981	-	0.1415
Insertion Sort	0.999899431	0.999667128	0.998299365	0.99743627	0.998106672	0.1075	0.1397	0.1496	0.1177	0.1083
Kruskal	0.999692859	0.999778195	0.999659212	0.999766269	0.999700213	0.1007	0.1068	0.1151	0.0970	0.1013
Matrix Mult.	0.991026869	0.987504129	0.986854719	0.99465194	0.990567396	0.0570	0.0510	0.0714	0.1265	0.0584
Merge Sort	0.999806391	0.999832814	0.999806429	0.999840959	0.994096345	0.0957	0.1275	0.1225	0.0886	0.0979
Quick Sort	0.999577773	0.999652397	0.999680258	0.999595041	0.999137649	0.1174	0.1279	0.1321	0.1099	0.1192
Selection Sort	0.999924321	0.999882252	0.999821454	0.999667636	0.997105498	0.1151	0.1560	0.1675	0.1287	0.1161
Tot.[5]	0.99361	0.98178	0.96865	0.98326	0.992487927	0.1218	0.1391	0.1289	0.1049	0.1215

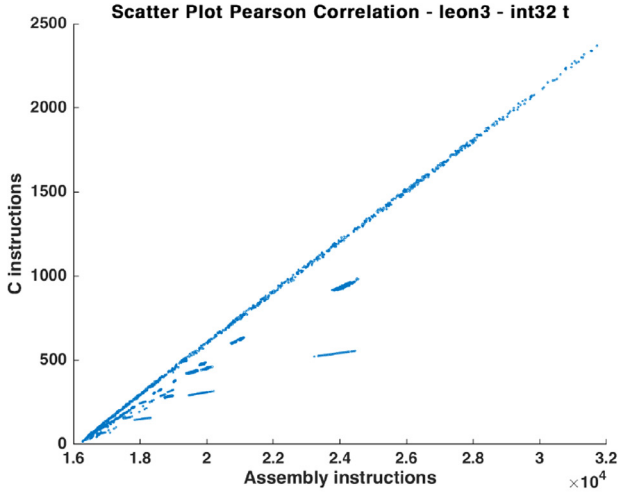
[1] Corr.: Pearson Correlation; [2] Corr. Tot.: Total Pearson Correlation considering all data types; [3] Slope: Regression Slope Parameter; [4] Slope Tot.: Total Regression Slope considering all data types; [4] Tot.: Total data set (considering all functions)



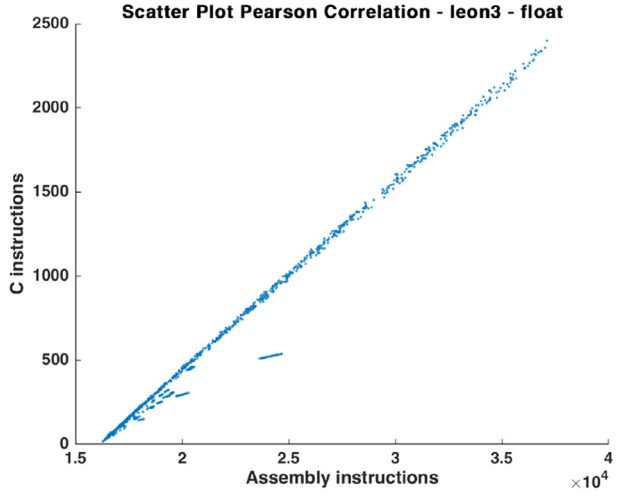
(a) LEON3 int8 Corr.



(b) LEON3 int16 Corr.



(c) LEON3 int32 Corr.



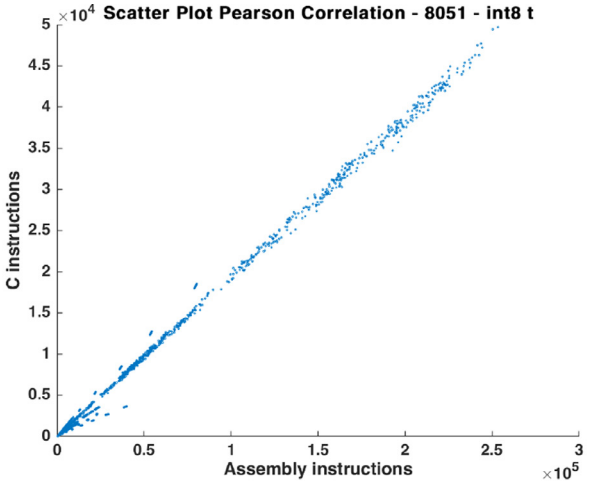
(d) LEON3 float Corr.

Fig. 5. Pearson correlation plot for LEON3.

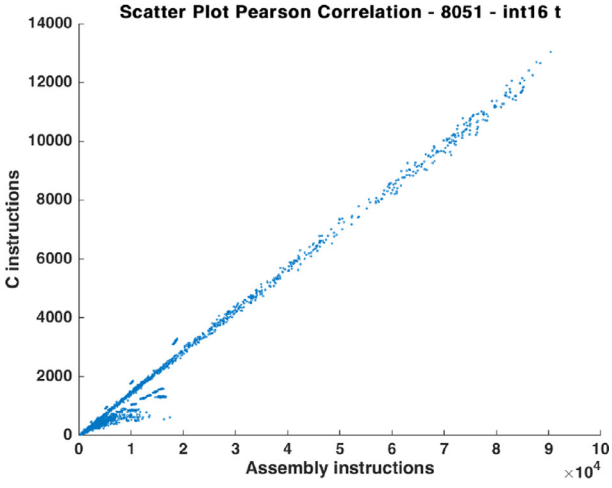
Table 5
8051 Statistical analysis results.

	Data Type Corr.[1]					Data Type Slope[3]				
Function	int8	int16	int32	float	Corr. Tot.[2]	int8	int16	int32	float	Slope Tot.[4]
A*	0.999018373	0.84914453	0.998332704	0.999596943	0.830460356	0.1955	0.0582	0.1048	0.0546	0.1376
Banker's Algorithm	0.966167708	0.961797134	0.950479804	0.971077972	0.340041773	0.1596	0.1060	0.0634	0.0201	0.0106
Bellman Ford	0.99794459	0.99543516	0.948538448	0.999616672	0.852702454	0.1098	0.0633	0.0363	0.0222	0.0976
Binary Search	0.999560803	0.999687367	0.999743356	0.985202744	−0.08002339	0.2128	0.1353	0.0783	0.0183	−0.002
Bubble Sort	0.999655934	0.999585159	0.998940075	0.596901958	0.974990261	0.1923	0.1405	0.0814	0.0215	0.1905
Dijkstra	0.99971141	0.9995696	0.998715493	0.998811434	0.539663795	0.1714	0.1316	0.0901	0.0295	0.0759
Floyd Warshall	0.999881657	0.999075444	0.999106673	0.99911282	0.810045772	0.0913	0.0827	0.0506	0.0171	0.0632
GCD	0.890585301	0.899892026	0.952382673	-	0.68826302	0.1734	0.1601	0.1212	-	0.1466
Insertion Sort	0.999950396	0.999902786	0.999545234	0.994771407	0.973606759	0.2592	0.1414	0.0759	0.0925	0.2503
Kruskal	0.999856993	0.999892487	0.999837653	0.999267543	0.991654051	0.2277	0.1709	0.1255	0.0533	0.2334
Matrix Mult.	0.99952095	0.995791004	0.987974866	0.956583715	0.869857949	0.1322	0.0619	0.0398	0.0206	0.1039
Merge Sort	0.999623696	0.999613733	0.999572569	0.9993614	0.71989699	0.1485	0.1001	0.0656	0.0122	0.0979
Quick Sort	0.999646522	0.999695728	0.999281295	0.997228213	0.905400216	0.1354	0.0919	0.0568	0.0242	0.1249
Selection Sort	0.999717704	0.999385492	0.998837314	0.99584793	0.966448824	0.2041	0.1098	0.0591	0.1083	0.1927
Tot.[5]	0.99293	0.98799	0.92563	0.74391	0.960083884	0.1937	0.1384	0.0732	0.0245	0.1783

