Measuring Energy Footprint of Software Features

Syed Islam, Adel Noureddine, Rabih Bashroush

Measuring Energy Footprint of Software Features

Syed Islam, Adel Noureddine, and Rabih Bashroush
School of Architecture, Computing and Engineering
University of East London, United Kingdom
Email: syed.islam@uel.ac.uk, a.noureddine@uel.ac.uk, r.bashroush@qub.ac.uk

***Abstract*—With the proliferation of Software systems and the rise of paradigms such the Internet of Things, Cyber-Physical Systems and Smart Cities to name a few, the energy consumed by software applications is emerging as a major concern. Hence, it has become vital that software engineers have a better understanding of the energy consumed by the code they write. At software level, work so far has focused on measuring the energy consumption at function and application level. In this paper, we propose a novel approach to measure energy consumption at a feature level, cross-cutting multiple functions, classes and systems. We argue the importance of such measurement and the new insight it provides to non-traditional stakeholders such as service providers. We then demonstrate, using an experiment, how the measurement can be done with a combination of tools, namely our program slicing tool (`PORBS`) and energy measurement tool (`Jolinar`).**

I. Introduction

With the proliferation of Software systems and the rise of paradigms such the Internet of Things, Cyber-Physical Systems and Smart Cities to name a few, the rate of energy consumption of Information and Communication Technologies (ICT) is rising at an alarming rate. It is estimated that ICT consumption will rise from 168 to 433 Gigawatts (7% to 14.5%) by 2020 [1]. Greenhouse Gas emissions (GH) from ICT are also expected to double to 1430 MtCO2e within the same period [2], highlighting the need for more energy efficient hardware as well as software.

In addition to traditional software properties often cited to aid in program comprehension [3], we argue that in the near future, it will become vital for software engineers to better understand the energy consumed by their applications. While software energy consumption is already a major software engineering concern in some application areas such as mobile computing (given the impact on battery life, and ultimately the product viability), monitoring and optimising the energy consumption of software systems is gaining considerable traction else where. A recent survey showed that software architects envisage energy becoming a major architectural concern in the next five years [4].

Current research focuses on measuring the energy consumption of applications at two main granularities. The first considers the energy consumption of a particular function or a set of functions, while the second level of granularity looks at the energy consumed by the entire software stack or process. Work in this area entails creating accurate models to estimate the energy consumption of software, which sometimes includes the use of complex formulas or additional hardware. Such measurements are *vertical* in nature and oriented towards measuring the consumption of software constructs. Although such approaches can aid software engineers identify bottlenecks, hotspots or energy smells in the code, they do not lend themselves to better understand consumption at feature-level, which is ultimately what is being consumed by end-users.

In this paper, we propose a novel approach for measuring energy consumption of end-user features, taking a more *horizontal* cross-cutting view. Our approach will enable newer levels of programme comprehension and understanding. On the one hand, it will empower software engineers analyse the consumption of a given feature, which can cross-cut multiple classes, or even programming languages. On the other hand, it will help cloud service providers isolate and better understand energy consumption of their system features, and where applicable, introduce better charging models for end-users based on real energy consumption.

The rest of the paper is structured as follows. The next section discusses related literature and motivation. Section 3 presents our feature oriented approach. Validation and results are then analysed in Section 4. Finally, Section 5 rounds off the paper with a summary and future work.

II. Motivation & Related Work

Current approaches for monitoring energy range from hardware devices to power models and tools [5]. Hardware-based solutions, such as power meter devices and dedicated integrated circuits or sensors, demand additional investment, complex installations, and treat the entire system as a black box. Other approaches overcome such large granularity limitations by estimating software energy using power models or a calibration process. Tools, such as pTop, PowerSpy, PowerAPI, JouleMeter or Energy Checker provide power and energy insights into software energy footprint but at the cost of limiting usability. For instance, some require a power meter in order to calibrate their models, or for runtime energy monitoring (PowerAPI, PowerSpy, JouleMeter), others require modifications to applications' source code or patching parts of the operating system to effectively monitor software energy consumption (pTop, Energy Checker). Source-based approaches provide energy readings for methods and functions [6] or for individual lines of code [7]. These approaches measure the energy consumption of software artefacts (*e.g.*, methods, classes, lines of code), mostly ignoring their functionality or the relationships between them.

