ECAlogic: Hardware-Parametric Energy-Consumption Analysis of Algorithms [*]

Marc Schoolderman Jascha Neutelings Rody Kersten Marko van Eekelen [†]

Institute for Computing and Information Sciences, Radboud University Nijmegen

{mschool,jneutelings,rodykers,marko}@science.ru.nl

Abstract

While green software is a popular topic in computer science nowadays, the average programmer still has little options for analysis of the energy-efficiency of his/her software. Analysis is mostly done dynamically, for which a complex measurement set-up is needed. Using a static analysis which predicts the energy-consumption, would be more accessible and more cost-effective.

This paper presents ECALOGIC[1], a tool that implements a static analysis that can bound the energy consumption of algorithms. The tool is parametric with respect to a set of hardware component models. Its results are symbolic over the program parameters.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; C.0 [*Computer Systems Organization General*]: Modeling of computer architecture

Keywords Energy-Consumption, Static Analysis, Resource Analysis, Green Software, Green Computation

1. Introduction

In the last decades, energy-efficiency has become an important topic in computer science. Greenness of hardware has received much attention. Research on green software has mostly been focused on providing guidelines and design patterns for programmers, as well as profiling of hardware for analysis of low-level software. Estimation of the energy-consumption of algorithms is still a challenging task for the average programmer, who does not have access to a measurement set-up.

We present ECALOGIC, a tool for static energy-consumption analysis. Using this tool in combination with a model of the target hardware, a programmer can statically bound the energy-consumption of his/her software design. The tool provides an upper bound that is symbolic over the input parameters of the program.

1.1 Energy Consumption Analysis

ECALOGIC is an implementation of the static analysis presented in [5] and [6]. This analysis takes a set of component models and an algorithm that controls such component abstractions as input and calculates an upper bound on the consumed time and energy. It is based on a Hoare logic with derivation rules. As an example, we show a simplified version of the if-rule, where Γ is a set of energy-aware component states and $\mathfrak{t}$ is the global time:

$$\frac{\{\Gamma_1;\mathfrak{t}_1\}e\{\Gamma_2;\mathfrak{t}_2\} \quad \{\Gamma_2;\mathfrak{t}_2\}S_1\{\Gamma_3;\mathfrak{t}_3\} \quad \{\Gamma_2;\mathfrak{t}_2\}S_2\{\Gamma_4;\mathfrak{t}_4\}}{\{\Gamma_1;\mathfrak{t}_1\}\textbf{if } e \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end if}\{\mathbf{lub}(\Gamma_3,\Gamma_4);\mathbf{max}(\mathfrak{t}_3,\mathfrak{t}_4)\}}$$

The guard and both branches are analysed separately. Then, the least upper bound of component states and the maximum of the timing of both branches is taken as the result of the conditional.

The input algorithms for the analysis are written in a simple "while"-type language. This language has a construction for explicitly calling *component functions*. Components are modelled at the API-level. A component model typically offers several component functions, each with an associated energy-footprint. As running example we take a wireless sensor node that takes measurements and communicates these wirelessly. The hardware components are a CPU, a sensor and a radio transceiver. The sensor might e.g. offer a component function $\texttt{Sensor}::\texttt{measure}()$.

Two separate forms of energy consumption are identified: *incidental*, where a call to a component function consumes energy immediately, and *time-dependent*, where the component consumes a certain amount of energy constantly depending on its state (e.g. "stand-by"). A call to a component function may inflict an incidental energy-usage, as well as change the component state, thereby influencing time-dependent energy-usage. A component state is modelled as a set of positive integers. There must be finitely many states and the states must form a lattice, making it possible to calculate a least upper bound over them. Also, the energy-usage of the component must be monotonic with respect to the component states. Hence, the lattice ordering must be such that greater states mean higher energy consumption.

The static analysis is focused on energy-consumption and assumes that several properties of the input algorithm are given, either by the programmer of by a pre-analysis using other tools. All while loops should be annotated by a symbolic upper bound on the number of iterations (ergo, all algorithms terminate). Furthermore, when a program variable is used in such an upper bound or a function-call, its value at that point in the program will influence the analysis. As the results of the analysis are symbolic over the input parameters of the program, at those program points an annotation is expected expressing the current value of the referenced variables in terms of the input parameters.