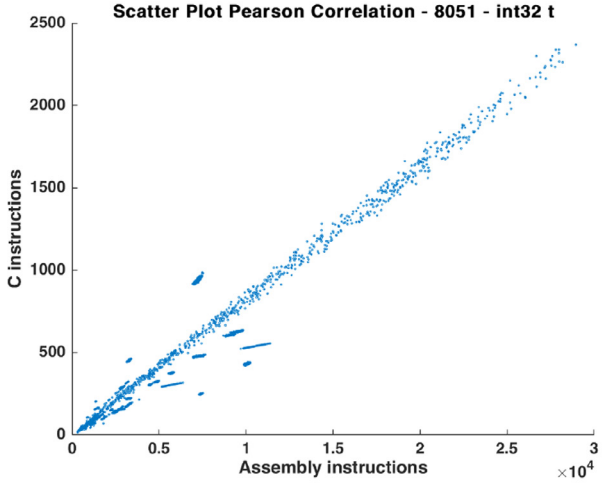
[1] Corr.: Pearson Correlation; [2] Corr. Tot.: Total Pearson Correlation considering all data types; [3] Slope: Regression Slope Parameter; [4] Slope Tot.: Total Regression Slope considering all data types; [4] Tot.: Total data set (considering all functions)



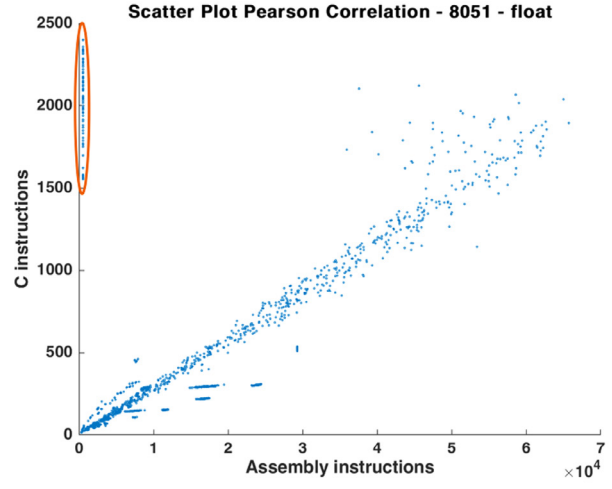
(a) 8051 int8 Corr.



(b) 8051 int16 Corr.

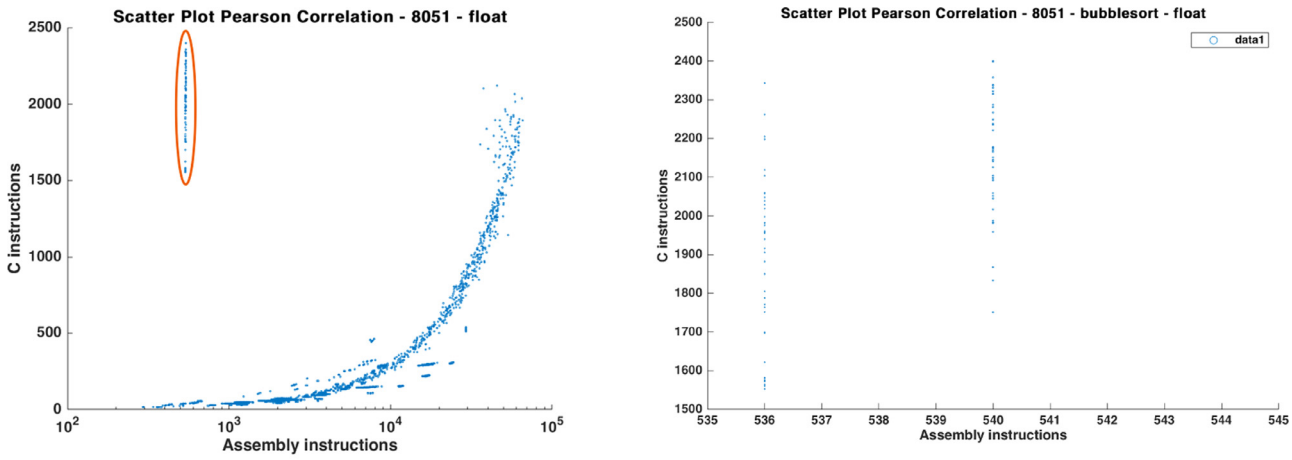


(c) 8051 int32 Corr.



(d) 8051 float Corr.

Fig. 6. Pearson correlation plot for 8051.



(a) 8051 Float Corr. (Log Scale) (b) 8051 Bubblesort Float Corr.

Fig. 7. Pearson correlation plot for 8051.

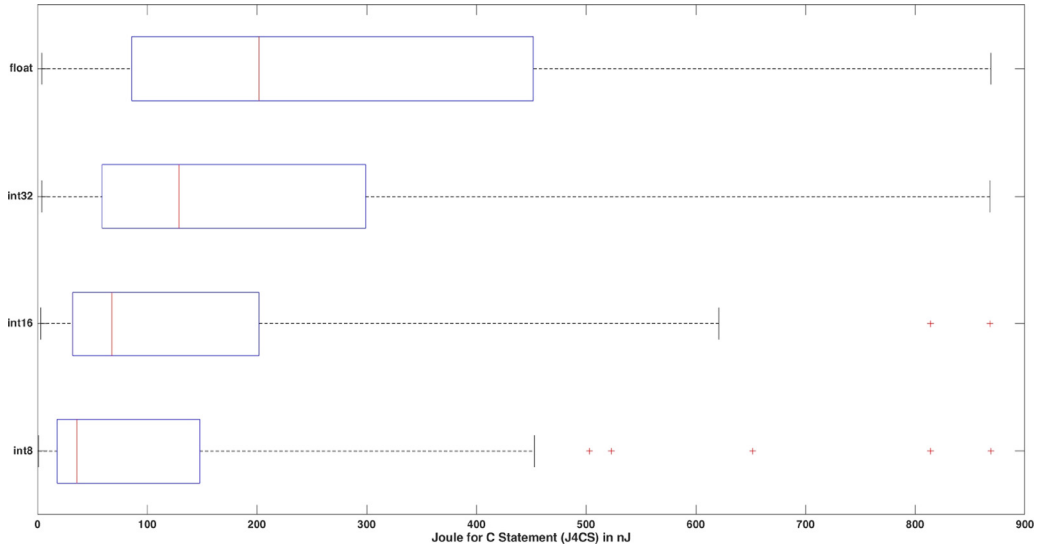


Fig. 8. J4CS BoxPlot results for LEON3 (LEON3FT-RTAX board).

