# Software Energy Estimation Based on Statistical Characterization of Intermediate Compilation Code

Carlo Brandolese, Simone Corbetta, William Fornaciari
Politecnico di Milano – Dipartimento di Elettronica e Informazione
Piazza L. Da Vinci, 32 – 20133 Milano, Italy
carlo.brandolese@polimi.it, scorbetta@elet.polimi.it , william.fornaciari@polimi.it

*Abstract*—**Early estimation of embedded software power consumption is a critical issue that can determine the quality and, sometimes, the feasibility of a system. Architecture-specific, cycle-accurate simulators are valuable tools for fine-tuning performance of critical sections of the application but are often too slow for the simulation of entire systems. This paper proposes a fast and statistically accurate methodology to evaluate the energy performance of embedded software and describes the associated toolchain. The methodology is based on a static characterization of the target instruction set to allow estimation on an equivalent, target-independent intermediate code representation.**

## I. Introduction

The increasing complexity of embedded software and the increasing demand for energy efficiency are pushing the efforts toward an early energy characterization of the target software. Energy-efficient hardware and software design are required to ensure continuous scalability [10], and power optimization is now an issue spanning all the design phases and abstraction levels [9]. As a matter of fact, due to the ever increasing complexity and density of embedded software in modern embedded systems, energy optimization is gaining utmost attention and it is of paramount importance to be able to early characterize the software at source-code level.

The early-stage characterization of embedded software power requirements can be performed either statically or dynamically. Static characterization has no overhead on the run-time performance of the application, since the analysis and processing phases are done off-line without, in general, executing the binary code. However, an off-line analysis does not consider input data and interaction with the operating environment. On the other hand, dynamic characterization is able to consider data-dependent energy requirements, at the cost of increased overhead. In this perspective, the methodology proposed in this paper tries to take advantage from both scenarios, at the same time mitigating their disadvantages. The proposed methodology is based on a static characterization of the structural features of the source code, while input-dependent energy estimate is done dynamically, through appropriate instrumentation.

### A. Related work

Traditionally, estimation of the software power requirements has been performed by means of ad-hoc, energy and cycle-accurate instruction-set simulators (ISS). These simulators provide precise estimates in terms of power and performance characteristics [11], but are generally very computationally demanding; in addition, they are not able to propagate and redistribute power consumption metrics to the basic entities (i.e., source code) that are responsible of that consumption.

To solve this issue, and to try to raise the abstraction level of the estimates, the authors in [9] propose to use sampling-based profiling tool (e.g., `gprof`) providing coarse-grained estimates, and merge those estimates with the output from traditional ISS. The rough estimates were done on a per-function basis, thus without any possibility to propagate the estimate to finer elements. Fine-grain estimates are very important for optimization purposes, since most of the optimization are done inside function body.

Compilation-based techniques are indeed an interesting approach to guide optimization starting from the entire source code analysis. A standard compiler is employed to generate an intermediate representation of the code, which is in turn transformed into a control flow graph, on which relevant power and performance figures are annotated to obtain cycle-accurate models. A great advantage of these techniques lays in their ability to account for the effects of different compilation optimizations. Nevertheless, they cannot redistribute the estimate to the source level entities causing power consumption.

A first attempt in redistributing energy estimates to source-level entities is proposed in [12], through instrumentation promotion of C code to C++ code.

### B. Novel contributions

The purpose of this paper is to propose a methodology to estimate performance and energy requirements in embedded software, conveying the advantages of ISS approaches and dynamic approaches, and mitigating their disadvantages. The proposed approach is based on a statistical analysis and characterization of an intermediate assembly-level code representation, instrumentation and execution on the host machine. The main novel contributions of the proposed methodology are outlined in the following.

