

Characterizing a Source Code Model with Energy Measurements

Thibault Béziers La Fosse, Jean-Marie Mottu, Massimo Tisi, Gerson Sunyé

▶ **To cite this version:**

Thibault Béziers La Fosse, Jean-Marie Mottu, Massimo Tisi, Gerson Sunyé. Characterizing a Source Code Model with Energy Measurements. Workshop on Measurement and Metrics for Green and Sustainable Software Systems (MeGSuS), Oct 2018, Oulu, Finland. hal-01952724

HAL Id: hal-01952724

https://inria.hal.science/hal-01952724

Submitted on 12 Dec 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Characterizing a Source Code Model with Energy Measurements

Thibault Béziers la Fosse, Jean-Marie Mottu, Massimo Tisi, Jérôme Rocheteau and Gerson Sunyé
ICAM, IMT Atlantique, LS2N
Nantes
Email: first-name.last-name@ls2n.fr

***Abstract*—Energy consumption is a critical point when developing applications. Either for battery-saving purposes, for lowering the cost of data-centers, or simply for the sake of having an eco-friendly program, reducing the energy needed to run a software becomes mandatory. Model-Driven Engineering has shown great results when it comes to program understanding and refactoring. Modeling the source code along with its energy consumption could be a powerful asset to programmers in order to develop greener code. For that purpose, this paper presents an approach for modeling energy consumption inside a source code model. Energy metrics are gathered at runtime, modeled using the standard Structured Metrics Meta-model, and associated to the source code model, enabling model-driven techniques for energy analysis and optimizations.**

I. Introduction

Energy Consumption becomes a major concern when developing software. The massive electricity needs of data-centers represented 1.8% total U.S. consumption in 2016, and an estimated account for about 2% of global greenhouse gas [1]–[3]. Furthermore, the expense to power and cool the data-centers escalated to a significant cost factor: for every $1.00 spent on new hardware, an additional $0.50 is spent on power and cooling [4]. Those factors severely limits the expansion of IT resources. Reducing the consumption of software systems has thus become an important ecological, economical, and technological challenge.

Traditionally, such concerns tend to be addressed at the lower levels (hardware or middleware), and several strategies have been presented in the past years towards that purpose [5]. However, promising results have been introduced using software-level energy management approaches [6], which, combined with lower level optimizations, can effectively reduce the energy consumption of programs.

Unlike hardware-level energy saving techniques, software-level approaches often need the implication of developers. In fact, to develop a *greener* code, a certain level of energy-awareness is helpful to make relevant design choices. A study showed that more than 80% of programmers did not consider the energy consumption when developing softwares, even if they know how important it can be, especially in the realm of mobile application development [7], [8]. We believe that a representation of a source code, along with energy consumption would help developers making the right design choices.

In the realm of Model-Driven Engineering (MDE), this representation is a model, conforming to a meta-model describing its syntax and semantic [9]. Such model is typically used during the software development, as a specification model to generate a source code, or as a source code model, reverse-engineered from the software in order to analyze or refactor it.

This paper proposes a MDE approach for modeling the source code of a software and characterizing it with energy measurements. A first model of this software is generated with the static reverse-engineering framework MoDisco [10]. In addition we propose a framework dynamically gathering energy measurements, at runtime.

Those measurements are persisted in a second model conforming to the Structured Metrics Meta-model[1] (SMM). In order to characterize the source code with energy metrics, the two models are associated: elements in the source code (e.g., Java methods) are linked to the energy measurements, enabling the analysis of the source code energy consumption.

This paper is organized as follows: Section II presents related work, Section III presents our approach, Section IV discusses the approach, Section V presents the threats to the validity of our approach, and Section VI concludes the paper.

II. Related Work

A large amount of research has been done around software-level energy monitoring, optimization, and management, resulting in the production of several well-known tools. For instance in the realm of software-based energy measurements, jRAPL [6] is a Java API for profiling Java programs running on CPUs supporting Intel's Running Average Power Limit (RAPL). It can be used through a single line of Java code, and provides the energy consumed by the CPU cores. However, the energy it measures includes the energy consumed by the system, and induces a considerable overhead [11], threatening the validity of the measurements.

