Understanding Green Software Development: A Conceptual Framework

Luca Ardito [1], Giuseppe Procaccianti [1], Marco Torchiano [1], Antonio Vetrò [1,2]
[1]Politecnico di Torino, [2]Technische Universität München

Abstract

The energy efficiency of IT has become one of the hottest topics in the last few years. The problem has been typically addressed by hardware manufacturers and designers, but recently the attention of industry and academia shifted to the role of software for IT sustainability. Writing energy efficient software is one of the most challenging issues in this area, because it requires not only a change of mindset for software developers and designers, but also models and tools to measure and reduce the effect of software on the energy consumption of the underlying hardware. In this paper we present a conceptual framework that provides a unifying view on the strategies, models and tools available so far for designing and developing greener software.

I. Introduction

IT energy consumption represents an increasingly relevant concern. Traditionally, it has only been addressed by hardware designers. However, as hardware got more and more powerful, the influence of software behavior on energy consumption grew significantly.

During the last few years, the authors have explored several facets of IT energy consumption from a software engineering perspective. What was once purely anecdotal evidence about the pivotal role of software on energy consumption is now supported by sound empirical data collected though a series of experiments on different hardware platforms: servers, desktop PCs, and mobile phones. Concerning servers, we analyzed the power consumption of three servers performing different tasks, and we observed that power consumption can increase up to 40% depending on the entity of the task [1]. We analyzed desktop computers from different technological generations, in distinct software usage scenarios, and found out that, depending on the software applications used, power consumption can increase up to 20% [2]. Finally, we profiled the power consumption of mobile devices, making a comparison between two generations of Android OS-based smartphones. Our results show how different execution profiles of the same application can significantly affect the power consumption of a mobile device [3].

Although the actual figures may vary depending on the specific hardware platform, the impact of software over energy consumption is definitely relevant. Realizing this implies a change of mindset for the software engineering community. Firstly, determining the responsibility of software means increasing the awareness of developers over the energy consumption caused by their applications. Secondly, a new perspective of software as hardware driver arises. In this paper we aim at providing a conceptual framework to support the new mindset by focusing on existing techniques and providing useful unit of measurement, in order to estimate the energy consumption of software.

II. A Framework for Energy-Efficient Software Strategies

There is a joke:

How many hardware engineers does it take to change a light bulb?
None. "We'll fix it in software".

As people say, there is a grain of salt in any joke. While, at the beginning of computer science, hardware and software were tightly mingled together and mostly indistinguishable, now things have changed and hardware and software became more distant: software layers are constantly increasing, in order to provide encapsulation and abstraction for software applications. The goal of layers is to abstract – in the sense of selective removal – several details and shield upper layers as much as possible from complexity that lies in the lower layers. The first details that have been removed are those concerning how hardware works. The result is that layers stand in the way of linking software to its physical consequences, such as its energy consumption.

In practice, in any programmable device, although the ultimate responsibility of energy consumption is always with the hardware, the way the energy is consumed is dictated by the software. The *abstract* model underlying the power consumption can be summarized as:

$$Power = Idle + \sum_{c \in Components} Hw_c \cdot Sw_c \quad \text{where} \quad 0 \leq Sw_c \leq 1 \tag{1}$$

The total power consumption $Power$ of an IT device – when turned on – is composed by an $Idle$ part that is present even when the device is sitting idle. The additional consumption depends on the individual hardware components maximum

consumption Hw_c which is modulated by how much work the software demands them to do, Sw_c. Depending on the software requests the hardware component may run at full throttle or remain idle.

This theoretical software power model supports higher – software level – strategies for increasing software energy efficiency. As a matter of fact, it gives developers a way to elaborate a strategy by analysing the causes of energy consumption and also to validate the efficacy of the formulated strategies by measuring their impact and effect. According to such perspective on power consumption, the two main strategies to achieve efficient green software can be organized in the framework presented in Figure 1.