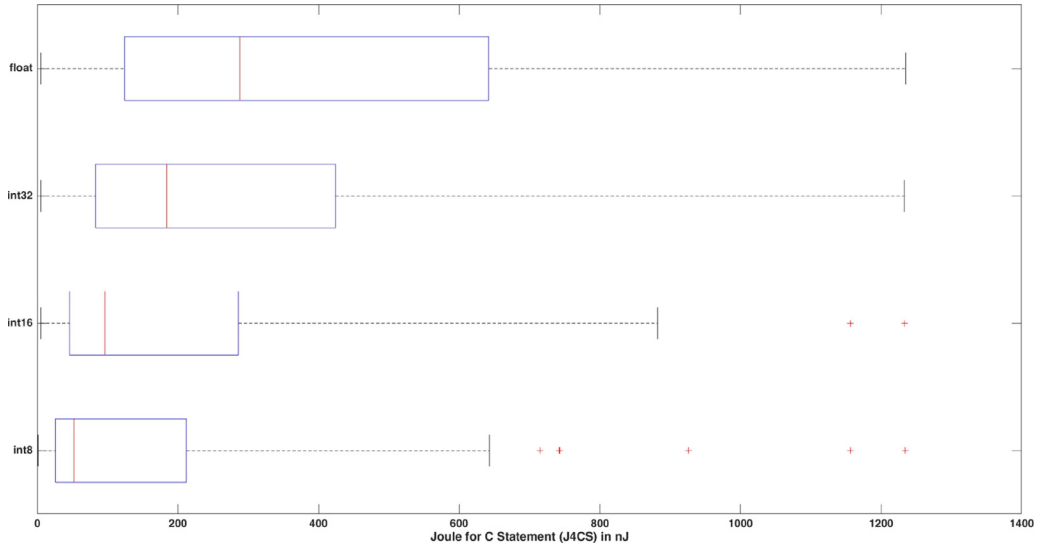


Fig. 9. J4CS BoxPlot results for LEON3 (UT699 board).

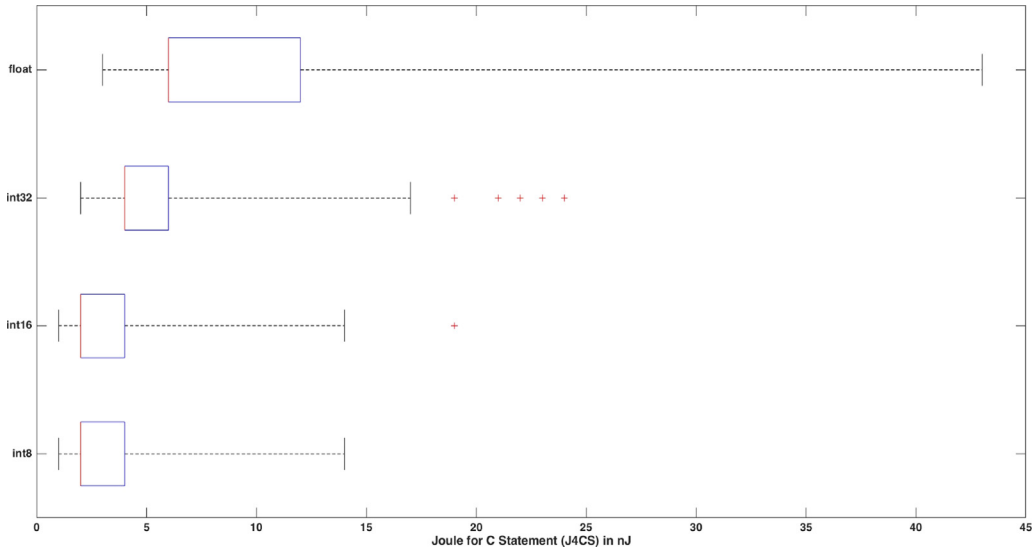


Fig. 10. J4CS BoxPlot results for 8051 (AT89C51 board).

Table 6
Board characteristics.

Parameters	RTAX	UT699	AT89C51
Clock	25 MHz	66 MHz	12 MHz
V_{dd}	3,3 V	3,6 V	6,0 V
$\bar{P}$	500 mW	178,2 mW	600 mW
MIPS	20	75	2
ϕ	0,025	0,002376	0,3

4.2.3. 8051 Micro-controller

The Intel 8051 micro-controller is built around an 8-bit CPU. Architectural model used is the Harvard Architecture, and therefore it partitions data and instruction by the use of two memories and two buses; indeed 8051 presents a PROM non-volatile memory which contains program instruction and a RAM memory for data. Furthermore, it presents an 8-bit Data Bus and a 16-bit Address Bus. I8051 registers are 8-bit registers. ALU works with 8-bit words and is provided with an accumulator register and communicates with four I/O 8-bit ports. The University of California has developed a project centered on 8051 microprocessor, which provides a number of tools useful for simulating C code on Intel 8051 microprocessor. The project name is Dalton and was developed by the Department of Computer Science of the University of California [40]. The Dalton Instruction Set Simulator (ISS) allows a user to simulate programs written for the 8051 and provides statistics on instructions executed, instructions executed per second, clock cycles required by the 8051, and average instructions per second for an 8051 executing the same program. For these characteristics, it has been chosen as the reference ISS for the measurement of the J4CS for 8051 microprocessor. The functions were compiled, with SDCC (Small Device C Compiler) [41]. SDCC is a free open source C compiler suite designed for 8 bit Microprocessors. The entire source-code for the compiler is distributed under GPL and has extensive language extensions suitable for utilizing various micro-controllers and underlying hardware. The Dalton ISS needs a.hex to do the simulation. This kind of file was generated with SDCC. In order to do a proper simulation, during the compilation two options was specified: *–mmcs51* and *–iram-size 128*. The first one refers to the family of the microprocessor while the second to the dimension of the internal RAM. The compilation generates an *.ihx* file that can be converted to *.hex* file using the *packihx* command. At the end, it is possible to execute the ISS. It generates a file that contains information about the simulation. After the simulation, the framework is ready to calculate the metric and some statistics on all input generated for the functions. These calculations are made with a program that returns two files containing metric values, for each input, and statistics on the sample.

Table 7
J4CS measured on LEON3FT-RTAX, UT699 and AT89C51 Board (in nJ).

Data Type	Min	Q_1^1	Median	Q_3^2	Max	AM[3]	SD[4]	Var[5]	GM[6]	85%[7]	95%[8]
LEON3FT-RTAX int8	1	17	36	148	869	100.73	137.03	18779	48.33	220	338
LEON3FT-RTAX int16	1	32	67	202	868	140.66	173.38	30061	75.33	312	523
LEON3FT-RTAX int32	4	58	129	299	868	213.11	208.60	43513	131.98	451	814
LEON3FT-RTAX float	4	86	202	452	869	270.45	240.16	57676	173.18	597	814
LEON3FT-RTAX AVG	5	48.25	108.5	241.5	869	181.24	189.79	37507	107.20	348.25	622.25
UT699 int8	1	25	51	211	1234	143.07	194.63	37882	68.549	312	481
UT699 int16	5	46	95	286	1233	199.79	246.29	60661	106.9	359	743
UT699 int32	5	83	184	424	1233	302.72	296.29	87789	187.45	564	1156
UT699 float	5	123	287	642	1235	384.22	341.13	1.1637e+05	246.1	743	1157
UT699 AVG	4	69.25	154.25	390.75	1233.75	257.45	269.58	68530.25	152.25	494.5	884.25
AT89C51 int8	1	2	2	3	14	2.2705	1.2716	1.6169	2.0124	3	5
AT89C51 int16	1	2	2	3	19	2.9666	1.413	1.9967	2.7245	4	6
AT89C51 int32	2	3	3	5	24	5.3924	5.2433	27.492	4.1399	8	23
AT89C51 float	3	5	6	11	43	9.8238	8.3489	69.704	7.7229	13	33
AT89C51 AVG	1.75	3	3.25	5.5	25	5.1133	4.0692	25.202	4.149925	7	16.75

[1]Q_1: First Quartile; [2]Q_3: Third Quartile; [3]AM: Arithmetic Mean; [4]SD: Standard Deviation; [5]Var: Variance; [6]GM: Geometric Mean; [7]85%: 85th Percentile; [8]95%: 95th Percentile;