The following applications also perform at the software level. BitWatts is a middleware extension of the PowerAPI toolkit, able to monitor the power consumption at the process-level in VM-based systems [12], [13]. It collects process usages and estimates a fine-grained power consumption based on a power model. Jalen uses statistical sampling for estimating the energy consumption of software code [14]. Relying on PowerAPI too, it isolates selected classes and methods and

[1] https://www.omg.org/spec/SMM/

monitors them through Java agents. PTOP proposes a process-level profiling tool, running at the kernel level and monitoring the energy consumptions of all the application of the system, at a small cost, and with no performance side effects on the running applications. PTOP, BITWATTS and JALEN measure powers at a fixed frequency, making challenging the energy measurement of short-running programs. Nonetheless they rely on power models and are not impacted by the system consumption. On the contrary, JRAPL can be queried at any time (e.g. comparing the energy consumed before and after running a portion of code, in order to provide the energy it consumed), but gives raw energy consumptions, includes the system's one, and has a major impact on the accuracy of the measurements.

Those approaches estimate power and energy consumption using energy models based on system specifications and usage. However they do not make the measurements available for Model-Driven analysis tools. Our approach does not aim at improving the precision w.r.t. related work, but representing it in an holistic model, enabling further model-driven analysis.

III. APPROACH

We detail in this section the several steps necessary to characterize our source code model with energy measurements. First, the source code model is statically built using the MoDisco reverse-engineering framework. Second, the code is instrumented, in order to add probes inside the program, which is then executed to gather the energy measurements at runtime. Third, the measurements are persisted in a model conforming to the Structured Metrics Meta-model, and associated with the MoDisco source code model. The Java program in Figure 1 is used to describe our approach. Our application is available on GitHub[2].

A. Source Code Model

The MoDisco framework [10] is designed to enable program analysis, by creating a static model of a source code using reverse-engineering. Considering a source code written in Java, MoDisco generates the corresponding model conforming to a Java meta-model. This model only represents static aspects of the code, such as classes, methods, statements or expressions. The energy measurements have to be dynamically gathered, and are not available in the initial MoDisco model. Nonetheless, a benefit of MDE is that such measurements can be modeled and then associated with our source code model, as presented in the next sections. Applying MoDisco on the program in Figure 1 would produce a model containing all the source-code elements of such program. An excerpt of this model is available in Figure 2.

B. Energy Measurements Computation

Once the source code model is generated, the energy is measured. In order to get such measurements for each *method* in the program, it has to be instrumented. We use the ASM

```java
package mainPackage;

public class App {

    public static void main(String[] args) {
        foo();
    }

    public static void foo() {
        //do something
    }

}
```

Figure 1. Simple Java program

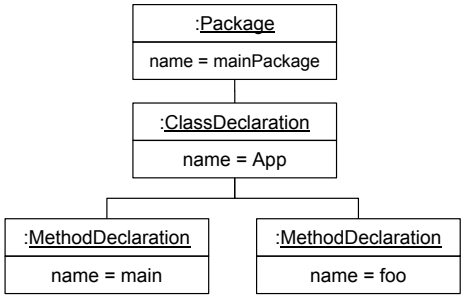


Figure 2. Excerpt of the MoDisco source code model

instrumentation library[3] for that purpose. The JVM byte-code is visited, and probes are added in the byte-code of every method of the program. A first probe is added in every method entry point, traces down the method call, gets the system time, and finally gets the energy consumed by the CPU at the specific moment, in microjoule.

Another probe is added at every method exit point. This second probe computes the duration of this method execution, and the energy it has consumed. To do so, the energy consumed by the CPU is fetched a second time, and subtracted to the value obtained in the first probe. Our energy measurements are performed using the same approach as `jRAPL` [6].

As an example, considering the Java source-code available in Figure 1, the byte-code generated, and its instrumented version would be the ones in Figure 3 and Figure 4. Figure 3 contains the byte-code of the `main` as method described in Figure 1. This byte-code simply embeds a static invocation of the `foo` method. Figure 4 also embeds this static invocation of the `foo` method, however, probes have been added at the entry and exit points of the method. The probe located at the entry point of the method is called by instantiating a class named `Probe`. The constructor of this class takes as a parameter the qualified name of the method it located in, and computes the energy consumed before the execution. Once the exit point of the method is reached, the `exit` method of the probe is called. This method gets the energy consumed, compares it with the value obtained at the entry point, and traces that value.