[*] This work is part of the IOP GenCom GoGreen project, sponsored by the Dutch Ministry of Economic Affairs, Agriculture and Innovation.

[†] Marko van Eekelen is also affiliated with the Faculty of Management, Science and Technology, Open University of the Netherlands.

[1] `http://resourceanalysis.cs.ru.nl/energy/`

FOAL '14, April 22, 2014, Lugano, Switzerland.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2798-5/14/04. . . $15.00.
http://dx.doi.org/10.1145/2588548.2588553

1.2 Related Work

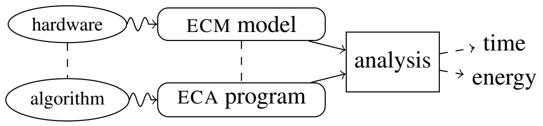
To our knowledge, ECALOGIC is the first tool that offers static energy consumption analysis for complete systems. Several tools perform a static analysis of the energy-consumption of the CPU based on per-instruction measurements, such as JOULETRACK [8] and WATTCH [2]. Furthermore, tools exist for energy profiling of software libraries, i.e. using dynamic analysis [4]. The tool that is most similar to ours is SEPROF [10]. This advanced tool combines dynamic profiling with static estimation of energy consumption. One difference is that, while ECALOGIC is geared towards complete systems, SEPROF only estimates the energy usage of the CPU. Moreover, while SEPROF *estimates* energy-usage, ECALOGIC gives *bounds* that are sound with respect to the hardware model.

In [9], an abstraction of the resource behaviour of components is presented, called Resource-Utilization Models (RUMs). Our component models can be viewed as an instantiation of a RUM. RUMs can be analysed, e.g., with the model checker Uppaal where our static analysis method employs the Hoare logic from [5]. A possible future research direction for ECALOGIC is to find a way to analyse also algorithms with RUMs as component models.

Finally, several generic resource consumption tools exist, such as COSTA [1] and RAML [3]. The difference with ECALOGIC is that these do not take a hardware model into account and are geared towards incidental resource consumption, making them less fit for energy-consumption analysis.

2. Tool Architecture

A schematic representation of ECALOGIC is as follows:



The algorithm and the hardware on which it will run must first be modelled. To capture the functionality of the algorithm, we offer the simple ECA programming language, described in Sect. 2.1. Each hardware component is modelled in a similar language, ECM, which is described in Sect. 2.2.

Component functions explicitly influence energy consumption. Other language constructs, for instance the evaluation of an arithmetic expression, also implicitly consume energy. This is modelled in the special `implicit` component. This component is assumed to be present in any system. It is modelled in ECM and therefore under full user control.

2.1 Input Language

For describing algorithms, we use the simple programming language ECA. A program is represented as a function with input parameters. The language is a simple "while"-type language, with the usual control structures and function calls. It has two major restrictions:

- All while-loops are bounded in the number of iterations. This upper bound must be specified explicitly and is assumed to be sound. It can either be inferred by a third-party tool or specified directly by the programmer.
- All variables are positive integers. There is no form of structured data. These can however be simulated by modelling them as *component functions*, as we will see below.