First of all, the methodology does not require to execute an ISS, even in those cases where the software is required to interface to the external world, e.g. to hardware peripherals on the target SoC. Secondly, the intermediate code (LLVM) is very close to the target assembly code, but still sufficiently abstract to allow for a source-level analysis. In this way, the analysis performed on the intermediate code combines efficiency, accuracy and flexibility. Finally, execution on the host machine does not require an instruction-set simulator of the target machine nor the bare hardware, thus reducing the costs and time of the analysis.

## II. Models

The basic ideas supporting the proposed approach have been explored by the authors in previous works [4]. The main disadvantages and limitations of such approaches were primarily related to the very abstract intermediate representation that was used, namely a decorated abstract syntax tree. The new approach proposed in this paper overcomes such limitations by adopting as intermediate representation the assembly-level LLVM code produced by the open-source project *LLVM Compiler Infrastructure* [3].

Based on this representation, the modeling approach can be split into three main phases:

1) *Source code modeling.* This first modeling phase is aimed at decoupling the influence of the *static structure* of the source code both from the specific target architecture and from the input runtime data.
2) *Dynamic behavior modeling.* The dynamic behavior is accounted for by performing a basic-block profiling at intermediate-representation level (i.e., LLVM code). Such a profiling can be accomplished by executing the code on a host machine, possibly resorting to some support libraries that model hardware devices.
3) *Target machine modeling.* Finally, the actual timing and energy characteristics of the target architecture need to be taken into account to derive the final estimate. To this purpose it is necessary to derive a target-specific cost model of the LLVM instruction set.

Each phase has been studied accurately and a simple yet sound mathematical model has been built as a foundation. Such models are described in the following.

### A. Source code and dynamic models

Given a *program run* (i.e., the program and its input data), let $c_e$ and $c_t$ be respectively the energy requirements ($\mu$J) and the execution time (clock cycles) in that execution. The translation into the LLVM intermediate representation leads to a list of basic-blocks, each composed of several instructions of different types. Let $B$ be number of basic blocks and $L$ be the number of different LLVM instructions. The *representation matrix* $\mathbf{R}$ determines the types of instructions belonging to a basic block. More precisely, $R_{i,j} = n$ means that the $i$-th basic block contains $n$ LLVM instructions of type $j$. The matrix $\mathbf{R}$ is thus a *static representation* of the source code.

The instrumentation process is performed by adding a suitable tracing function call to each basic-block. Once the instrumented program is compiled, linked and run, the profile $p = [\, p_1\, p_2\, \ldots\, p_B\,]$ is available. The element $p_i$ is the execution count of the $i$-th basic block. From $p$ and $\mathbf{R}$, it is possible to calculate the vector of execution counts of all instruction types as:

$$s = p \cdot \mathbf{R} = [\, s_1\; s_2\; \ldots\; s_L\,] \tag{1}$$

where $s_j$ is the cumulative execution count of the instruction of type $j$ in that program run. Since each instruction of the intermediate language corresponds to one ore more elementary operations executed by the target architecture, the evaluation of the overall cost of a program run requires decomposing the intermediate instructions into such elementary contributions. The elementary operations considered here depend on the target architecture model and more precisely on the target assembly language. Let $K$ be the number of instructions of the target assembly language and $\mathbf{T}$ be the *translation matrix* with $L$ rows corresponding to the types of LLVM instructions, and $K$ columns corresponding to the different target assembly instructions::

$$\mathbf{T} = \begin{bmatrix} T_{1,1} & \cdots & T_{1,K} \\ \vdots & \ddots & \vdots \\ T_{L,1} & \cdots & T_{L,K} \end{bmatrix} = \begin{bmatrix} T_1 \\ \vdots \\ T_L \end{bmatrix} \tag{2}$$

where $T_{i,j} = n$ indicates that the LLVM instruction of type $i$ requires $n$ instruction of type $j$ of the target assembly. The same matrix can be also viewed as a collection of $L$ row vectors $T_i$. Each vector represents the *translation model* of a specific LLVM instruction.

