Unit Testing of Energy Consumption of Software Libraries

Adel Noureddine, Romain Rouvoy, Lionel Seinturier

Unit Testing of Energy Consumption of Software Libraries

Adel Noureddine[1,2], Romain Rouvoy[1,2] and Lionel Seinturier[1,2,3]
[1] Inria Lille – Nord Europe
[2] University Lille 1 - LIFL CNRS UMR 8022, France
[3] Institut Universitaire de France
firstname.lastname@inria.fr

ABSTRACT

The development of energy-efficient software has become a key requirement for a large number of devices, from smartphones to data centers. However, measuring accurately this consumption is a major challenge that state-of-the-art approaches have tried to tackle with a limited success. While monitoring applications' consumption offers a clear insight on where the energy is being spent, it does not help in understanding how the energy is consumed. In this paper, we therefore introduce JalenUnit, a software framework that infers the energy consumption model of software libraries from execution traces. This model can then be used to diagnose application code for detecting energy bugs, understanding energy distribution, establishing energy profiles and classifications, and comparing software libraries against their energy consumption.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Energy, Modeling, Benchmarks

Keywords

Energy measurement, Power modeling, Software metrics, Empirical benchmarking

1. INTRODUCTION

While the global rise of energy costs and its predicted growth for the next 20 years [10] present motivations for energy efficiency and optimizations, technological advances—in particular in the domain of *Information and Communications Technology* (ICT)—hold important keys for achieving energy efficiency. Although ICT helps to reduce the energy consumption of other sectors [15], its power consumption is predicted to rise from 7% of the global power consumption to more than 14.5% in 2020 [14]. Therefore, optimizing the energy consumption of ICT, or Green IT, is an economical necessity and a technological challenge. This means that not only the hardware, but also software also needs to become greener in a near future. Measuring the energy consumption of software, at runtime or by using profilers, allows users and developers to acknowledge the energy cost of their applications. However, these approaches are specific to the execution context used to monitor the energy consumption. They do not provide insights into the energy consumption variations, therefore limiting these approaches to debugging, software profiling or understanding the execution trace of software. We argue that having an energy evolution model of software code, based on their input parameters, offers new metrics and models for modeling energy consumption and assisting developers to use software libraries based on their predicted input parameters.

In this paper, we propose an approach and a toolkit to automatically infer the energy models of software libraries based on their input parameters. We present our energy unit-testing framework, named JalenUnit, for inferring automatically the energy consumption model of software libraries according to input parameters. The framework takes a Java library and cycles through all its packages, classes and methods, and runs energy variation benchmarks on each of its accessible methods.

The remainder of this paper is organized as follows. In Section 2, we describe our motivations and outline the limitations of the state-of-the-art approaches. We present JalenUnit in Section 3, our framework for inferring automatically the energy consumption model of software libraries. In Section 4, we validate our framework with experimental benchmarks on a RSA algorithm, Google Guava and the Violin String libraries, and we discuss our results and our approach in Section 5. Finally, we conclude in Section 6 and outline future work.

2. MOTIVATIONS & RELATED WORKS

State-of-the-art approaches in this area offer to measure or estimate the energy consumption of computers and applications using various techniques. Techniques varies from using hardware circuits, such as dedicated ASICs [6] (*Application Specific Integrated Circuit*) or sensors and smart meters as in [13] or as in data centers [4]; hardware powermeters or multimeters such as with PowerScope [3] or Intel's Energy Checker; or coarse-grained software only estimation of energy consumption as in [11, 12, 5, 2]. Some hardware-

SAC '14 March 24-28, 2014, Gyeongju, Korea.

based approaches, such as a powermeter, limit their usability to either a research prototype, or coarse-grained hardware monitoring (as in data centers).