Note that if a second method is called inside a first method, then the energy consumed by this first method includes the

[2]https://github.com/atlanmod/EnergyModel

[3]https://asm.ow2.io/

```
public static void main(java.lang.String[]);
descriptor: ([Ljava/lang/String;)V
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=0, locals=1, args_size=1

0: invokestatic  #2    // Method foo:()V
3: return
```

Figure 3. Non-instrumented byte-code of the main method

```
public static void main(java.lang.String[]);
descriptor: ([Ljava/lang/String;)V
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=3, locals=2, args_size=1

0: new           #11  // class Probe
3: dup
4: ldc           #25  // String mainPackage.App$main
6: invokespecial #16  // Method Probe."<init>":[...]
9: astore_1
10: invokestatic  #28 // Method foo:()V
13: aload_1
14: ldc           #25 // String App$main
16: invokevirtual #19 // Method Probe.exit:[...]
19: return
```

Figure 4. Instrumented byte-code of the main method

energy consumed by the second one. For instance, if we consider the code in Figure 1, the energy consumed by the `main()` method will include the energy consumed by the `foo()` method. Hence it could be possible to approximate the energy consumed only by `main()` by subtracting the energy `foo()` consumed, thanks to the available energy model.

Running the instrumented code triggers the probes to gather measurements. For each method, the following metrics are obtained:

1) Its timestamp
2) The energy it consumed
3) The methods that have been called inside.

Running the code can be done either by executing the main method of a program, or through test cases. During the execution, the measurements are published on a messaging queue, in order to be analyzed after the execution. Indeed, analyzing the traces is an expensive operation, and performing it after the execution limits the overhead induced by the execution of the instrumented statements.

Furthermore, our approach is relying on existing energy measurement tools, and do not aim at improving their accuracy. Performing the measures causes an important workload, limiting the accuracy of the energy measurements [15]. For that reason, we can only provide the energy consumed at the method level, or at coarser granularities.

C. Energy Measurements Modeling

In this main step of our approach, we persist the energy measurements into a model conforming to SMM and associate it to the source code model that MoDisco generates. This is performed by reading the measurements sent in the messaging queue one by one. Energy consumed, timestamps and internal method calls are persisted as elements in the model. Finally the measurements are linked to the methods from the source-code level, hence characterizing those.

In Figure 5, the energy consumed is contained in the `Measurement` elements, as values. The method calls are represented using the `MeasurementRelationship`. The "before" and "after" links point towards the `Measurement` elements, describing in which order the methods have been called. Here for instance, `main()` calls `foo()`.

Furthermore, the timestamps of the methods invocations are available in the `ObservedMeasure` elements. Finally the source code model elements (e.g. `MethodDeclaration`) are associated to the `Measurement` using the link labeled as `measurand`.

IV. Discussion

This source code model characterized with energy measurements offers developers a better energy-awareness than standard software engineering techniques: First, an holistic model of the program is created. The association between MoDisco's Java meta-model and SMM offers a representation of the source code characterized with several dynamic information, available for analysis purposes. Second, the program can be refactored and optimized only by working at the model level.

Programmers can specify transformation rules for refactoring [16], without having to go back to the source code level, in order to optimize the energy consumption using existing energy-efficient practices [17]–[20]. As stated by G. Pinto et al., refactoring approaches focusing on improving the energy consumption of programs are lacking [21]. Our model could help programmers refactoring their applications towards that purpose, by specifying a model transformation to apply to the source-code model, and generating the program optimized [22]. The energy measurements present in the model can be used as predicates for the transformation. For instance, a model transformation could be specified, only targeting the methods that consume more than a specified energy. This transformation would change all the existing Doubles to Floats, as it has been shown that it can reduce the energy consumption [6]. The modified source code of the program can then be generated again, using MoDisco code generator, and be used to consume less energy.

Furthermore, our model can easily be used by external model-driven tools. The fact that this model is persisted with the Eclipse Modeling Framework's XMI standard enables an important re-usability, especially within the Eclipse environment [23]. As an example, the *Sirius* Framework is built on top of the Graphical Modeling Framework (GMF), and enables the editing, manipulation and creation of models through custom