Modern software systems are typically composed of more than one programming language and often comprise dozens of features that cross-cut hundreds of components. Thus, the comprehension of such information would allow developers to focus their optimisation efforts on entire features consumed by end-users, rather than only software constructs. This, for example, could help cloud providers model their billing structure around the energy consumption of particular system features.

III. Approach

Our approach to measuring software energy consumption at the feature-level has two steps. The first step involves the identification and extraction of an executable subset of the original program that implements the feature of interest (feature slicing). The second involves the measurement of the energy consumed by the feature slice (energy measurement). The two steps are discussed below.

A. Feature slicing

The first step in our approach is to identify what parts of the system constitutes a feature of the software. Unlike formal functional components of a software system, what constitutes a feature is often down to the granularity at which the software system is modelled. At higher architectural levels, a feature constitutes work done by several modules to achieve some purpose; However, even a small utility function within that same system can be regarded as a feature when taking a more fine-grained view.

Program Slicing is a technique for identifying the influence or dependencies for a specific point of interest within the code, referred to as the slicing criteria. Program Slicing is well researched and has many applications [8], including comprehension, testing, debugging, maintenance, re-engineering, re-use, and refactoring. Although useful for debugging, the original notion of program slicing now known as static slicing is not suitable for feature extraction. The static form is a conservative approach where any possible influences on the slice criterion for all possible inputs are identified. This conservatism over approximates and includes much more dependencies than what a typical execution of a feature would require. For example, if a static slice was taken with respect to variable n on line 38 of program p (Figure 1), the static slice would include the entire program. Several extensions to the original notion of slicing have been proposed over the years that attempt to tackle feature extraction [9], [10], [11]. However, most of these techniques do not yield a slice that is an executable program in its own right. Furthermore, none of these techniques can handle multi-language systems.

Recently proposed Observation-based Slicing [12] is a technique that finds its origin in Weiser's original motivation for slicing [13]: uninteresting statements can be deleted (sliced out) of a program to help focus attention on relevant statements (i.e., the slice). Operationally, PORBS[1] (our observation-based slicer) achieves this by tentatively deleting one or more statements and then observing the behaviour of the resulting program. It attempts to find a subset of the program (slice) that preserves the behaviour of the original program with respect to a certain set of inputs and the slice criterion.

[1]http://www.syedislam.com/orbs.html

-s	-u	Program P
1	1	`public class SortUniqeUtility {`
2		
3	1	`  static ArrayList l = new ArrayList();`
4		
5	1	`  public static void main (String[] args)...{`
6	1	`    readFile(args[0]);`
7		`    if (args[1].equals("-s"))`
8		`      sort();`
9		`    if (args[1].equals("-u"))`
10		`      unique();`
11	1	`    writeList();`
12	1	`  }`
13		
14	1	`  static void readFile(String s) ... {`
15	1	`    FileReader fr = new FileReader(s);`
16	1	`    BufferedReader br = new BufferedReader(fr);`
17	1	`    String line = "";`
18	1	`    while ((line = br.readLine()) != null) {`
19	1	`      l.add(line);`
20	1	`    }`
21		`    br.close();`
22		`    fr.close();`
23	1	`  }`
24		
25		`  static void sort(){`
26		`    Collections.sort(l);`
27		`  }`
28		
29		`  static void unique(){`
30		`    Set<String> hs = new HashSet<>();`
31		`    hs.addAll(list);`
32		`    l.clear();`
33		`    l.addAll(hs);`
34		`  }`
35		
36	1	`  static void writeList(){`
37	1	`    for (String n : l)`
38	1	`      System.out.println(n); // slice on n`
39	1	`  }`
40	1	`}`

Fig. 1. Observation-based Slice for Line 38 with option -s and -u.