In previous works, we developed power models in order to estimate the energy consumption of software [8, 7]. Two components compose our approach: PowerAPI for monitoring software, and Jalen for detecting energy hotspots in software code using byte code instrumentation of Java applications. We compared the energy consumption of several programming languages, and of different algorithm implementations [8]. We then proposed our approach to detect energy hotspots in software at code level—*i.e.*, methods and classes [7]. However, our approach is limited into measuring the energy consumption of software code, and does not provide variation models. Measuring the energy consumption of software is not but one step into producing energy efficient code. The energy information reported is static, *e.g.*, values are related to an execution of software in one specific configuration. Changing a parameter in a method or modifying an input parameter therefore requires a new execution of the application in order to get the new energy consumption and the impact of this change. Thus, what if developers had tools to empirically measure the energy consumption of their software code, and get empirical data about the energy variation trends in their code? And also get the impact of changing input data and parameters on the energy consumption of methods? These data can be used to diagnose the code to further understand the energy distribution across the application, or establish an energy variation profile or classification of software based on their input parameters. One additional motivation of automating these measurements relates to software libraries. The latter are used by other software and therefore improvement in their energy efficiency would benefit a large pool of applications. Benchmarking libraries for their energy consumption and proposing empirical models of the variation of their energy consumption are *win-win* situations for software developers.

In this paper, we propose JalenUnit, an energy framework for modeling the energy variation of software code based on their input parameters. JalenUnit uses both PowerAPI and Jalen as underlying energy measurement tools, and generates then run energy benchmarks for all the methods available in an application or a library. Finally, it analyzes the results and constructs an energy variation model of software code.

3. JALEN UNIT FRAMEWORK

JalenUnit is an energy framework that generates energy models for software code from empirical benchmarks. Next, we describe our approach, and the implementation of JalenUnit.

3.1 Approach

Our approach models the energy variation trend of a software method by running benchmarks on a method while changing its parameters. Concretely, we provide the energy variation model of a method based on the variation of its parameters. This provides a relational table between methods and their energy model, therefore allowing developers to choose the most energy efficient method for their software. In details, we vary the value of the input parameters of methods and measure their energy consumption using each of these values. In the end, we collect the energy consumption of the method for each value of its parameters, therefore allowing us to have an energy variation profile of the method.

3.1.1 Jalen: Measuring Energy Consumption

In order to measure the energy consumption of each method while varying its parameters, we use a new version of our code-level measurement tool, Jalen. This new version uses statistical sampling in order to estimate the energy consumption of software code, rendering the tool usable in production (in comparison to a high overhead to the previous byte code instrumentation version). Jalen is also capable of isolating selected classes or packages along measurements. Therefore, this allows measuring the energy consumption of software libraries without disturboing the *Java Virtual Machine*'s (JVM) or the application's classes and methods. Jalen, is a Java agent that hooks during the JVM launch, and starts monitoring and collecting energy-related information of the software. Technically, Jalen uses PowerAPI to get the energy consumption of the software [7, 8]. Then, using this information, it monitors the software and correlate energy information to portions of code—*i.e.*, methods in our case. Information related to threads and CPU time are collected from the JVM, while disk accesses are estimated per-method using statistical sampling (by filtering methods associated to the `java.io` and `java.nio` packages).

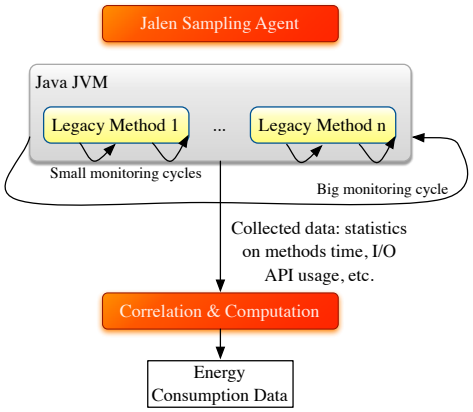


Figure 1: Jalen approach for measuring energy consumption of software code.

Our measurement methodology follows a two-cycle approach and is reported in Figure 1. First, a *big* monitoring cycle where power consumption of software is gathered from software monitoring using PowerAPI; and then a *small* monitoring cycle where statistical information is collected on each running method. For each *big* monitoring cycle, Jalen calls PowerAPI in order to get the energy consumption of the application as a whole. During the *small* monitoring cycle, we collect the number of times a method appears in our statistical sampling (measured at a higher frequency). For example, two method `AT` and `BT` are executing for 10 seconds. The *big* cycle is at 1 second and the *small* cycle is at 10 milliseconds. The method `AT` is captured 7 times during the *small* cycle while `BT` is captured 3 times. Each of these methods have different execution times and CPU utilization, therefore both methods are scheduled and executed accordingly (for example, method `BT` waits for a network answer,

thus the JVM executes `AT` during the wait). We then correlate theses statistics with the CPU time of threads (gathered from the JVM), in order to estimate the energy consumption of methods.