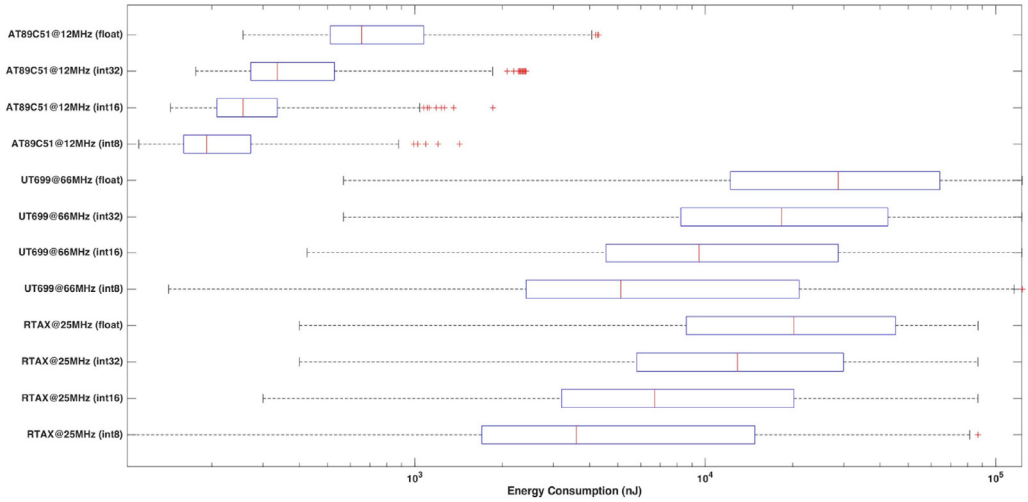


Fig. 11. J4CS-based SW comparison.

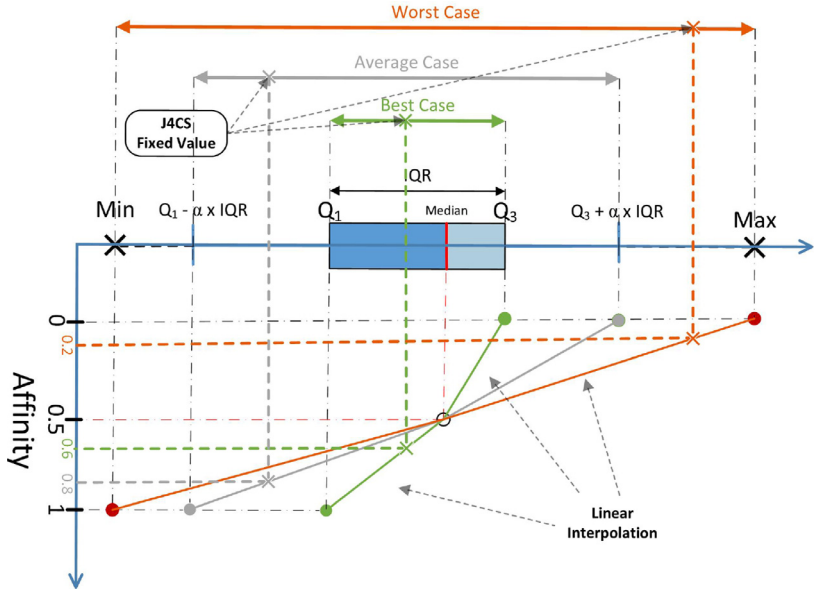


Fig. 12. J4CS-based refinement (considering affinity value).

4.2.4. 8051 Statistical analysis

Table 5 shows Pearson correlation and slope between executed assembly instructions and executed C statements in details. Differently from LEON3 results, it is possible to note that the correlation is not so close to 1 (there are also values that are under 0.9), while the slope indicates that the estimation uncertainty depends on the number of data input bits. Some specific functions are more sensitive to data types compared to other ones (behavior similar to LEON3 processor). This last results depends on different functions implementations (number of branches, loop, and complex arithmetic operations), and the fact that 8051 has an 8-bit internal architecture, and no Floating Point Unit.

The different correlation and slope values in Table 5 (and even negative for binary search algorithm) mean that the values are very sparse in the scatter-plot. In the case of a single function, putting together all the types of data, many straight lines for each data type have been founded, with different slopes and above all different weights (the number of points from which each line is composed), while the experimental data are arranged very well on the linear regression straight line for data types (i.e., int8, int16, int32, and float). The more the correlation is low or even negative, the more the lines by data type are open to each other, with different numbers of points and also not common intercepts for the *Y* axis of the graph. From Table 5 it is possible to note also that the worst values are associated to float data types, since the 8051 did not have Floating Point Unit.

Fig. 6 shows the scatter plot related to the Pearson correlation. Compared to LEON3 scenarios, the correlation point distributions is not so linear as the LEON3 scenarios. This sparse points distribution depends on 8051 internal processor architecture (8-bit 8051 CISC ISA). Another difference is the internal RAM that the 8051 (and Dalton ISS [40]) has compared to the external TSIM LEON3 RAM memory. This internal 128 KB RAM limits the data input ranges, and the possible test-bench simulation activities. Meanwhile, there is a similar points placing behaviors with respect to LEON3 scenarios (i.e., the isolated points under the linear regression line) that introduce errors in the estimation activities.

Furthermore, the Fig. 6d is the only plot that has a strange point cluster (orange circle). Fig. 7a shows the 8051 float correlation with *x*-axis in logarithmic scale. It is worth noting that there are some points outside the main regression line. These points are related to one function, the *Bubblesort*. Fig. 7b presents the correlation plot for the *bubblesort* function in more details. From the graph, it is possible to note that there are 2 fixed assembly instructions values corresponding to a different number of executed C statements. This behaviors is not normal, while in the other case (int8, int16 and int32) the points are close to the regression line and they do not follow a strange pattern. This problem is probably due to errors in the ISS compilation/execution so they have been deleted from the dataset and not considered.

4.2.5. Use case scenario

In order to evaluate results in a real scenario, three boards have been considered by taking voltage and frequency information (the power information can be found in the processor/board data-sheets): LEON3FT-RTAX [42], LEON3FT-UT699 Single-Core SOC [43] and AT89C51 ATMEL Development Board [44] (based on 8051 architecture). The processors parameters used to evaluate J4CS [36] are shown in Table 6.

It is worth noting that LEON3-FT is a System-On-Chip design based on LEON3FT core, and it has the same ISA of the classical LEON3 processor. Therefore, the number of assembly instructions executed by the ISS is the same for both processors since they rely on the same compiler. So, considering these characteristics, the average energy consumption associated to each executed assembly instruction is: ($\bar{E}_{RTAX} = 0,8$ *nJ/Instr.*, $\bar{E}_{UT699} = 11,364$ *nJ/Instr.* and $\bar{E}_{AT89C51} = 0,16$ *nJ/Instr.*). The obtained results for the executions of the benchmark functions are summarized in Table 7.

For each function, different data types have been considered (*int8, int16, int32,* and *float*). Indeed, both timing [3] and energy, especially their average values, change with respect to the dimension of data.

Figs. 8–10 show the distribution related to J4CS evaluated for RTAX, UT699 and AT89C51 boards, according to the reference benchmark. The described evaluation process of J4CS for the three boards has required a total of near 12 h on a standard workstation (Intel i7, 1.5 GHz, 16 GB RAM). However, as highlighted before, this is a one-time effort to make available J4CS for subsequent analysis (as shown in the next section).

5. J4CS-Based energy consumption estimation

The availability of J4CS is very useful for very fast early-stage estimation, comparison and selection. Indeed, by having available J4CS for different processors, with a single host-based profiling it is possible to estimate the energy consumption of a function of interest for the whole processors set, giving very fast preliminary comparison and selection activities. As an example, strating from a target function *tf()* and considering a specific *golden input* $\boldsymbol{x}$, by means of a host-based profiling (that takes less than a second on the same workstation described in the previous section), it is possible to count the number of executed C statements during the execution of tf($\mathbf{x}$) (e.g., 100). Then, as shown in Fig. 11 (the *x*-axis is in a logarithmic scale), it is straightforward to compare the whole processors set by multiplying 100 for the related J4CS. Depending on a possible energy consumption constraint it is then possible to select a specific processor or, at least, to reduce the set to few of them in order to be considered for further analyses.