Indicating with $\mathcal{L}_m$ the LLVM instruction of type $m$ and with $\mathcal{T}_i$ the target assembly instruction of type $i$, the cost of the LLVM instruction for the specific target architecture is:

$$\text{cost}(\mathcal{L}_m) = \sum_{i=1}^{K} T_{m,i} \cdot \text{cost}(\mathcal{T}_i) \tag{3}$$

or, in matrix form:

$$\text{cost}(\mathcal{L}_m) = T_m \cdot [\,\text{cost}(\mathcal{T}_1)\; \ldots\; \text{cost}(\mathcal{T}_K)\,]^T = T_m \cdot K^T \tag{4}$$

Combining all the equations introduced so far, and indicating with $K_e$ and $K_t$ the cost vectors related to energy and execution time respectively, the overall costs of the specific program run can be calculated by the following relations:

$$c_e = p \cdot \mathbf{R} \cdot \mathbf{T} \cdot K_e^T \tag{5}$$

$$c_t = p \cdot \mathbf{R} \cdot \mathbf{T} \cdot K_t^T \tag{6}$$

These two equations, along with the definition given above of the matrices involved, summarize the proposed methodology.

### B. Target Architecture Characterization

The model described in the previous section relies on several matrices that need to be determined by means of asuitable procedure, in particular:

- $\mathbf{R}$. The representation matrix is determined by the front-end portion of the toolchain described in Section III. The source code is compiled into LLVM code, which is in turn transformed into a basic-block model file describing the mix of LLVM instructions per each basic block.
- $\mathbf{T}$. The translation matrix is constructed by juxtaposition of the translation model vectors $T_i$ for each LLVM instruction. Such vectors express the static correlation between the mix of instructions used by the LLVM compiler and the actual mix of assembly instructions

generated by the target compiler. The procedure adopted to determine the individual translation model vectors is the core topic of this section.

- $K_e, K_t$. These vectors are assumed to be known and provided by the target architecture manufacturer. They express, for each instruction of the target assembly, the average execution time (clock cycles) and the energy consumption ($\mu$A/clock cycle). While execution times are always known (at least in best and worst case conditions), energy characterization is usually much complex to obtain. Several analyses, though, have shown that the current absorbed per clock cycle by the instructions of RISC-like architectures have a very limited variance. In this case, the overall energy consumption mainly depends on the execution time of each assembly instruction.
- $p$. The profile vector models the dynamic behavior of an application run, i.e. a program and a specific set of input data. It is derived by the back-end portion of the toolchain, as described in Section III.

Let us now consider the methodology adopted to determine the translation vectors for each LLVM instruction $\mathcal{L}_m$. The procedure we have adopted starts from the identification of a suitable set of simple but varied source codes and determine the correlation between the counts of each LLVM instruction of the LLVM-compiled code and the counts of the target instructions of the same code compiled with the target compiler.

Let then $\mathcal{S} = \{S_0, S_1, \ldots, S_N\}$ be the set of source codes, $L_{i,j}$ the number of LLVM instructions of type $j$ in the LLVM code corresponding to the source code $S_i$ and $D_{i,k}$ the number of target instructions of type $k$ in the destination assembly code corresponding to the same source code $S_i$. The analysis of all source codes for each instruction leads to the over-constrained system of $N \times L$ equations:

$$L_{i,j} = \sum_{k=1}^{K} D_{i,k} \cdot x_{j,k} \tag{7}$$

in the $N \times K$ unknowns $x_{j,k}$. Collecting all the equations corresponding a specific LLVM instruction $\mathcal{L}_j$ we obtain a system of $N$ equations in $K$ unknowns, that can be written in matrix form as:

$$\mathbf{L}_j = \mathbf{D}\mathbf{x}_j \tag{8}$$

where $\mathbf{L}_j$ is an $N \times 1$ vector of LLVM instruction counts, $\mathbf{D}$ is the $N \times K$ matrix of target instruction counts and $\mathbf{x}_j$ is the unknown $K \times 1$ vector constituting the desired model of the instruction $j$ of the LLVM instruction set.