3.1.2 Jalen Unit: Benchmarking Energy Variations

JalenUnit is our energy framework that generates energy models for software code based on empirical benchmarks. It provides benchmarks for modeling the energy consumption of software methods through automatic benchmarking.For instance, it generates individual benchmarks for each method in a software library, and for each of its parameters. These benchmarks stress the method based on a set of input values for its parameters. These values are determined through different injectors, and multi-parameters methods are managed through different strategies. Next, all generated benchmarks are executed. For each, we measure its energy consumption, then the results are aggregated and analyzed to produce the method's energy profile and variation model. Concretely, JalenUnit cycles through every package, class, and method in a Java library. For each method and each of its parameters, an energy benchmark is created following a variation strategy for the benchmarked parameter. The benchmark is then executed and Jalen is used to measure its energy consumption. Finally, energy data for the benchmark and the variation of parameters is reported in an output file that is later plotted as a graph.

JalenUnit is built as a Java application that loops over all methods in a Java software library and generates energy benchmarks. The latter are then executed and their energy consumption is measured using Jalen automatically. In details, JalenUnit generates and runs a benchmark for each method while varying its parameters. This variation of parameters is done through injectors implemented for Java primitive and object types. The framework can, therefore, be extended with application-specific injectors describing alternative variation strategies. Java objects can be benchmarked automatically if their injector model is implemented in the framework.

The initial implementaiton of JalenUnit provides injectors for primitive types: `Integer`, `Double`, `Long`, `Float`, `Boolean`, and `Character`, in addition to the `String` class. We prefer to implement our own injector instead of using existing injectors, such as YETI [1] which performs random testing, because we want to provide different strategies for benchmarking and testing methods. This provides a good strategy for detecting abnormal behavior in software code, such as exceptions or huge CPU load for certain values. However, it does not offer a comprehensive strategy for evaluating the energy variation of methods by input parameters. For example, we develop an injector for integers where the integer values evolve with an increment, from a start value to a final value (*e.g.*, integer values from 10 to 100 with a hop of 10 leads to 10 benchmarks with values of 10, 20, ... to 100). Another injector for integers evolves the integer randomly using the `Math.random` method in Java. Although integers are all of the same size, changing their value impacts the execution of methods, therefore their energy consumption. For example, an integer parameter that is used as a final value to a for-loop may have a high impact because increasing its value implies that tasks are being executed for longer period of time and consuming more energy.

Injectors for other types also implement different variation strategies, such as varying the length of a string parameter randomly, or from a start value to a final value, or choosing the characters of the string from a subset of the alphabet. The variation strategies are endless, and offer the advantage of better flexibility and extendibility of the framework. This flexibility is also useful for domain specific applications, where random testing is not representative of the *real world* workload. By providing an extensible framework and providing freedom of choice in method variation model strategies, we propose a solution that can be customized for specific needs. Therefore, better representative energy variation models can be empirically achieved. Concretely, an injector is a Java class implementing the `Iterator` and `EnergyModel` interfaces. The latter adds additional methods to the iterator `next` and `hasNext` methods, such as a `getDefaultValue` method that returns an object of a default value of the injector. The following listing provides an excerpt of code of the default integer injector (syntax modified and shortened for space concerns):

```
public IntegerModel(int start, int end, int inc);
public boolean hasNext()
        return this.current <= this.end;
public Object next() {
        int result = this.current;
        this.current += this.inc;
        return result;
}
public Object getDefaultValue()
        return this.start;
```

Multi-parameters methods are managed by varying one parameter at a time, while the others use a default value. Others strategies are possible, such as varying multiple parameters while fixing the values of some, or modifying all parameters randomly. We are aware that more comprehensive strategies are required for a refined energy variation model, therefore our multi-parameters strategy is just an initial implementation for handling the complexity of multiple parameters. Benchmarks are then run and their energy consumption is measured using Jalen. Finally, the generated energy results are aggregated and the energy variation model of method is inferred.