A partial grammar of the language is shown in Fig.1. Continuing with our running example of a wireless sensor node, a simple program that switches the radio on, takes N measurements and transmits these, looks as follows:

```
⟨program⟩ ::= {⟨comp-imp⟩ ⟨sep⟩} {⟨fun-def⟩ ⟨sep⟩}
⟨comp-imp⟩ ::= 'import' 'component' id {'.' id} ['as' id]
⟨fun-def⟩ ::= 'function' id ['(' [id {',' id}] ')'] ⟨fun-body⟩
⟨fun-body⟩ ::= ':=' ⟨expr⟩
   | ⟨stat-list⟩ 'end' 'function'
   | ⟨empty⟩
⟨stat-list⟩ ::= {⟨statement⟩ ⟨sep⟩}
⟨statement⟩ ::= 'skip'
   | id ':=' ⟨expr⟩
   | ⟨fun-call⟩
   | 'if' ⟨expr⟩ 'then' ⟨stat-list⟩ 'else' ⟨stat-list⟩ 'end' 'if'
   | 'while' ⟨expr⟩ 'bound' ⟨expr⟩ 'do' ⟨stat-list⟩ 'end' 'while'
   | '{' ⟨annot-elem⟩ {',' ⟨annot-elem⟩} '}' [⟨statement⟩]
⟨fun-call⟩ ::= [id '::'] id '(' [⟨expr⟩ {',' ⟨expr⟩}] ')'
⟨annot-elem⟩ ::= id '<-' ⟨expr⟩
⟨expr⟩ ::= ⟨expr⟩ ⟨bin-op⟩ ⟨expr⟩
   | id
   | ⟨fun-call⟩
   | '(' ⟨expr⟩ ')'
⟨bin-op⟩ ::= 'or'|'and'|'='|'<>'|'>'|'<'|'>='|'<='|'+'|'-'|'*'|'/'|'^'
⟨sep⟩ ::= ';' | end-of-line
```

Figure 1. Partial grammar of the input language ECA.

```
function alwaysOn(N)
  Radio::on()
  while N > 0 bound N do
    Value := Sensor::measure()
    Radio::queue(Value)
    Radio::send()
    N := N−1
  end while
  Radio::off()
end function
```

Here the parameter N acts as an upper bound on the number of iterations of the while loop. It is allowed to use any expression as an upper bound, as long as it can be evaluated in terms of the parameters of a function. In many cases this can be done directly, as above. If, however, the upper bound of a loop references variables whose values are only available at run time, an annotation with a Hoare-style precondition is required to relate each of those variables to the parameters. An example of this is given in Section 3.

2.2 Component Models

Hardware components models are defined by 1. a (possibly empty) set of component states, 2. a function phi which maps component states to power draw, and 3. a set of component functions. A simple model for a radio looks as follows:

```
component Radio(active: 0..1)
  initial active := 0

  component function on uses 400 time 400 energy
    active := 1
  end function

  component function off uses 200 time 200 energy
    active := 0
  end function

  component function queue(X) uses 30 time 30 energy
  component function send uses 100 time 100 energy

  function phi := 2 + 200 * active
end component
```

In this example, the radio has two states: off (0) or on (1). There are component functions to turn the radio on/off, queue a measure-

```
⟨component⟩ ::= {⟨class-imp⟩ ⟨sep⟩} 'component' id ['(' [⟨var-def⟩
        {',' ⟨var-def⟩}] ')'] {⟨member⟩ ⟨sep⟩} 'end' 'component'
⟨class-imp⟩ ::= 'import' 'class' id {'.' id} ['as' id]
⟨var-def⟩ ::= id ':' numeral '..' numeral
⟨member⟩ ::= 'initial' id ':=' numeral
    | ⟨fun-def⟩
    | ⟨comp-fun-def⟩
⟨comp-fun-def⟩ ::= 'component' 'function' id ['(' [id {',' id}] ')']
        [⟨uses-clause⟩] ⟨fun-body⟩
⟨uses-clause⟩ ::= 'uses' numeral 'energy' [numeral 'time']
    | 'uses' numeral 'time' [numeral 'energy']
```

Figure 2. Grammar of the component modelling language ECM.

ment for sending and for transmitting the queue. The function on has an incidental energy usage of 30 and changes the state of the component to active. The function phi gives the energy consumption per time-unit, depending on the state of the radio. Note that this is where the timing analysis is needed.

An important constraint on the phi function is monotonicity with respect to the ordering of the states: a higher state implies a higher energy usage. ECALOGIC checks whether this constraint holds. Apart from that, component functions have the same expressive power as the ECA language. Hence, more detailed models can easily be constructed.

3. Tool Application

To use the tool, the target platform must first be modelled. This is a complex step, as building a precise model requires measurements of the actual energy consumption. Depending on the goals of the user, educated guessing might suffice when modelling, or a standard ECM model (e.g. for the CPU) taken from a library of component models might be used. If precise results are required, accurate modelling is paramount. If the user wants to compare implementation variants, a less precise modelling will often suffice.