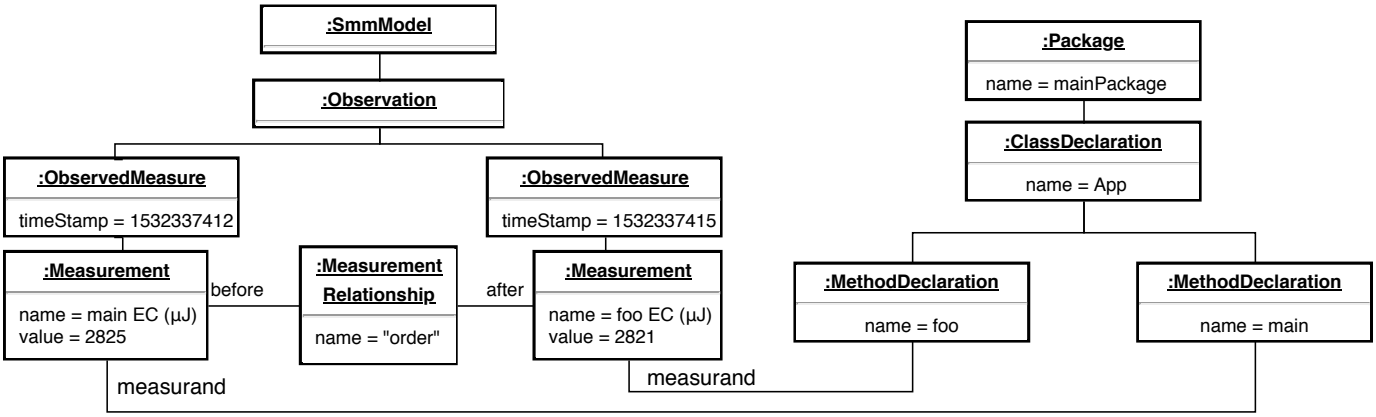


Figure 5. Excerpt of the source code model (the `MethodDeclaration` instances) characterized with energy measurements

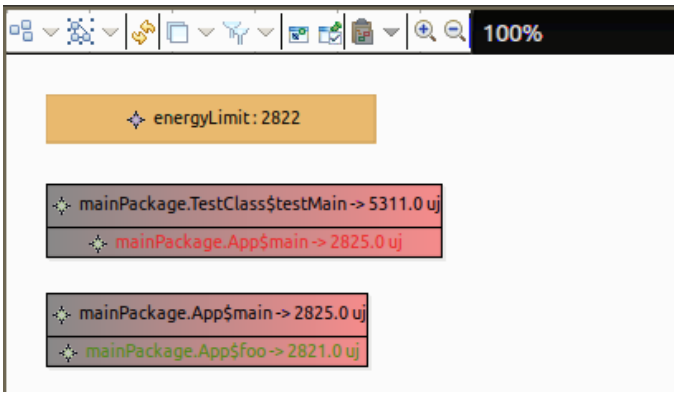


Figure 6. A graphical representation of the Energy model inside *Sirius*

graphical interfaces [24]. In order to help developers locating energy consuming method, we used *Sirius* to build a simple graphical representation of the source code, highlighting the methods that consume more than a specified threshold. A screen-shot of such application is available in Section IV. This figure shows the energy consumed by the `main` and `foo` methods, as presented in Figure 5, and the energy consumed by a test case launching the `main` method. An energy limit has been set to 2822µJ, hence highlighting all the methods consuming more than that threshold. This example shows that this model can be used in order to provide developers feedbacks about the energy consumption of their programs, and hence help them doing the right design choices, and improving their energy awareness.

V. Threat to validity

Several points have to be considered, in regard to the validity of our approach. First of all, our current model does not aim at containing the precise energy consumption measurement for each method. It stores in a standard model the information we gather from our measuring tools. In our implementation we used RAPL to measure the energy consumed by the program. RAPL induces an overhead, that has not been quantified here, and which threatens the accuracy of the measurements. Furthermore, RAPL also includes the energy that the exploitation system is consuming. For that reason, the energy measurements we are able to gather with our current tooling might not be accurate. Finer energy estimations will be done later over this model.

Furthermore, MDE suffers from some well-known scalability issues. Our model is persisted using the XMI standard, however, as explained by Javier Espinazo-Pagán et al., XMI files cannot be partially loaded [25]. For that reason the XMI persistence layer tends to scale badly when the models are too large: in order to be used, the entire model have to fit entirely in memory. We encountered this issue when generating the MoDisco model of Java projects having more than 10,000 classes. This well-known scalability issue can be resolved using scalable persistence layers for EMF models, such as CDO[4] or NeoEMF [26]. Our current prototype does not rely on those scalable persistence layers yet.

VI. Conclusion and Future Work

This paper presents an approach for modeling the source code of an application and decorating it with energy measurements. Those measurements are dynamically gathered, modeled using SMM, and associated with a source code model statically generated by MoDisco. This model can be used by programmers as a basis for analyzing and optimizing programs, and offers a better energy-awareness, useful for implementing greener applications.

In our future work, we plan to refine the energy measurements gathered in our approach. In fact, measuring the energy consumed by the program using RAPL induces an overhead and includes the entire system energy consumption. Furthermore, instead of using XMI for persisting our models, a more scalable persistence layer, such as NeoEMF, will be used.

References

[1] G. Cook, "How clean is your cloud," *Catalysing an energy revolution*, p. 11, 2012.