4. EXPERIMENTATIONS

We illustrate and validate our approach by modeling the energy variation of an RSA algorithm, and of the `Joiner.join` method taken from the Google Guava's library [1].We then run our JalenUnit framework on the Violin Strings Java library [2]. This library is a collection of 138 methods designed for manipulating strings. It extends the functionalities of Java's `String` class by offering methods usually found in other programming languages, such as C++. The methods of the library use different input parameters: strings, characters, integers, or booleans. We use our default injectors for these types. In particular, the strings injector model injects strings with different sizes, ranging from a length of 100 up to 1,000 characters with a hop of 200. The integer, float, double and long injectors' models inject numbers ranging from 100 to 1000 with a hop of 200. The character injector model injects a random character of the 26 characters of the English alphabet. Finally, the boolean injector model injects both *true* or *false* values. Experimentations are done on a

[1]https://code.google.com/p/guava-libraries/
[2]http://www.schmeling-consulting.de/violinstrings.html

Dell OptiPlex 745 with an Intel Core 2 Duo 6600 processor at 2.40 GHz and running Lubuntu Linux 13.04, version 1.6 of PowerAPI, the statistical version of Jalen, and Java 7. Energy data are computed each 500 ms and the sampling interval is set to 10 ms.

4.1 RSA Encryption/Decryption

We take an RSA asymmetric encryption/decryption algorithm [9] and measure its energy consumption while varying the length of the RSA public and private keys. RSA algorithm is an example of an algorithm whose input parameters (the RSA key) impact the functionality of the said algorithm, *e.g.*, in term of security, robustness of encryption, and speed of encryption/decryption process. The algorithm generates an RSA key, then encrypts and decrypts 10 times a random `BigInteger` with a bit length of 10,000. The results, in Figure 2a, show an exponential rise in the energy consumption of the RSA algorithm when increasing the RSA key length. However, we want to understand which portion of the code is responsible for the exponential increase in the energy consumption. Results at the code level, in Figure 2b, show that two methods, in the `java.math` package, are responsible for the majority of the energy consumption: `BigInteger.oddModPow`, and `BigInteger.montReduce`. From these methods, `oddModPow` have a clear exponential increase, while `montReduce` follows a logarithmic growth. The exponential energy variation in Figure 2a is explained and identified in Figure 2b when benchmarking at the code level. It allows us to assess that `oddModPow` method is the culprit of the exponential evolution as it is the only method having an exponential variation and consuming around 80% of the total energy. RSA encryption/decryption algorithm is an exponential one as described in [9]. Therefore, our experiment results provide additional validation to our measurement approach. In particular, the method responsible for the exponential growth in energy consumption in our implementation of RSA algorithm is the method that does the exponential calculation, `oddModPow`.

4.2 Joiner.join method

We infer the energy variation model of Google Guava's `Joiner.join` method. When joining 2 strings, the method `join` calls 18 times other methods and constructors of the Google Guava library, in particular the method `appendTo`. We use version 14.0.1 of the library, and stress the method `join` of the class `com.google.common.base.Joiner` by varying the number of strings to join (from 2 to 50 strings) while fixing the string size (*i.e.*, 100 characters). We generate a random string that we use during the join call. We run the join stress one million times with the generated string, and repeat the stress 10 times with different strings of the same size. Finally, we record the energy consumption of the overall execution. The energy consumption results (cf. Figure 2c) show that the method `Joiner.appendTo` consumes most of the energy (going from 97.14% to 99.36% of the overall energy consumption). The energy variation alternates phases of constant energy consumption with others of direct increase. These numbers are explained by the implementation of the method `appendTo`. It cycles through the strings to join (given in parameter as an iterable collection) and appends it to an *appendable* object also given in parameter. `Appendable.append` finally preforms the append of the two or more strings. When varying the number of strings, the JVM is required to allocate memory for these strings. The strings in the string builder object are stored as an array of characters, and the JVM doubles the size of the array (until the new characters fits in the array) when appending new characters exceeding the initial size of the array [3]. By default, the buffer size is 8192 characters in the JVM. Our experiment is run 10 times therefore reaching the limits of the buffer when joining 8 strings of 100 characters each (totaling more than 8000 characters). When the limit is reached, the JVM doubles the buffer allocation allowing more memory for joining the strings. This explains the burst of energy consumption when joining 9 strings in Figure 2c. The joining of the strings has stable energy consumption and bursts of energy occur when the JVM needs to increase its buffer. This allocation occurs at a doubling interval, thus after joining 8, 16, 32, 64, *etc.* strings.