This set of equations is though too general, since it assumes that an LLVM instruction can statistically be translated into a combination of possibly *all* target instructions. This is clearly visible from (8), where the coefficients of the equations are in the same matrix $\mathbf{D}$ for all instructions, i.e. for all $j$.

To limit this generality, we restrict the set of target instructions that can contribute to the model of each specific LLVM instruction. This can be easily achieved defining an *initial model vector* $\mathbf{M_j} = \{m_{j,1}, m_{j,2}, \ldots, m_{j,K}\}$ of coefficients $m_{j,p} = \{0, 1\}$. A value $m_{j,p} = 1$ indicates that the target

**Algorithm 1** Solve a constrained least square problem.

```
$M = \text{diag}(\mathbf{M}_j)$
$A = D \cdot M$
$B = \mathbf{L}_j$
repeat
  $X = (A^T \cdot A)^{-1} \cdot A^T \cdot B$
  for $i = 1$ to $K$ do
    if $x_i < 0$ then
      $m_{i,i} = 0$
    end if
  end for
  $A = A \cdot M$
until $X > 0$
```

instruction of type $p$ participates to the model of the LLVM instruction of type $j$. The general model of (8) can be modified to include the model vector by simply transforming the vector $\mathbf{M_j}$ in a diagonal matrix and multiplying it by the coefficient matrix $\mathbf{D}$, that is:

$$\mathbf{L}_j = \mathbf{D}\,\text{diag}(\mathbf{M}_j)\,\mathbf{x}_j = \mathbf{D}'_j\mathbf{x}_j \tag{9}$$

where $\mathbf{D}'_j = \mathbf{D}\,\text{diag}(\mathbf{M}_j)$. This restriction leads to a more realistic representation of LLVM instructions. By solving the problem of (9) in least square sense some of the coefficients of the vector $\mathbf{x}_j$ may be negative, leading to a hardly meaningful interpretation of the model. For this reason a further constraint is imposed to force the coefficients to be strictly positive, leading to the following formulation of the problem:

$$||\mathbf{D}'_j\mathbf{x}_j - \mathbf{L}_j||_2^2 \quad \text{with} \quad \mathbf{x}_j \geq 0 \tag{10}$$

This equation belongs the class *bound-constrained least square* problems and in particular to the class of *non-negative least square* problems or *NNLV*. It can be solved several ways, summarized and described for example in [1]. The algorithm that we have adopted is the Lawson–Hanson method described in full detail in [2], Ch. 23, p. 161, and outlined by the pseudocode of Algorithm 1. The idea behind this approach is that of iteratively excluding those terms of the equations, i.e. columns of the matrix $\mathbf{D_j}$, that provide a negative contribution to the model until all the coefficients in $\mathbf{x}_j$ are positive or zero.

Once all equations have been solved, the translation matrix can be built, recalling (2), as $T = [x_0^T \; x_1^T \; \cdots \; x_L^T]$.

## III. Flow

The analysis and estimation flow has been implemented using the LLVM compiler infrastructure and developing a completely custom toolset called SWAT – *SoftWare Analysis Toolset*. The portion of the flow that is relevant for the estimation scenario outlined in this paper is outlined in Fig. 1. It starts from the set of application source files (`*.c`) and through a sequence of transformations produces some reports (`*.rpt`) and a set of back-annotated source files (`*.ba`).

The transformation performed by the tools of the flow have been collected into four *phases*, indicated by numbered black boxes, and are detailed in the following.