6. J4CS-Based energy consumption estimation validation

In order to validate the proposed metric, an error estimation evaluation has been performed with respect to the benchmark.

Table 8
AT89C51 board relative error results (in %).

	Int8				int16				int32				float			
Function	[1]Q_1	Median	[2]AM	[3]Q_3	[1]Q_1	Median	[2]AM	[3]Q_3	[1]Q_1	Median	[2]AM	[3]Q_3	[1]Q_1	Median	[2]AM	[3]Q_3
A*	4.99	4.99	−7.83	−42.5	46.34	46.34	22.20	19.51	21.33	21.33	−41.32	−31.10	8.31	−10.01	−79.69	−101.69
Banker's Algorithm	31.01	31.01	21.70	−3.47	35.67	35.67	6.73	3.51	43.02	43.02	−2.36	5.04	40.99	29.19	−15.64	−29.80
Bellman Ford	14.22	14.22	2.64	−28.66	25.47	25.47	−8.06	−11.78	2.97	2.97	−74.32	−61.71	18.68	2.42	−59.37	−78.89
Binary Search	30.48	30.48	21.10	−4.26	24.87	24.87	−8.92	−12.68	35.35	35.35	−16.14	−7.74	83.55	80.26	67.77	63.82
Bubble Sort	−12.27	−12.27	−27.43	−68.41	6.53	6.53	−35.51	−40.19	1.77	1.77	−76.47	−63.70	23.31	7.98	−50.29	−68.69
Dijkstra	1.67	1.67	−11.60	−47.49	10.47	10.47	−29.80	−34.28	−5.10	−5.10	−88.84	−75.18	18.15	1.78	−60.42	−80.06
Floyd Warshall	53.49	53.49	47.21	30.23	61.88	61.88	44.73	42.83	67.41	67.41	41.44	45.68	50.41	40.50	2.81	−9.07
GCD	−55.38	−55.38	−76.36	−133.07	48.96	48.96	25.99	23.44	85.59	85.59	74.11	75.98	-	-	-	-
Insertion Sort	−40.74	−40.74	−59.74	−111.11	−5.90	−5.90	−53.56	−58.86	−11.25	−11.25	−99.88	−85.41	−26.54	−51.85	−148.03	−178.40
Kruskal	−19.51	−19.51	−35.65	−79.27	−12.11	−12.11	−62.56	−68.17	−42.31	−42.31	−155.69	−137.19	−37.12	−64.55	168.76	201.67
Matrix Mult.	38.20	38.20	29.86	7.31	45.34	45.34	20.75	18.02	48.16	48.16	6.86	13.60	29.78	5.74	−37.61	−54.46
Merge Sort	−10.49	−10.49	−25.40	−65.73	14.99	14.99	−23.26	−27.51	16.84	16.84	−49.39	−38.58	57.78	49.34	17.26	17.26
Quick Sort	−0.96	−0.96	−14.59	−51.45	20.05	20.05	−15.92	−19.91	11.88	11.88	−58.31	−46.85	20.28	4.34	−56.24	−75.37
Selection Sort	−51.17	−51.17	−71.57	−126.75	−19.33	−19.33	−73.03	−79.00	−33.97	−33.97	−140.71	−123.29	−35.544	−62.65	−165.66	−198.19
Tot. Rel.[4]	1.17	1.17	−14.83	−51.76	21.65	21.65	13.58	17.50	17.16	17.16	−48.64	−38.86	19.42	2.49	−32.02	−45.51
Tot. Abs.[5]	25.90	25.90	32.33	57.12	26.99	26.99	30.78	32.83	30.49	30.49	66.13	56.96	34.64	31.58	71.50	89.02

[1]Q_1: First Quartile; [2]AM: Arithmetic Mean; [3]Q_3: Third Quartile; [4] Tot. Rel: Total Relative Error (all functions); [5] Tot. Abs.: Total Absolute Error (all functions)

Table 8 shows some results related to AT89C51 board. It is worth noting that the error depends on the specific J4CS considered (in this example we considers the first quartile, the arithmetic mean, the median and the third quartile). Errors ranges are highly variable, where median errors are less than other ones, depending on assembly and C statements distributions. In order to reduce errors and variance associated to the estimations, a further assumption can be considered.

Figs. 8 –10 present the specific processor characterization w.r.t. different boards. It is possible to fix the J4CS value introducing the concept of "Affinity" defined in [45]. Since the execution time of different functions depends on some architectural features of specific executors classes, this dependency can be defined using the "Affinity" metric, which suggests the most suitable processor class for the execution of a given functionality. This value, in the range of 0 and 1, provides a quantification of the matching between the structural and functional features of the functionality implemented by the considered processor classes function and the architectural features.

Starting from the affinity value, it is possible to refine the J4CS definition using the following equations:

Definition 6.1. *J4CS-A (Joule for C Statements considering Affinity metric).* Considering a single C function z_i, with a specific affinity value A_{ij} evaluated with the method proposed in [45], and the $J4CS(p_j)$ distribution evaluated for a specific processor, as presented in Eq. (17), it is possible to chose a fixed value for $J4CS-A(p_j, A_{i,j})$ depending on an "Affinity" value and distribution parameters [{Min, Q_1, Med (Median), Q_3, Max} of $J4CS(p_j)$ from Table 7]. Three different scenarios are considered:

1. Best Case

$$\begin{aligned} J4CS-A(p_j,A_{i,j}) &= 2\cdot(Med-Q_3)\cdot A_{i,j}+Q_3 && If\ A_{i,j}<0.5\\ J4CS-A(p_j,A_{i,j}) &= Med+2\cdot(A_{i,j}-0.5)\cdot(Q_1-Med) && If\ A_{i,j}\geq 0.5 \end{aligned} \tag{18}$$

2. Average Case

$$\begin{aligned} J4CS-A(p_j,A_{i,j}) &= 2\cdot(Med-[Q_3+\alpha\cdot IQR])\cdot A_{i,j}+(Q_3+\alpha\cdot IQR) && If\ A_{i,j}<0.5\\ J4CS-A(p_j,A_{i,j}) &= Med+(A_{i,j}-0.5)(Q_1-\alpha\cdot IQR)-2\cdot Med && If\ A_{i,j}\geq 0.5 \end{aligned} \tag{19}$$

3. Worst Case

$$\begin{aligned} J4CS-A(p_j,A_{i,j}) &= 2\cdot(Med\ -\ Max)\cdot A_{i,j}+Q_4 && If\ A_{i,j}<0.5\\ J4CS-A(p_j,A_{i,j}) &= Med\ +\ 2\cdot(A_{i,j}-0.5)(Min-Med) && If\ A_{i,j}\geq 0.5 \end{aligned} \tag{20}$$

Eq. (18) –(20) are derived considering a linear interpolation between affinity value and J4CS distribution. Fig. 12 shows the graphical representation of the equations above.

Table 9 shows the relative and absolute errors associated to the AT89C51 board where the affinity value has been introduced in order to reduce the estimation error. The total relative and absolute mean errors (considering all the functions in the benchmark and a test-bench composed of 100 inputs and executions for each functions) associated to the validation activity is in the range of $\{+/-0\ldots15\}$, with an highest error equal to about 34%, and the error reduction (absolute and relative) compared to Table 8 is in the range of $\{50\ldots110\%\}$. The only worst situation is the float case, where the error range is $\{+/-0.5\ldots18\%\}$. This is an interesting result since the estimation activity takes only few seconds (the time to profile the functions with the different inputs), without execute the specific functions on the reference target.