4.3 Violin String Library

Violin String is a Java library for manipulating strings that extends the functionalities of Java's `String` class. Next, we report the results over two representative methods of the library (in terms of parameters number and variation). The benchmark runs each method a different amount of times in order to get enough execution traces for estimating the energy consumption.

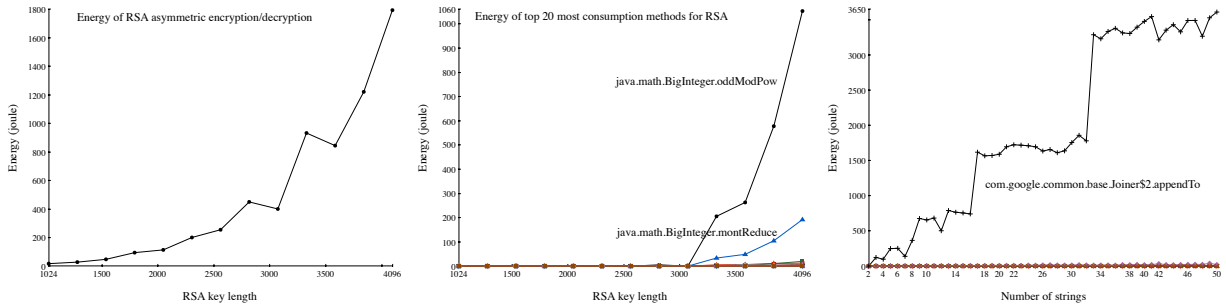
4.3.1 Two-parameter method: Copies

The `copies` method takes an input string and an input number, *nCopies*, and generates a string consisting of *nCopies* of the input string. The benchmark runs the `copies` method 100,000 times. This is due to the limitation of the underlying PowerAPI monitoring cycle, where a minimum cycle of 500 milliseconds is required for accurate measurements. Results depicted in Figure 3 show a clear linear variation of the energy consumption when varying the size of the string (while fixing the number of copies to 100), and when varying the number of copies (while fixing the size of the string to 100 characters). In details, the method first creates a string buffer of a size equal to *nCopies* $\times$ size of the string. Then, it appends the string *nCopies* times in a for-loop using Java's `append` method. The latter calls the method `String.getChars`, which in turn invokes `System.arraycopy` (that finally performs the copy). The code of the method `copies` explains the energy consumption while evolving the size of the string to copy. In particular, a larger string requires more energy to append it to the `StringBuffer` object (thus a bigger loop over the characters array to copy). And a higher number of copies means the same repetitive task is executed multiple times, therefore the energy evolves linearly.

4.3.2 Three-parameter method: Translate

The method `translate` *converts all of the string's characters which are contained in the input set of characters to the corresponding character in the output set of characters.* The benchmark runs the method 1,000 times. The results in Figure 4 report the energy progression when varying the first three parameters of the method. We notice that the model is linear when varying the length of the input string (first parameter), and the length of the input set of characters (second parameter, *setin*). The model is constant when varying

[3] http://www.javamex.com/tutorials/memory/string_buffer_memory_usage.shtml



(a) Energy variation model of RSA algorithm.

(b) Energy variation model of RSA methods.

(c) Energy variation model of the `join` method.

Figure 2: Evolution of the energy consumption of RSA asymmetric encryption/decryption according to key length, and Guava's join method according to the number of strings.

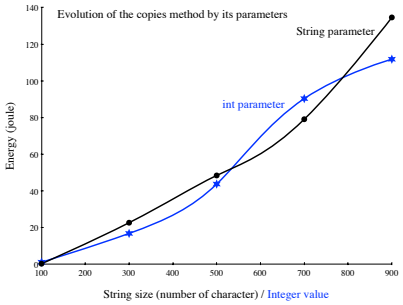


Figure 3: Energy variation of `copies` by its parameters.