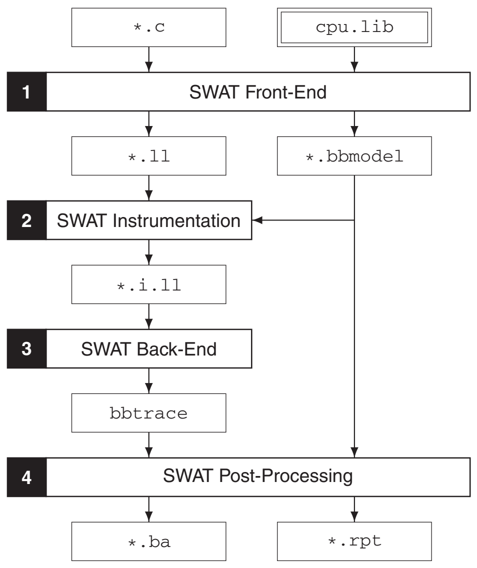

Fig. 1. Simplified estimation flow.

1) *SWAT Front-End.* Compiles each source file into architecture-independent LLVM code, which is then read by a SWAT tool to extract a *model* of each basic block (`*.bbmodel`) consisting of the list of opcodes, functions called directly or though pointers, size, timing, energy and location data for back-annotation. Data for timing and energy characterization is found in the *target CPU library* (`cpu.lib`), which is the result of the characterization phase (Section IV-B). Finally, all entities names (basic blocks, functions, ...) are converted into unique numeric identifiers to improve performance.
2) *SWAT Instrumentation.* Instrumentation consists in a two-step process. The first step (*meta-instrumentation*) enriches each basic block of the LLVM code with all the relevant figures in the form of a special comment. This step requires the knowledge of the assembly structure and semantics and is thus performed by a custom extension of the assembly analysis libraries of the LLVM compiler. The second phase, on the other hand, is agnostic of the details of the LLVM code and operates only on the tags generated during meta-instrumentation by translating them into actual calls to suitable tracing functions. Translation is based on *expansion rules* collected into an instrumentation library. The output is a new LLVM assembly file (`*.i.ll`).
3) *SWAT Back-End.* The back-end of the SWAT flow performs two main operations. First, it translates all the instrumented LLVM files into host assembly code, which is then assembled and linked into an executable program. Secondly, it runs the executable and collects the execution trace (`bbtrace`). In general, such a trace can contain several kinds of information concerning the execution; when used in the estimation flow described in this paper, it consists of a list of the identifiers of the basic blocks that have been executed.
4) *SWAT Post-Processing.* The post-processing phase analyzes the execution trace, converts it into basic block execution counts, combines the counts (i.e., the dynamic behavior of the program) with the static costs from the basic block models (the static view of the program) and calculates timing and energy (currently implemented in a preliminary version) figures for each C code line. The output of this phase, and of the estimation process, consist of a set of textual reports (`*.rpt`) and an annotated version of each source file (`*.ba`)

The output of the flow is thus twofold: on one hand it consists of a very detailed (source code line) characterization of the program, and on the other hand it collects overall figures and statistic concerning the entire application. Such overall figures are those reported in Section IV, when discussing the experimental results.

## IV. Results

This section discusses the results we have obtained by applying the proposed characterization and estimation methodologies to the ReISC 3 Core developed by STMicroelectronics. To evaluate the accuracy and efficiency of the methodology and of the SWAT toolchain, our estimates have been compared against the execution times calculated using the cycle-accurate ReISC instruction-set simulator.

### A. Target architecture

The experiments to assess the accuracy and performance of the proposed flow have been conducted using the STMicroelecronics ReISC core. ReISC (*Reduced Energy Instruction Set Computer*) is an ultra-low power, up-to-32bit processor [5] [6] [7] offering hardware support for 8/16/20/32 data sizes, variable instruction length and support for secure data.

The core has an enhanced RISC architecture with a 3-stages pipeline. The Harvard topology allows concurrent data memory accesses and instruction fetches, while instruction cache, data cache and MMU are optional plug-ins. The register file and the ALU are optimized to work with different data sizes with a granularity of a single instruction. Long and small integers, pointers, and char data types can live together in the pipeline and in the register file, saving power while keeping low-end 32bit processors performance. Short relative jumps are supported at no code-space expense, while optimized long branch instruction can jump directly into the whole address space. The ReISC roadmap includes multi-core SoCs and many DSP extensions to the instruction set.