A typical use case is the comparison of different implementations of an algorithm. In the case of a wireless sensor node, a strategy to conserve energy is to send data packets in batches of size B, only turning the radio on right before sending.

```
function buffering(N, B)
  while N > 0 bound N/B do
    K := B
    { K <- B }
    while K > 0 and N > 0 bound K do
      Value := Sensor::measure()
      Radio::queue(Value)
      K := K − 1
      N := N − 1
    end while
    Radio::on()
    Radio::send()
    Radio::off()
  end while
end function
```

Note that the annotation { `K <- B` } is necessary to express that the symbolic value of the variable K in terms of the function parameters is B. ECALOGIC issues a diagnostic whenever the symbolic value of a loop bound or function argument cannot be determined.

Using a simple implicit component and sensor model (as in [6]), we can now compare the two implementations:

implementation	*time*	*energy*
alwaysOn(N)	$600 + 195 \cdot N$	$83600 + 40200 \cdot N$
buffering(N,B)	$\left(130 + \frac{740}{B}\right) \cdot N$	$\left(1070 + \frac{105640}{B}\right) \cdot N$

These results also provide information on appropriate values for the block size B. If we are interested in a constantly functioning sensor node, we should consider very large N. It is then clear that the buffering implementation is more efficient for $B \geq 3$.

4. Conclusion and Future Work

We have presented ECALOGIC, a tool for static energy analysis of software, and demonstrated its use by modelling a wireless sensor node. Using this tool, we can statically derive a *symbolic* upper bound on energy consumption to analyse and compare different algorithms on different hardware configurations. In the Software Analysis course at Radboud University Nijmegen students have used ECALOGIC for exercises, successfully modelling various algorithms and hardware components.

In the future we aim to further improve the tool as follows:

- Since ECALOGIC currently only supports the ECA language, the software to be analysed must be expressed in this language. When analysing existing software, this is restrictive. We are working towards supporting a well-behaved subset of C.
- We want to increase the interoperability with other analysis tools such as RESANA [7] for deriving loop bounds and a CEGAR-based tool [9] for deriving component models.
- To increase the practical applicability, we will start an open library where users can submit ECM models for hardware components. Users with access to physical measurement tools could take a model from the library and validate it.

In this way, ECALOGIC contributes to the development of a tool-supported design methodology achieving in a cost-effective way guaranteed bounds on the energy consumption of IT-systems.

Acknowledgments

We would like to thank Dorus Peelen for his valuable work on the ECALOGIC web-interface.

References

[1] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In *FMCO'07*, volume 5382 of *LNCS*, pages 113–133. Springer, 2008.

[2] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. *SIGARCH Comput. Archit. News*, 28(2):83–94, May 2000.

[3] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate amortized resource analysis. In *POPL'11*, pages 357–370. ACM, 2011.

[4] A. Kansal and F. Zhao. Fine-grained energy profiling for power-aware application design. *SIGMETRICS Perform. Eval. Rev.*, 36(2):26–31, Aug. 2008.

[5] R. Kersten, P. Parisen Toldin, B. van Gastel, and M. van Eekelen. A Hoare logic for energy consumption analysis. In *FOPARA'13*, 2014. Under submission.

[6] P. Parisen Toldin, R. Kersten, B. van Gastel, and M. van Eekelen. Soundness Proof for a Hoare Logic for Energy Consumption Analysis. Technical Report ICIS–R13009, Radboud University Nijmegen, 2013.

[7] O. Shkaravska, R. Kersten, and M. Van Eekelen. Test-based inference of polynomial loop-bound functions. In *PPPJ'10*, pages 99–108. ACM, 2010.

[8] A. Sinha and A. P. Chandrakasan. JouleTrack: A web based tool for software energy profiling. In *DAC '01*, pages 220–225. ACM, 2001.

[9] S. te Brinke, S. Malakuti, C. M. Bockisch, L. M. J. Bergmans, M. Akşit, and S. Katz. A tool-supported approach for modular design of energy-aware software. In *SAC'14*. ACM, March 2014.

[10] S.-L. Tsao and J. J. Chen. SEProf: A high-level software energy profiling tool for an embedded processor enabling power management functions. *Journal of Systems and Software*, 85(8):1757 – 1769, 2012.