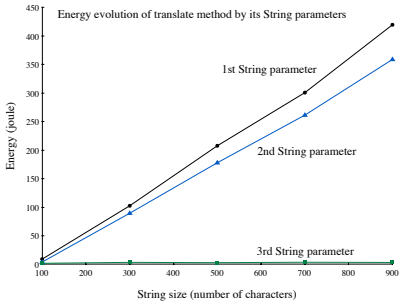


Figure 4: Energy variation of `translate` by its 3 parameters.

the length of the output set of characters (third parameter, *setout*). In particular, the third parameter, *setout*, is used only twice in the method: once to get its length, and another time to get a character at a given position in an if/else condition. Both usages are relatively simple to execute, therefore consuming little energy, thus explaining the low impact of varying this parameter and the *flat* variation of the energy consumption. On the other hand, varying the string to convert (first parameter) or the input set of characters (second parameter) has a linear impact on the energy consumption. The method `ViolinString.indexOfAnyOf` is invoked upon the first and second parameters, and the method `String.getChars` is called upon the first one too in the implementation of the `translate` method. `indexOfAnyOf` implementation also calls `String.getChars` on the *setin* parameter. The latter method uses `System.arraycopy` in order to copy an array of objects (*e.g.*, here an array of characters), and is responsible for linear energy variation as we reported in Section 4.3.1 with the `copies` method.

5. DISCUSSIONS

Our work on modeling the energy consumption variation of software code based on input parameters provides us an additional layer of understanding of the energy consumption and distribution in software. But also, it assists us with methodologies and tools to assess the impact of input parameters on the energy consumption. Our results show higher complexity in the distribution of energy in software code, the importance of taking into account the implementation of Java's JVM, and the side effects that may happen. The learnings we got from our work are summarized in the next paragraphs.

5.1 Model energy variation through empirical benchmarking

The first conclusion of our work is that we can model the energy consumption variation of software code through empirical benchmarking. In our approach, we show the validity of empirical benchmarking when studying the variation of the RSA asymmetric encryption/decryption algorithm (see Section 4.1). The energy consumption variation is exponential and is on par with the exponential complexity of the algorithm. In addition, monitoring the variation at software code allows detecting the methods responsible for the said variation, and the implementation source code of these methods validates our approach.

5.2 Impact of Java's JVM and system calls

Our experiments show the impact of the JVM characteristics, core methods and implementation, and those of system calls. In the `Joiner.join` experiment in Section 4.2, the variation in energy consumption is explained by the need for the JVM to allocate more momory for appending strings. In addition, our results on the Violin Strings library (cf. Section 4) show how the variation of string manipulation methods is impacted by the JVM's methods it calls. The library's `copies` method uses Java's `append` method, which in turn calls `String.getChars`. The latter finally uses `System.arraycopy` to perform characters' copies. Therefore, a larger string to copy requires more loops over the string characters in `System.arraycopy`, leading to linear energy

variation based on the string size. `Translate` method is another example where both the method's own implementation and JVM's methods have an impact on the energy variation modeling. One of its parameters, the string *setout*, has little impact on the energy variation. This is because it is used in a context where the variation of its size has negligible impact on the performance and complexity, thus energy consumption, of the method. However, two other parameters, strings *s* and *setin*, have linear impact because the execution of the method ultimately calls `System.arraycopy` on these parameters. Our results show the importance of the underlying Java JVM implementation and performances in order to better understand the energy consumption, distribution and variation in software. Providing energy efficient software is therefore depending on this knowledge and on the lessons we learnt from our approach and experimentations.

5.3 Limitations and Future Directions

JalenUnit framework allows modeling energy consumption variation of software code. Our results are promising into understanding energy interaction in software, however some limitations are to be noted and tackled in future works. Our framework benchmarks methods individually; therefore interactions between methods and their side effects are not studied here. The impact of varying the parameters on other methods is an interesting topic we will be addressing as a future direction. In addition, our model is inferred through empirical benchmarking but its mathematical analysis and notation is still manual. Automatic analysis of the empirical data and the inferring of the mathematical $\mathcal{O}$ notation, and specific variation formulas are the next direction in our work. Ideally, a mathematical formula would provide the energy consumption of a method based on the values of its parameters. Finally, our framework infers energy variation models based on CPU energy. However, more hardware components are involved in the execution of software, in particular the disk, memory and network. We are currently expanding our framework into expanding the variation model into these components. Our previous work in measuring energy consumption shows that network energy is negligible compared to the CPU energy [7]. We also acknowledge in our preliminary experiments that disk and memory energy accounts for a considerable part of energy consumption in our configuration machines. Therefore, our main future direction is studying the effects of these hardware components on the energy variation model of methods while varying their parameters.