[4]https://www.eclipse.org/cdo/

[2] M. Webb *et al.*, "Smart 2020: Enabling the low carbon economy in the information age," *The Climate Group. London*, vol. 1, no. 1, pp. 1–1, 2008.

[3] A. Shehabi, S. Smith, D. Sartor, R. Brown, M. Herrlin, J. Koomey, E. Masanet, N. Horner, I. Azevedo, and W. Lintner, "United states data center energy usage report," 2016.

[4] M. J. Scaramella and M. Eastwood, "Solutions for the datacenter's thermal challenges," *IDC, January*, 2007.

[5] S. Mittal, "A survey of techniques for improving energy efficiency in embedded computing systems," *International Journal of Computer Aided Engineering and Technology*, vol. 6, no. 4, pp. 440–459, 2014.

[6] K. Liu, G. Pinto, and Y. D. Liu, "Data-oriented characterization of application-level energy optimization," in *International Conference FASE'15*. Springer.

[7] I. Manotas, L. Pollock, and J. Clause, "Seeds: a software engineer's energy-optimization decision support framework," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 503–514.

[8] C. Pang, A. Hindle, B. Adams, and A. E. Hassan, "What do programmers know about software energy consumption?" *IEEE Software*, vol. 33, no. 3, pp. 83–89, 2016.

[9] J. Bézivin, "On the unification power of models," *Software & Systems Modeling*, vol. 4, no. 2, pp. 171–188, 2005.

[10] H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot, "Modisco: a generic and extensible framework for model driven reverse engineering," in *Proceedings of the IEEE/ACM International Conference ASE'10*. ACM.

[11] H. Zhang and H. Hoffman, "A quantitative evaluation of the rapl power control system," *Feedback Computing*, 2015.

[12] A. Bourdon, A. Noureddine, R. Rouvoy, and L. Seinturier, "Powerapi: A software library to monitor the energy consumed at the process-level," *ERCIM News*, vol. 2013, no. 92, 2013.

[13] M. Colmant, M. Kurpicz, P. Felber, L. Huertas, R. Rouvoy, and A. Sobe, "Process-level power estimation in vm-based systems," in *Proceedings of EuroSys'15*. ACM.

[14] A. Noureddine, R. Rouvoy, and L. Seinturier, "Unit testing of energy consumption of software libraries," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. ACM, 2014, pp. 1200–1205.

[15] S. a. Rivoire, M. A. Shah, P. Ranganathan, and C. Kozyrakis, "Joulesort: a balanced energy-efficiency benchmark," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. ACM, 2007, pp. 365–376.

[16] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[17] R. Pereira, M. Couto, J. Saraiva, J. Cunha, and J. P. Fernandes, "The influence of the java collection framework on overall energy consumption," in *GREENS'16*. ACM, 2016.

[18] G. Procaccianti, H. Fernández, and P. Lago, "Empirical evaluation of two best practices for energy-efficient software development," *Journal of Systems and Software*, vol. 117, pp. 185–198, 2016.

[19] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "Enerj: Approximate data types for safe and general low-power computation," in *ACM SIGPLAN Notices*, vol. 46, no. 6. ACM, 2011, pp. 164–174.

[20] W. G. da Silva, L. Brisolara, U. B. Correa, and L. Carro, "Evaluation of the impact of code refactoring on embedded software efficiency," in *Proceedings of the 1st Workshop de Sistemas Embarcados*, 2010, pp. 145–150.

[21] G. Pinto, F. Soares-Neto, and F. Castor, "Refactoring for energy efficiency: a reflection on the state of the art," in *Proceedings of the Fourth International Workshop on Green and Sustainable Software*. IEEE Press, 2015, pp. 29–35.

[22] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "Atl: A model transformation tool," *Science of computer programming*, vol. 72, no. 1-2, pp. 31–39, 2008.

[23] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*. Pearson Education, 2008.

[24] V. Viyović, M. Maksimović, and B. Perisić, "Sirius: A rapid development of dsm graphical editor," in *Intelligent Engineering Systems (INES), 2014 18th International Conference on*. IEEE, 2014, pp. 233–238.

[25] J. E. Pagán, J. S. Cuadrado, and J. G. Molina, "Morsa: A scalable approach for persisting and accessing large models," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2011, pp. 77–92.

[26] G. Daniel, G. Sunyé, A. Benelallam, M. Tisi, Y. Vernageau, A. Gómez, and J. Cabot, "Neoemf: a multi-database model persistence framework for very large models," *Science of Computer Programming*, vol. 149, pp. 9–14, 2017.