Figure 1 shows a program P, which is a utility program with two features, sort and unique. The first, reads a file and outputs contents of the file in sorted order, while the second, reads a file and outputs non-repeating contents. The features are triggered by providing the name of the input file and options -s (sort) or -u (uniq), respectively. We present this simplistic version of the utility tool to explain our approach ignoring details such as error checking. The code has been structured in such a way that the `main`, `readFile` and `writeList` methods will be required for both utilities. Each specific feature is then provided by one of the two methods, namely sort by `sort` method and uniq by `unique` method.

Running PORBS slicer on the code with an appropriate input file and a feature options causes PORBS to produce an executable slice implementing the certain feature. For example, Column 2 of Figure 1 marks lines included in the slice for the feature option -s and Column 3 marks the lines included in the slice for feature option -u. Both slices are

executable programs on their own right and will produce the same output as the original program (P) for respective features. PORBS is therefore able to extract executable slices that contains all the code pertaining to the implementation of a particular feature within a program. We are therefore using PORBS in our approach to extract a minimal set of code that implements a certain feature.

B. Energy measurement

We estimate the energy consumption of features using Jolinar [2], an accurate, lightweight and easy-to-use tool for energy measurements. Jolinar is based on our energy models introduced and validated in [14], [15]. These models use publicly available hardware OEM specifications to calculate resource utilisation for monitored applications, then estimate their energy consumption based on energy formulas. First, we measure resource utilisation by the hardware (so far, CPU, disk and memory) and apply energy models to estimate how much the hardware modules are currently consuming. Next, we capture the resources used by the monitored application (for example, number of CPU cycles, or number of bytes written to the disk the application is using), and apply a second set of energy models to estimate software energy consumption.

The models in Formulas 1 and 2 takes into consideration modern processors' characteristics, such as DVFS (Dynamic Voltage and Frequency Scaling), CPU's TDP (Thermal Design Power) and real-time processor frequency and voltage changes. The power consumption is calculated every 500 milliseconds, following power variation whenever frequency or voltage changes.

$$P^f_{CPU} = \frac{0.7 \times TDP}{f_{TDP} \times V^2_{TDP}} \times f \times V^2 \times \frac{t^{PID}_{CPU}}{t_{CPU}}(d) \quad (1)$$

$$P_{CPU} = \frac{\sum_{f \in frequencies} P^f_{CPU} \times t^f_{CPU}}{\sum_{f \in frequencies} t^f_{CPU}} \quad (2)$$

Disk and memory energy models use a similar approach of capturing resources and estimating energy consumption. Disk energy model uses the number of bytes read and written by the application (see Formula 3), while the memory mode uses the percentage of RAM memory occupied by the application. These models were validated with an error margin of around 3% on average [14], [15].

$$P_{disk} = Bytes_{read} \times \left(\frac{DiskPower_{read}}{DiskRate_{read}}\right) + Bytes_{write} \times \left(\frac{DiskPower_{write}}{DiskRate_{write}}\right) \quad (3)$$

IV. Validation

We validate our approach using the use case shown in Figure 1 where we identify two executable program slices, the first one is for the sort feature and the second one for the uniq feature. Following the extraction of these slices using PORBS, we apply our energy model using Jolinar to estimate the amount of energy consumed by each of the slices when processing a file with 100,000, 1,000,000 and 10,000,000 integer values using a machine running an Intel Pentium T3400 processor running at 2.16 GHz and a Seagate Momentus 5400.5 SATA hard disk.