In this paper we refer to the ReISC III Core, which is the main building block of the family. It belongs to memory/register architecture, rather than the prevailing register/register architecture adopted by most of the commercial RISC microprocessors. This allows reducing the energy for inter-instruction data transfer, and to obtain a more compact instruction size. Multiple addressing modes are supported by the ReISC instruction set: absolute addressing, indirect addressing, incremental addressing, index addressing, circular addressing, bit-reversed addressing etc.

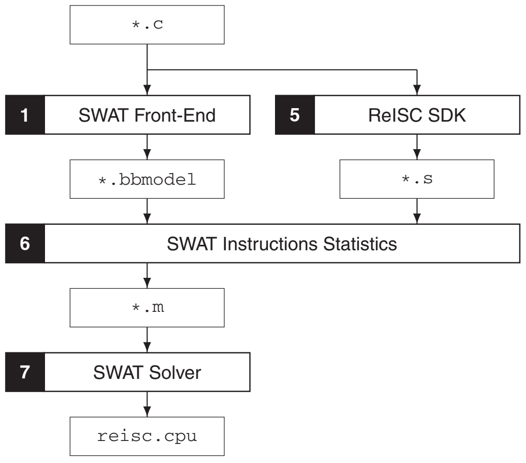

Fig. 2. ReISC 3 core characterization flow.

## B. Characterization methodology

The methodology adopted for the characterization of ReISC Core follows the model described in Section II-B and has been implemented using the LLVM infrastructure, the LLVM compiler front-end, the SWAT toolchain and the ReISC software development kit. A simplified view of the flow is shown in Fig. 2 and is described in the following.

1) *SWAT Front-End.* This part of the flow is the same used for estimation (see Section III). The only difference is that, in this case, the basic block models generated obviously do not contain any timing or energy figures of the target core. They are only used to derive LLVM instruction usage statistics.
5) *ReISC SDK.* The ReISC compiler is used to compile into symbolic assembly code (`*.s`) the same source files.
6) *SWAT Instruction Statistics.* Each pair of assembly files is processed by the SWAT tools that extract instruction statistics both from an LLVM basic block model file and from a generic assembly file (in this case a configuration file describing the assembly syntax is required). Such statistical information is provided in the form of tables that are, eventually, translated in Octave format for solving.
7) *SWAT Solver.* The solver is basically a set of scripts for Octave implementing Algorithm 1 and some statistical analysis functions.

To significantly exercise the two front-ends the set of input source files used for model construction must be selected to be as large and varied as possible. We have used a subset of the source codes constituting the `gcc` test suite. The set contains 2 135 source files with a size ranging from 4 to 1 350 line of code, for an overall count of 177 000 source code lines. This set of files allowed us to build and solve 34 over-constrained systems of equations, one system per each LLVM instruction. The large number of equations (449 on average), compared with the number of parameters (9 in the most complex case) guarantees a good basis for a statistical analysis.

TABLE I
WCET BENCHMARK CHARACTERISTICS

| Benchmark | LOC | S | L | N | A | B | R |
|---|---|---|---|---|---|---|---|
| `cover` | 240 | × | × | | | | |
| `bs` | 114 | | × | | × | | |
| `bsort100` | 128 | | × | × | × | | |
| `fibcall` | 72 | × | × | | | | |
| `matmult` | 163 | × | × | × | × | | |
| `cnt` | 267 | | × | × | × | | |
| `compress` | 508 | | × | × | × | | |
| `recursion` | 41 | × | | | | | × |
| `edn` | 285 | × | × | × | × | × | |
| `expint` | 157 | × | × | × | | | |
| `lcdnum` | 64 | | × | | | × | |