Finally, Fig. 13 shows the relative error results graph w.r.t. *Insertion Sort* function. In this case the errors are less then 10%. The other functions have a similar behavioral pattern, with some functions that arrive to relative errors less then 15% at most.

Table 9
AT89C51 board Error Results (in %) considering Affinity values.

	Int8			int16			int32			float		
Function	[1] Aff.	[2]Rel.	[3]Abs.	[1] Aff.	[2]Rel.	[3]Abs.	[1] Aff.	[2]Rel.	[3]Abs.	[1] Aff.	[2]Rel.	[3]Abs.
A*	0.5	4.9955	11.7152	0.45	6.7401	38.1508	0.48	0.6858	30.3601	0.441	−0.4464	7.4036
Banker's Algorithm	0.015	−2.4440	9.9131	0.47	2.8766	9.0229	0.445	−0.8453	9.1918	0.38	5.1252	13.2049
Bellman Ford	0.4	5.6460	14.5508	0.48	0.1357	17.6572	0.49	−10.6119	18.3243	0.425	−0.0185	24.3389
Binary Search	0.45	−11.2165	19.7001	0.48	−0.6609	5.5448	0.45	−9.8993	11.1606	0.1	34.88	40.07
Bubble Sort	0.6	−1.0513	2.5975	0.5	6.5398	7.4153	0.5	1.7746	2.9615	0.425	5.6814	7.4037
Dijkstra	0.5	1.6725	8.8011	0.5	10.4797	14.5653	0.56	−0.9043	9.2400	0.42	−0.6719	4.5762
Floyd Warshall	0.4	−2.3208	16.5335	0.4	−2.9037	17.2394	0.35	−1.0251	14.7806	0.31	−8.8814	16.6744
GCD	0.85	−1.0007	1.9384	0.45	5.5825	6.0357	0.02	−11.2415	14.7339	-	-	-
Insertion Sort	0.8	1.4812	2.5001	0.6	4.6832	4.8414	0.65	0.1265	1.9446	0.48	2.8127	6.7120
Kruskal	0.68	1.9949	14.7680	0.6	−0.9022	8.0750	0.9	−4.3662	6.8132	0.48	−5.3124	9.2739
Matrix Mult.	0.4	15.9635	21.0259	0.45	−1.1063	24.6616	0.45	11.8742	23.0171	0.4	−1.1025	15.5807
Merge Sort	0.6	0.5581	4.3580	0.5	14.9922	14.9922	0.5	16.8476	16.8476	0.3	3.7551	4.4141
Quick Sort	0.5	0.5581	4.3580	0.48	−7.1272	7.1272	0.5	11.8866	11.8866	0.4	−14.7891	14.7891
Selection Sort	0.8	−5.8196	6.1361	0.7	4.5302	4.5302	0.9	1.7502	1.7502	0.5	18.6732	19.1008
Tot. [4]	-	0.6441	9.9211	-	3.1328	12.8471	-	0.4323	12.3580	-	3.0543	14.1186
Tot. Red. AM[5]	-	−104.3430	−69.3129	-	−76.93	−58.262	-	−100.889	−81.313	-	−109.539	−80.254
Tot. Red. Med.[6]	-	−44.9518	−61.6945	-	−89.822	−52.401	-	−97.4809	−59.469	-	22.6611	−55.293

[1]Aff: Affinity; [2]Rel: Relative Error; [3] Abs: Absolute Error; [4] Tot.: Total Error (all functions); [5] Tot. Red. AM: Total Reduced Relative and Absolute Error (reduced % respect to AM in Table 8); [6] Tot. Red. Med.: Total Reduced Relative and Absolute Error (reduced % respect to Median in Table 8);

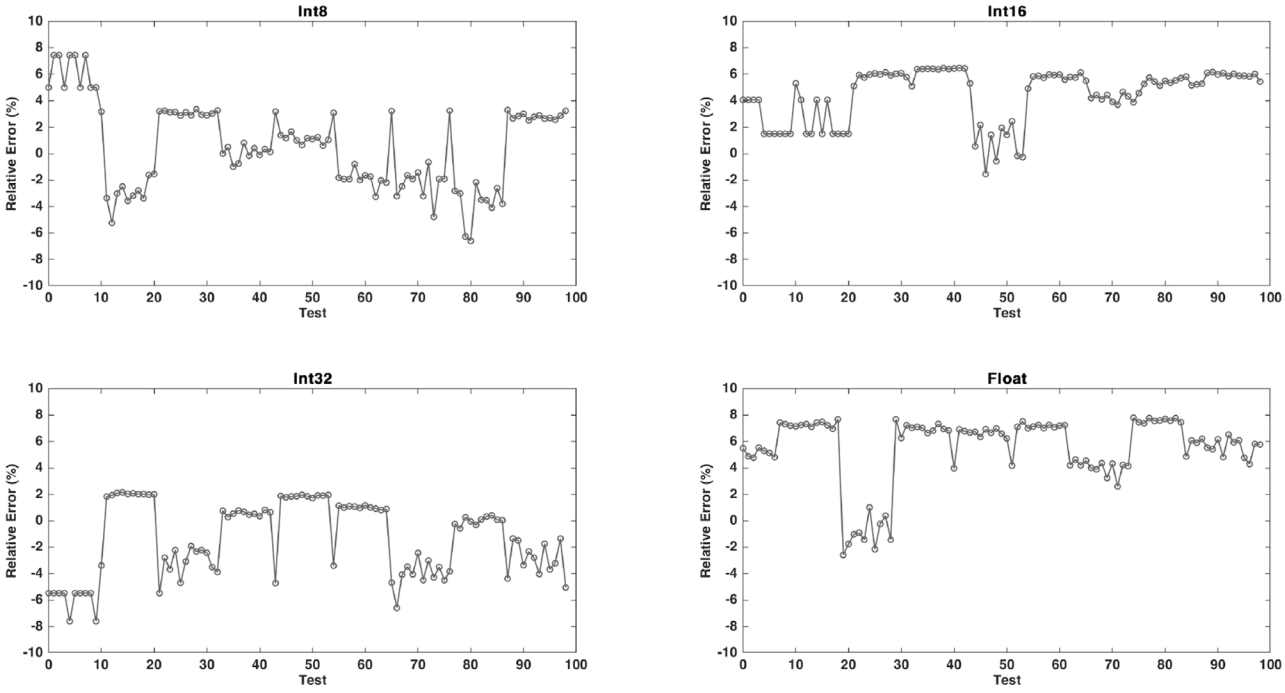


Fig. 13. Relative error plot compared to insertion sort function tests.

7. Conclusion and future work

This work has presented a metric useful to estimate in an early-stage design phase, the energy consumption related to the execution of embedded SW on a target processor. Such a metric, called J4CS (Joule for C Statements) is good for very fast estimation, comparison and selection activities. Then, more accurate approaches at lower abstraction levels can be used for more precise and time-consuming estimations. Beyond the pure SW domain, this metric can be easily exploited into specific HW/SW Co-Design methodologies and tools (e.g., [46]), in order to consider energy requirements during system-level design space exploration. Indeed, it is worth noting that this metric can be evaluated also in the HW domain, by using *High-Level Synthesis* (HLS) tools and *Hardware Description Language* (HDL) simulators able to provide energy information as output. Such values can be used to substitute the $\overline{E}'(p_j) \cdot I(p_j, z_i, b_{i,k})$ numerator value in Eq. (17). Moreover, J4CS can be also useful in ESL energy consumption estimation approaches that rely on the availability of an estimated energy consumption for each statement composing the ESL specification (e.g.,[47]). Future works will concentrate on the validation of J4CS-A metric with respect to the energy consumption measured on the actual boards and choosing other functions, different from the reference ones. Other future activities will focus on reducing absolute error estimations, introducing more accurate statistical analysis and models able to better consider processor architectural features. Some interesting opportunities, still at early-stage, will be related to the use of HW profilers [48], in order to evaluate estimation errors directly on-target and to the combined exploitation firstly of the Affinity metric [45], so to reduce such errors by identifying a proper distribution subset, and secondly of a more detailed static analysis of source-code, in order to assign different weights to different statements.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This work has been partially supported by the ECSEL RIA 2017-783162 FitOptiVis and ECSEL RIA 2018-826610 COMP4DRONES European projects.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.micpro.2020.103200.