6. CONCLUSION AND FUTURE WORK

In this paper, we present the JalenUnit framework. It allows gathering energy consumption data of software code while varying the input parameters of the latter code. The framework builds on automatic benchmarking of software methods, and on detecting the energy variation trends based on parameters variation. Therefore, it allows better understanding of software energy consumption and provides insights on the variation of energy consumption of software code. This prediction model is useful for software developers to detect energy bugs, understand energy distribution, establish energy profiles or classifications, or compare software libraries against their energy consumption. We demonstrate the usability of our approach using empirical experimentation on various Java software libraries.

As a matter of future work, we plan to: *i)* improve the usability of our JalenUnit framework with more injectors, better support for class dependencies, and more strategies for multi-parameters methods; *ii)* extend our framework and prediction models for different programming languages, in particular for other virtual machine based languages; *iii)* integrate inferred energy models in development environments, therefore allowing our framework to be used also during development.

7. REFERENCES

[1] York Extensible Testing Infrastructure. https://code.google.com/p/yeti-test/.

[2] T. Do, S. Rawshdeh, and W. Shi. pTop: A Process-level Power Profiling Tool. In *HotPower'09: 2nd Workshop on Power Aware Computing and Systems*, Big Sky, MT, USA, october 2009.

[3] J. Flinn and M. Satyanarayanan. PowerScope: A Tool for Profiling the Energy Usage of Mobile Applications. In *2nd workshop on Mobile Computer Systems and Applications*, 1999.

[4] C. Germain-Renaud, F. Furst, M. Jouvin, G. Kassel, J. Nauroy, and G. Philippon. The green computing observatory: A data curation approach for green it. In *9th International Conference on Dependable, Autonomic and Secure Computing (DASC)*, pages 798–799, 2011.

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

[6] D. McIntire, T. Stathopoulos, and W. Kaiser. ETOP: sensor network application energy profiling on the LEAP2 platform. In *6th international conference on Information processing in sensor networks*, pages 576–577. ACM, 2007.

[7] A. Noureddine, A. Bourdon, R. Rouvoy, and L. Seinturier. Runtime monitoring of software energy hotspots. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 160–169, New York, NY, USA, 2012. ACM.

[8] A. Noureddine, A. Bourdon, R. Rouvoy, and L. Seinturier. A preliminary study of the impact of software engineering on greenit. In *1st International workshop on Green and Sustainable Software (GREENS)*, pages 21–27, June.

[9] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, Feb. 1978.

[10] C. Ruhl, P. Appleby, J. Fennema, A. Naumov, and M. Schaffer. Economic development and the demand for energy: A historical perspective on the next 20 years. *Energy Policy*, 50(0):109 – 116, 2012.

[11] C. Seo, S. Malek, and N. Medvidovic. An energy consumption framework for distributed java-based systems. In *22nd international conference on Automated software engineering*, pages 421–424. ACM, 2007.

[12] C. Seo, S. Malek, and N. Medvidovic. Estimating the energy consumption in pervasive java-based systems. In *6th International Conference on Pervasive Computing and Communications*, pages 243–247. IEEE, 2008.

[13] A. E. Trefethen and J. Thiyagalingam. Energy-aware software: Challenges, opportunities and strategies. *Journal of Computational Science*, (0), 2013.

[14] W. Vereecken, W. Van Heddeghem, D. Colle, M. Pickavet, and P. Demeester. Overall ict footprint and green communication technologies. In *4th International Symposium on Communications, Control and Signal Processing*, pages 1 –6, 2010.

[15] M. Webb. *SMART 2020: enabling the low carbon economy in the information age, a report by The Climate Group on behalf of the Global eSustainability Initiative*. GeSI, 2008.