TABLE II
ENERGY ESTIMATION RESULTS AND ERRORS

| Buffer Size | ReISC ISS Energy (nJ) | SWAT Flow Energy (nJ) | Relative Error |
|---|---|---|---|
| 14 | 38,10 | 36,62 | -3,89 % |
| 20 | 56,23 | 52,88 | -5,96 % |
| 30 | 74,43 | 69,57 | -6,52 % |
| 50 | 110,28 | 102,09 | -7,42 % |
| 100 | 199,82 | 192,82 | -3,50 % |
| 150 | 288,46 | 302,44 | 4,85 % |

## C. Estimation results

The estimation flow has been applied to a set of 10 source code examples taken from the WCET benchmark suite developed at the Mälardalen Real-Time Research Center (MRTC) of the Mälardalen University [8]. These benchmarks have been selected as a first validation set because most of them only use integer arithmetic and do not require external library functions.

At the present status of development of our characterization methodology, floating point arithmetic has not been considered since the target core has no floating-point unit and thus all operations are emulated via software routines. The same restriction applies to the usage of external library functions, which would need a similar off-line characterization process.

Table I summarizes the characteristics of the benchmarks selected for evaluation and the meaning of the last 6 columns is as follows: (S) single path program, (L) contains loops, (N) contains nested loops, (A) uses arrays and/or matrices, (B) performs bitwise operations, (R) contains recursion.

It is worth noting that some of the benchmarks have been designed to operate on fixed or fixed-size data. To better evaluate the accuracy of our estimation methodology and toolchain, we have modified such programs making them dependent on a macro parameter whose value is set on the compiler's command line.

As an example, consider the `compress` benchmark, originally designed to compress a 50 bytes data buffer. We have modified the code by making the buffer size dependent on a macro and run the simulation and estimation flows for each assignment. The results are summarized in Table II, where the reference and estimated energies (nJ) are reported.

TABLE III
ENERGY CONSUMPTION ESTIMATION ACCURACY

| Benchmark | Error | Error (absolute) | Standard deviation |
|---|---|---|---|
| `cover` | -2,05 % | 2,38 % | 0,0170 |
| `bs` | -2,20 % | 5,02 % | 0,0780 |
| `bsort100` | -10,78 % | 10,78 % | 0,0164 |
| `fibcall` | 8,59 % | 8,59 % | 0,0419 |
| `matmult` | -12,94 % | 12,94 % | 0,0616 |
| `cnt` | 5,08 % | 5,08 % | 0,0526 |
| `compress` | -3,74 % | 5,36 % | 0,0447 |
| `recursion` | 3,63 % | 3,63 % | 0,0076 |
| `edn` | -5,74 % | 5,74 % | 0,0000 |
| `expint` | -1,23 % | 2,24 % | 0,0217 |
| `lcdnum` | -4,15 % | 4,15 % | 0,0211 |

The estimates refer to a ReISC III core with 1.0 V power supply and operating at 50 MHz. The preliminary characterization provided by STMicroelectroncs indicates an energy of 19 pJ per clock cycle (i.e. 950 $\mu$A/cc) for the core and of 20 $\mu$W/MHz (i.e. 1 mA/cc) for the scratchpad memory. The reference energy consumption figures have been determined using the `reisc-run` instruction-set simulator, configured to neglect the contribution of the memory subsystem to the overall energy consumption.

Similar experiments have been performed for other benchmarks and the results are reported in Table III. With an overall average error of 5.98%, these first estimation results are more than encouraging. Nevertheless, they show that the relative estimation error is slightly biased towards an underestimation. Current work is devoted to analyze this phenomenon to determine the reasons behind it.

### D. Performance

We have measured the execution times of both our toolchain and the REISC instruction-set simulator, used with tracing enabled. The target-specific estimation flow requires compilation and instruction-set simulation and almost no overhead for post-processing, while the SWAT flow requires two compilations (LLVM and host-native), model construction, instrumentation, tracing, postprocessing and back-annotation.