References

[1] K. Grttner, R. Grgen, S. Schreiner, F. Herrera, P. Peil, J. Medina, E. Villar, G. Palermo, W. Fornaciari, C. Brandolese, D. Gadioli, E. Vitali, D. Zoni, S. Bocchio, L. Ceva, P. Azzoni, M. Poncino, S. Vinco, E. Macii, S. Cusenza, J. Favaro, R. Valencia, I. Sander, K. Rosvall, N. Khalilzad, D. Quaglia, CONTREX: Design of embedded mixed-criticality control systems under consideration of extra-functional properties, Microprocess. Microsyst. 51 (2017) 39–55, doi:10.1016/j.micpro.2017.03.012.

[2] Y. Park, S. Pasricha, F.J. Kurdahi, N. Dutt, A multi-granularity power modeling methodology for embedded processors, IEEE Trans. Very Large Scale Integr. VLSI Syst. 19 (4) (2011) 668–681, doi:10.1109/TVLSI.2009.2039153.

[3] V. Muttillo, G. Valente, L. Pomante, V. Stoico, F. D'Antonio, F. Salice, CC4CS: An off-the-shelf unifying statement-level performance metric for HW/SW technologies, in: Companion of the 2018 ACM/SPEC International Conference on Performance Engineering, in: ICPE '18, ACM, New York, NY, USA, 2018, pp. 119–122, doi:10.1145/3185768.3186291.

[4] GCov Profiler, 2018. (accessed: 21.10.2019) https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[5] J. Castillo, H. Posadas, E. Villar, M. Martínez, C.R. Darwin, Energy consumption estimation technique in embedded processors with stable power consumption based on source-code operator energy figures, in: XXII Conference on Design of Circuits and Integrated Systems, in: DCIS'07, 2007, p. 1.

[6] V. Muttillo, J4CS: An early-stage statement-level metric for energy consumption of embedded SW, in: 2019 8th Mediterranean Conference on Embedded Computing (MECO), 2019, pp. 1–5, doi:10.1109/MECO.2019.8760288.

[7] C. Brandolese, W. Fornaciari, L. Pomante, F. Salice, D. Sciuto, Affinity-driven system design exploration for heterogeneous multiprocessor SoC, IEEE Trans. Comput. 55 (5) (2006) 508–519, doi:10.1109/TC.2006.66.

[8] H. Sultan, G. Ananthanarayanan, S.R. Sarangi, Processor power estimation techniques: a survey, Int. J. High Perform. Syst. Archit. 5 (2) (2014) 93–114, doi:10.1504/IJHPSA.2014.061448.

[9] F.N. Najm, A survey of power estimation techniques in vlsi circuits, IEEE Trans. Very Large Scale Integr. VLSI Syst. 2 (4) (1994) 446–455, doi:10.1109/92.335013.