[2]http://www.noureddine.org/research/jolinar

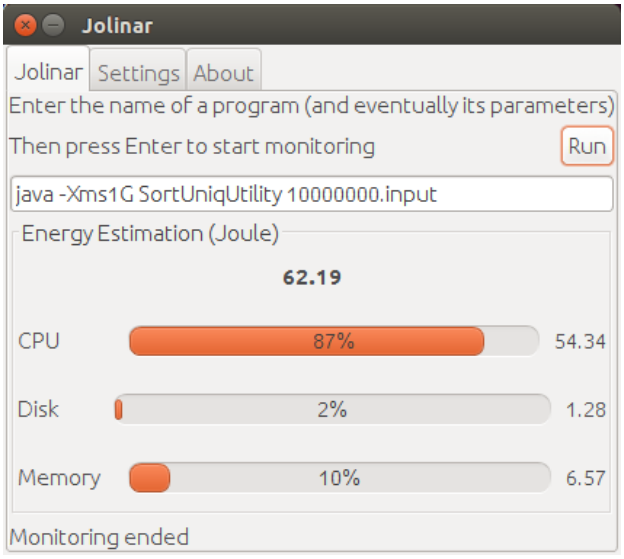

Fig. 2. Jolinar's output for uniq slice with 10'000'000 inputs

Figure 2 shows the energy output running for uniq slice with 10,000,000 inputs as provided by Jolinar. Table I outlines the energy results for the different inputs, detailing energy consumed by the CPU, disk and memory, for both the sort and uniq slices. The results show an exponential growth in the energy consumption for the sort feature, going from 10 joules to more than 1415 joules when integer inputs are multiplied by 100. The majority of this energy is due to sorting calculations (therefore the CPU), with an increase of memory energy for larger data sets. Nearly a third of the energy (31.2%) consumed by the 10 million integer sort feature is due to memory access (read/write) for sorting. In particular, `Collections.sort()`, which is used in the sort feature (see line 26 in Figure 1), requires a temporary storage of up to n/2 object references [16]. On the other hand, the uniq feature exhibits a much lower memory energy consumption as the `unique` method (see line 29, Figure 1) only adds input to a hashset and a list.

Figure 3 and 4 show the power consumption of the sort and uniq features with 10,000,000 inputs, respectively. Initially, there is an increase in disk power consumption due to disk activities reading the input file (*e.g.*, method `readFile(String s)` in lines 14–23 in Figure 1). Memory consumption for sort is constant across the experiment as Java allocates a large memory buffer for sorting and did not require additional memory space. More interestingly, the graph outlines spikes in the CPU power consumption which are explained by the sorting cycles in the algorithm and the copying of data in-between. With a shorter experiment duration, uniq seeks processing power for copying and clearing Java

Input File Size	Input Size	sort CPU	sort Disk	sort Memory	sort Total	uniq CPU	uniq Disk	uniq Memory	uniq Total
556K	100000	9.51	0.01	0.23	9.75	5.49	0.01	0.07	5.57
5.4M	1000000	55.6	0.12	7.46	63.18	8.68	0.12	0.22	9.02
54M	10000000	972.3	1.18	441.44	1414.92	54.34	1.18	6.53	62.05

TABLE I
ENERGY CONSUMPTION BY THE FEATURES SORT AND UNIQ FOR VARIOUS INPUT SIZES

collections objects (a hashset and a list) as seen in Figure 4.

Comprehension of energy consumption by feature can help developers identify the features that are consuming more energy and use additional information (such as importance and likely frequency of use) to decide on where to focus optimisation efforts (*e.g.* optimise code or dedicate energy efficient resources to frequently used features).