Two are the main advantages of the proposed approach: first, profiling and tracing are performed on basic blocks rather than on single assembly instructions, and, second, tracing is performed by executing native code on the host rather than using an instruction-set simulator. This leads to a speedup of the SWAT flow of approximately 410 times over the target ISS based approach.

## V. CONCLUSIONS

This paper presented a model, a methodology and a complete toochain to perform performance and energy estimation of embedded applications, operating at source code level. The approach is based on the decoupling of static source code modeling, dynamic data-dependent behavior analysis and target architecture characterization. This has been achieved by extending and revising a model previously developed by the authors and by implementing a complete toolchain to automate the process. The toolchain is based on several tools developed from scratch, some tools built on top of the LLVM Compiler Infrastructure and the LLVM compilation front-end.

As a first target platform the ultra-low power ReISC III core developed by STMicroelectronics has been selected, since accurate timing and preliminary average energy figures are available. The energy estimation results obtained have been compared against the figures provided by the proprietary ReISC instruction-set simulator. Models for different architectures have also been built and are curretly being tuned and verified with respect to the specific power-enabled instruction-set simulators. The average absolute error obtained on a set of integer benchmarks is less than 6 % and the estimation toolchain proved to be appproximately 400 times faster than instruction-set simulator. Such high performance enables the usage of the estimation flow within an iterative optimization engine based, for example, on design space exploration.

Future work will concentrate on the characterization of the floating-point emulation library of the ReISC III processor and on the analysis of the growing set of results that the authors are collecting. Further effort will be devoted to a better modeling of the memory contribution to the overall power consumption.

## ACKNOWLEDGMENTS

This research activity has been supported by European Union funding within the framework of the IP 247999 Project "COMPLEX – *COdesing and power Management in PLatform-based design space EXploration*".

## REFERENCES

[1] J. L. Mead, R. A. Renaut, *Least squares problems with inequality constraints as quadratic constraints*, Linear Algebra and its Applications, Vol. 432, No. 8, pp. 1936–1949, Apr. 2010.

[2] C. L. Lawson, R. J. Hanson, *Solving Least-Squares Problems*, Prentice-Hall, 1974.

[3] The LLVM Compiler Infrastructure, available at http://www.llvm.org.

[4] C. Brandolese, W. Fornaciari, D. P. Scarpazza, *Source-Level Energy Estimation and Optimization of Embedded Software*, IEEE Latin America Symposium on Circuits and Systems, (LASCAS'10) Foz do Iguazu, Paran, Brazil, February 2010.

[5] E. Guidetti, STMicroelectronics, *Energy Management Complexity for Ultra Low Power Multi-Processor System-on-Chip: An Industrial Perspective about System Architectures, Technology and Design Methodology*, Talk at the Design Automation & Test in Europe, DATE'2010, Dresden, Germany, Mar. 2010.

[6] E. Guidetti, STMicroelectronics, *Single and multiprocessor architectures for wireless sensors with very small energy budgets*, Second Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures, International Conference on Architecture of Computing Systems (ARCS'2011), Lake Como, Italy, Feb. 2011.

[7] STMicroelectronics, AST Advanced Microprocessor Design, *ReISC III Architecture Manual and Instruction Set*, Version 2.2, Nov. 2009.

[8] WCET Benchmark Programs, available at http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.

[9] T. Simunic, L. Benini and G. De Micheli, *Energy-efficient design of battery-powered embedded systems*, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 9, No. 1, pp. 15–28, 2001.

[10] International Technology Roadmap for Semiconductors, Design chapter, 2010. Available at http://www.itrs.net/

[11] D. Brooks, V. Tiwari, M. Martonosi. *Wattch: a framework for architectural-level power analysis and optimizations*. Proceedings of International Symposium on Computer Architecture. New York, USA, 2000.

[12] M. Ravasi and M. Mattavelli. *High-level algorithmic complexity evaluation for system design*. Journal of Systems Architecture, Vol. 48, No. 13–15, pp. 403–427, 2003.