[10] T. Simunic, L. Benini, G. De Micheli, Cycle-accurate simulation of energy consumption in embedded systems, in: Proceedings 1999 Design Automation Conference (Cat. No. 99CH36361), 1999, pp. 867–872, doi:10.1109/DAC.1999.782199.
[11] W. Ye, N. Vijaykrishnan, M. Kandemir, M.J. Irwin, The design and use of simplePower: a cycle-accurate energy estimation tool, in: Proceedings 37th Design Automation Conference, 2000, pp. 340–345, doi:10.1145/337292.337436.
[12] A.B. Abril Garcia, J. Gobert, T. Dombek, H. Mehrez, F. Petrot, Cycle-accurate energy estimation in system level descriptions of embedded systems, in: 9th International Conference on Electronics, Circuits and Systems, vol. 2, 2002, pp. 549–552, doi:10.1109/ICECS.2002.1046224.
[13] V. Tiwari, S. Malik, A. Wolfe, Power analysis of embedded software: a first step towards software power minimization, in: IEEE/ACM International Conference on Computer-Aided Design, 1994, pp. 384–390, doi:10.1109/ICCAD.1994.629825.
[14] A. Sinha, A.P. Chandrakasan, JouleTrack-a web based tool for software energy profiling, in: Proceedings of the 38th Design Automation Conference (IEEE Cat. No. 01CH37232), 2001, pp. 220–225, doi:10.1109/DAC.2001.156139.
[15] S. Lee, A. Ermedahl, S.L. Min, N. Chang, An accurate instruction-level energy consumption model for embedded RISC processors, SIGPLAN Not. 36 (8) (2001) 1–10, doi:10.1145/384196.384201.
[16] S. Lee, A. Ermedahl, S.L. Min, N. Chang, Statistical derivation of an accurate energy consumption model for embedded processors, 2002.
[17] M. Sami, D. Sciuto, C. Silvano, V. Zaccaria, Instruction-level power estimation for embedded VLIW cores, in: Proceedings of the Eighth International Workshop on Hardware/Software Codesign. CODES 2000 (IEEE Cat. No. 00TH8518), 2000, pp. 34–38.
[18] S. Nikolaidis, N. Kavvadias, T. Laopoulos, L. Bisdounis, S. Blionas, Instruction level energy modeling for pipelined processors, J. Embedded Comput. 1 (3) (2005) 317–324.
[19] S. Sultan, S. Masud, Rapid software power estimation of embedded pipelined processor through instruction level power model, in: 2009 International Symposium on Performance Evaluation of Computer Telecommunication Systems, vol. 41, 2009, pp. 27–34.
[20] C. Brandolese, W. Fornaciari, L. Pomante, F. Salice, D. Sciuto, A multi-level strategy for software power estimation, in: Proceedings 13th International Symposium on System Synthesis, 2000, pp. 187–192, doi:10.1109/ISSS.2000.874048.
[21] E. Senn, N. Julien, J. Laurent, E. Martin, Power consumption estimation of a C program for data-intensive applications, in: B. Hochet, A.J. Acosta, M.J. Bellido (Eds.), Integrated Circuit Design. Power and Timing Modeling, Optimization and Simulation, Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 332–341.
[22] S. M, H. Blume, T. Noll, Power estimation on functional level for programmable processors, Adv. Radio Sci. 2 (2004), doi:10.5194/ars-2-215-2004.
[23] J. Livonius, H. Blume, T. Noll, Flpa-based power modeling and power aware code optimization for a TriMedia DSP, in: Proceedings of the ProRISC Workshop, 2005.
[24] H. Blume, D. Becker, L. Rotenberg, M. Botteck, J. Brakensiek, T. Noll, Hybrid functional- and instruction-level power modeling for embedded and heterogeneous processor architectures, J. Syst. Archit. 53 (2007) 689–702, doi:10.1016/j.sysarc.2007.01.002.
[25] C. Brandolese, F. Salice, W. Fornaciari, D. Sciuto, Static power modeling of 32-bit microprocessors, IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. 21 (11) (2002) 1306–1316, doi:10.1109/TCAD.2002.804104.
[26] C. Brandolese, W. Fornaciari, F. Salice, D. Sciuto, Source-level execution time estimation of C programs, in: Ninth International Symposium on Hardware/Software Codesign. CODES 2001 (IEEE Cat. No. 01TH8571), 2001, pp. 98–103.
[27] C. Brandolese, S. Corbetta, W. Fornaciari, Software energy estimation based on statistical characterization of intermediate compilation code, in: IEEE/ACM International Symposium on Low Power Electronics and Design, 2011, pp. 333–338, doi:10.1109/ISLPED.2011.5993659.
[28] C. Brandolese, W. Fornaciari, F. Salice, D. Sciuto, Timing and energy estimation of C programs, ACM Trans. Embedded Comput. Syst.(TECS) (2002).
[29] L. Bogdanov, Look-up table-based microprocessor energy model, in: Fifth International Scientific Conference ăEngineering, Technologies and Systemsg (TECHSYS 2016), 2016, pp. 180–185.
[30] k. Liu, A Simulation Based Approach to EstimateEnergy Consumption for Embedded Processors, Wrocaw University of Technology, 2015 Master's thesis.
[31] M. Hubner, J. Becker, Multiprocessor System-on-Chip: Hardware Design and Tool Integration, Springer, 2011, doi:10.1007/978-1-4419-6460-1.
[32] V. Tiwari, S. Malik, A. Wolfe, M.T. Lee, Instruction level power analysis and optimization of software, in: Proceedings of 9th International Conference on VLSI Design, 1996, pp. 326–328, doi:10.1109/ICVD.1996.489624.
[33] I. Nikolaidis, Arm system-on-chip architecture, 2nd edition [book review], Network, IEEE 14 (2000), doi:10.1109/MNET.2000.885658. 4–4
[34] M. Siegesmund, Embedded C Programming: Techniques and Applications of C and PIC MCUS, first ed., Newnes, Newton, MA, USA, 2014.
[35] CC4CS benchmark, 2018. (accessed: 21.10.2019) https://github.com/vnzstc/cc4cs.
[36] LEON3 processor, 2018. (accessed: 21.10.2019) https://www.gaisler.com/.
[37] Synthesizable VHDL Model of 8051, 2018. (accessed: 21.10.2019) http://www.newit.gsu.by/resources/CPUs/i8051/VHDL/Synthesizeable%20VHDL%20Model%20of%208051.htm.
[38] TSIM2 ERC32/LEON simulator, 2018. (accessed: 21.10.2019) https://www.gaisler.com/.
[39] LEON Bare-C Cross Compilation System (BCC), 2018. (accessed: 21.10.2019) https://www.gaisler.com/index.php/products/operating-systems/bcc.
[40] Dalton Project: 8051 microcontroller, University of California, 2018. (accessed: 21.10.2019) http://www.ann.ece.ufl.edu/i8051/.
[41] SDCC - Small Device C Compiler, 2018. (accessed: 21.10.2019) http://sdcc.sourceforge.net/.
[42] LEON3-FT SPARC V8 Processor LEON3FT-RTAX, 2018. (accessed: 21.10.2019) https://www.gaisler.com/doc/leon3ft-rtax-ag.pdf.
[43] UT699 32-bit Fault-Tolerant SPARCTM V8/LEON 3FT Processor, 2018. (accessed: 21.10.2019) https://www.cobhamaes.com/pagesproduct/datasheets/leon/UT699LEON3FTDatasheet.pdf.
[44] AT89C51 ATMEL Development Board, 2018. (accessed: 21.10.2019) https://www.indiamart.com/proddetail/at89c51-atmel-development-board-15939053291.html.
[45] C. Brandolese, W. Fornaciari, L. Pomante, F. Salice, D. Sciuto, Affinity-driven system design exploration for heterogeneous multiprocessor SoC, IEEE Trans. Comput. 55 (5) (2006) 508–519, doi:10.1109/TC.2006.66.
[46] V. Muttillo, G. Valente, D. Ciambrone, V. Stoico, L. Pomante, HEPSYCODE-RT: A real-time extension for an ESL HW/SW co-design methodology, in: Proceedings of the Rapido'18 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools, in: RAPIDO '18, ACM, New York, NY, USA, 2018, pp. 6:1–6:6, doi:10.1145/3180665.3180670.
[47] L. Berardinelli, A. Di Marco, S. Pace, L. Pomante, W. Tiberti, Energy consumption analysis and design of energy-aware WSN agents in fUML, in: G. Taentzer, F. Bordeleau (Eds.), Modelling Foundations and Applications, Springer International Publishing, Cham, 2015, pp. 1–17.
[48] A. Moro, F. Federici, G. Valente, L. Pomante, M. Faccio, V. Muttillo, Hardware performance sniffers for embedded systems profiling, in: 2015 12th International Workshop on Intelligent Solutions in Embedded Systems (WISES), 2015, pp. 29–34.

Vittoriano Muttillo received his Bachelor's degree and Master's Degree (summa cum laude) in Computer Science Engineering, and his PhD in Information and Communication Technologies (cum laude) from the University of L'Aquila. In 2014 he was a researcher at the Centre of Excellence DEWS (Design Methodologies for Embedded controllers, Wireless interconnect and System-on-chip), working on the development of middleware for FPGA's embedded multi-core architectures in the context of CRAFTERS (Constraint and Application driven Tailoring Framework for Embedded Real-time Systems) ARTEMIS-JU European Project. Currently, he is a research fellow in the area of Information and Communication Technologies (ICT) at the Department of Information Engineering, Computer Science and Mathematics (DISIM), University of L'Aquila. His research interests focus on Embedded Systems, with a particular emphasis on Electronic Design Automation and Model-Based System-Level HW/SW Co-Design area. He works on the development of EDA tools, mainly oriented to properly manage Mixed-Criticality and Cyber-Physical Systems on heterogeneous multi/many-core platforms.

Paolo Giammatteo is a Research Fellow with master degree in Physics and PhD in Power Electronics. His topics of interest are machine learning, big data, embedded systems, statistical physics, mathematical modeling of complex systems, scale-free networks, analysis. During his academic years, he matured a good experience in various programming languages, designing and developing software for devices such as GPU and embedded electronic systems for data acquisition and control of engineering systems. He also improved his informatics skills through a job experience in software development for web applications and database management systems for massive data analysis.

Vincenzo Stoico has received the Bachelor's degree in Computer Science from the University of L'Aquila (Italy) in 2017, and a Double Degree in Software Engineering from University of L'Aquila (Italy) and Mlardalens University (Sweden) in 2019. He graduated with a Master Thesis entitled "A Model-Driven Approach for modeling Heterogeneous Embedded Systems". From 2019, he is a Ph.D. student at University of L'Aquila. His activities focus mainly on Model-Based Design for Embedded Systems.

Luigi Pomante has received the 'Laurea' (i.e. BSc+MSc) Degree in Computer Science Engineering from 'Politecnico di Milano' (Italy) in 1998, the 2nd Level University Master Degree in Information Technology from CEFRIEL (a Center of Excellence of 'Politecnico di Milano') in 1999, and the Ph.D. Degree in Computer Science Engineering from 'Politecnico di Milano' in 2002. He had been a Researcher at CEFRIEL from 1999 to 2005 and, in the same period, he had been also a Temporary Professor at 'Politecnico di Milano'. From 2006, he is an Academic Researcher at Center of Excellence DEWS ('Universitá degli Studi dell'Aquila', Italy). From 2008 he is also Assistant Professor at 'Universitá degli Studi dell'Aquila' (he is responsible of the 'Embedded Systems' course). His activities focus mainly on Electronic Design Automation (in particular Electronic System-Level HW/SW Co-Design) and Networked Embedded Systems (in particular Wireless Sensor Networks). In such a context, he has been author (or co-author) of more than 100 articles published on international and national conference proceedings, journals, and book chapters. He has been also session chair, reviewer, and member of several TPCs related to his research topics. From 2010, he has been in charge of scientific and technical issues on behalf of DEWS in several European and national research projects.