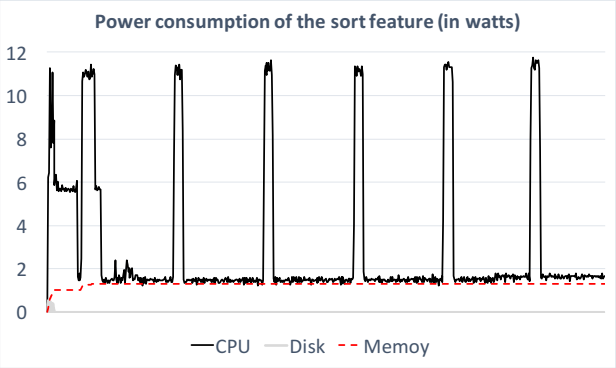

Fig. 3. Power consumption of the sort slice with 10,000,000 inputs

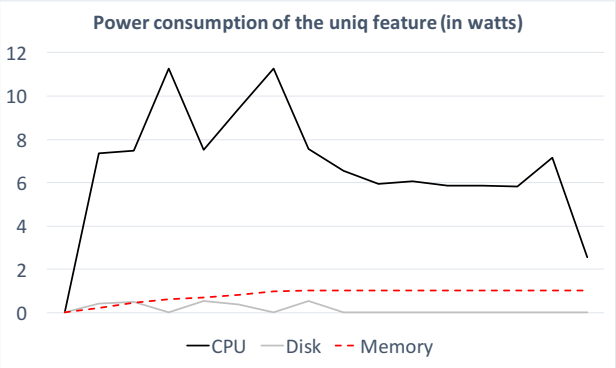

Fig. 4. Power consumption of the uniq slice with 10,000,000 inputs

V. SUMMARY AND FUTURE WORK

In this paper, we present a novel approach for better understanding the amount of energy consumed by software at a feature level as compared to traditional methods which analyse energy consumption based on software artefacts (*e.g.* function or process). We argue the value of such analysis and validate our approach by using `PORBS` to extract executable slices from an example program for each specific feature. We then use `Jolinar` to measure the amount of energy consumed by these features. Our validation shows that we are able to identify and measure the energy consumption of features (slices), which could help software engineers conduct a new level of analysis and optimisation (e.g. focus on what matters most).

Future work includes extending our empirical validation using large real-world cloud-based systems where such information is likely to have a huge impact. We also plan to automate the energy monitoring at feature-level, and expand our energy models to additional hardware and platforms in the cloud. Finally, our future work will also consider calculating in real-time the GHG emissions of features based on energy consumption and power grid mix (*e.g.* % renewables) data in countries where such information is readily available.

REFERENCES

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

[2] Molly 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)*. GeSI, 2008.

[3] M. A. Storey. Theories, methods and tools in program comprehension: past, present and future. In *13th International Workshop on Program Comprehension*, May 2005.

[4] R. Bashroush, E. Woods, and A. Noureddine. Data center energy demand: What got us here won't get us there. *IEEE Software*, 33(2):18–21, Mar 2016.

[5] Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. A review of energy measurement approaches. *SIGOPS Oper. Syst. Rev.*, 47(3):42–49, November 2013.

[6] Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. Monitoring energy hotspots in software. *Automated Software Engineering*, 22(3):291–332, 2015.

[7] Ding Li, Shuai Hao, William G. J. Halfond, and Ramesh Govindan. Calculating source line level energy information for Android applications. *Proceedings of the 2013 International Symposium on Software Testing and Analysis - ISSTA 2013*, page 78, 2013.

[8] Andrea De Lucia. Program slicing: Methods and applications. In 1^{st} *IEEE International Workshop on Source Code Analysis and Manipulation*, pages 142–149, Los Alamitos, California, USA, 2001.

[9] Andrea De Lucia, Anna Rita Fasolino, and Malcolm Munro. Understanding function behaviours through program slicing. In 4^{th} *IEEE Workshop on Program Comprehension*, pages 9–18, March 1996.

[10] W. E. Wong, S. S. Gokhale, J. R. Horgan, and K. S. Trivedi. Locating program features using execution slices. In *Application-Specific Systems and Software Engineering and Technology*, pages 194–203, 1999.

[11] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. 29(3), 2003. Special issue on ICSM 2001.

[12] David Binkley, Nicolas Gold, M. Harman, Syed Islam, Jens Krinke, and Shin Yoo. ORBS: Language-independent program slicing. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, FSE 2014, pages 109–120, 2014.

[13] Mark Weiser. Program slicing. In 5^{th} *International Conference on Software Engineering*, pages 439–449, San Diego, CA, March 1981.

[14] Adel Noureddine, Aurelien Bourdon, Romain Rouvoy, and Lionel Seinturier. A preliminary study of the impact of software engineering on greenit. In *Proceedings of the First International Workshop on Green and Sustainable Software*, Piscataway, NJ, USA, 2012. IEEE Press.

[15] Adel Noureddine. *Towards a Better Understanding of the Energy Consumption of Software Systems*. Theses, Université des Sciences et Technologie de Lille - Lille I, March 2014.

[16] Java class collections. https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html.