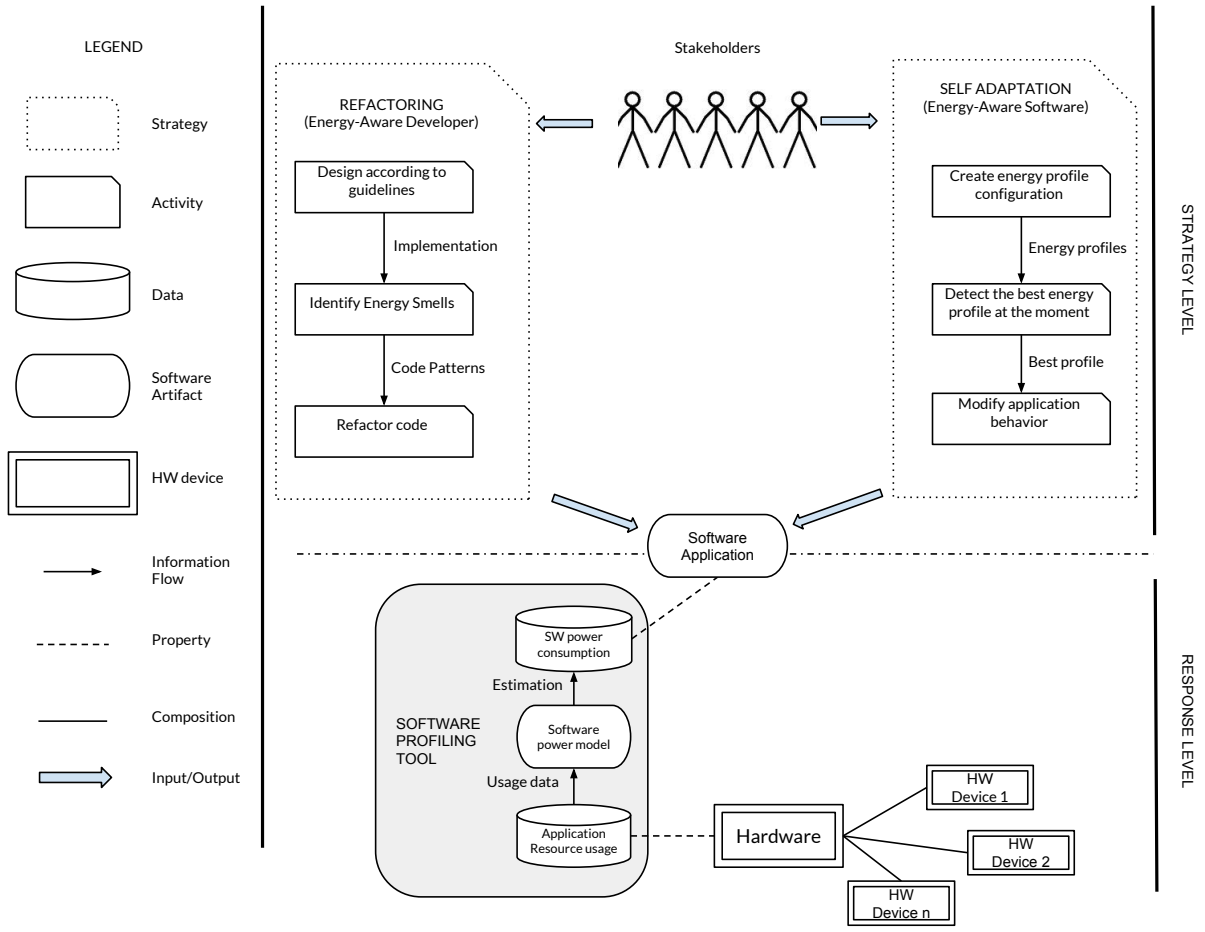

Fig. 1: Framework for Energy-Efficient Software Strategies

At the top of the figure are those stakeholders who are interested to or affected by sustainability issues (a list of them is provided by Penzenstadler et al.[4]): they trigger the need for developing energy-efficient software. This serves as an input to the Strategy level, where operational decisions for *greening* the software are taken. The proposed framework includes two main strategies: refactoring and self-adaptation.

The refactoring strategy is shown on the left side and described in detail in Section IV-A. It is focused on minimizing software instructions and code patterns that may cause higher energy usage. The self-adaptation strategy, shown on the right side of the figure, has the main goal of creating an energy-aware application that is able to choose among different configurations, with respect to different scenarios and contexts, that we call "energy profiles" (see Section IV-B). The two strategies are not meant to be mutually exclusive: they can be applied together in the same development process. In addition, other technological, human or process strategies can be plugged in whenever their impact is measurable and linkable to a software application and its power consumption, through modelling and profiling. Both strategies should be applied iteratively, by verifying the energy efficiency improvements, through power profiling tools, and consequently adapted. They should also be applied carefully, keeping in mind the software mission and its main functionalities, the required quality of service, and the interests of the stakeholders. For example, applying self-adaptation to reduce the network usage might improve energy efficiency, but it might also violate service level agreements on response time or availability.

The bottom part of the figure shows the "Response" level, meant to identify opportunities for energy optimization and/or to assess the energy savings gained by applying our strategies. Resource usage information, such as memory accesses, device usage, CPU mode and such, is collected from the hardware. This information is used as input for software profiling tools,

that analyze software applications during execution and provide on-line power consumption estimation values with different granularity. Software power models represent a crucial component of our framework, because they enable the formulation and validation of the strategies. The software power consumption models are described in more detail in the following section, along with their implementation inside software energy profiling tools such as *Joulemeter*[5], *ARO*[6], *Power TOP*[7], and *PowerTutor*[8].

III. Software power consumption models

To date, many research efforts have been devoted to predict how much energy a computer system will consume when running a specific application or performing a specific task. The modeling approaches can be "*white-box*" i.e. code-level or instruction-level metrics, and "*black-box*" i.e. runtime metrics, such as usage ratios of system resources (CPU, RAM, etc.). Of course, timing is an important factor: white-box models can be used to provide both an on-line (run-time) and off-line (compile-time) estimation of the software impact, while black-box can only perform on-line because they rely on resource usage data. However, real-time estimation is hard to achieve with both of the approaches. Due to the different time scale between hardware and software events, a certain amount of latency between the actual value and the prediction has to be taken into account.

From our experience, the effectiveness of the chosen predictors varies greatly with respect to the hardware system considered. In embedded systems, for example, we have observed that code-level constructs may have an observable impact over power consumption only in some cases [9]. However, as the system architecture becomes more complex, these models appear to be too fine-grained to describe the effect of software over power consumption. In these cases, a resource-usage based model might be more meaningful: initial work [2] has successfully proven the correlation between indicators of hardware resources and the power consumption of a desktop computer system. As a matter of fact, most of the software power profiling tools commercially available are based upon these types of models.

Choosing the appropriate resources (or, more precisely, resource usage metrics) as predictors is the key to build an accurate resource-based power consumption model. Typically, CPU is the most important component to monitor: this is why, especially on more advanced mobile systems such as smartphones, most tools focus on the CPU usage as a predictor for software power consumption. However, our experiments have proven that other metrics, such as memory usage and, more importantly, I/O operations, have a significant correlation with power consumption.

Moreover, in some usage scenarios, software applications may require the activation of high power-consuming peripherals (e.g. GPS modules, 3G and WiFi antennas) that significantly modify the consumption profile of the device. In Figure 2 we can see an example showing typical power consumption of a mobile device in different usage scenarios [10] .

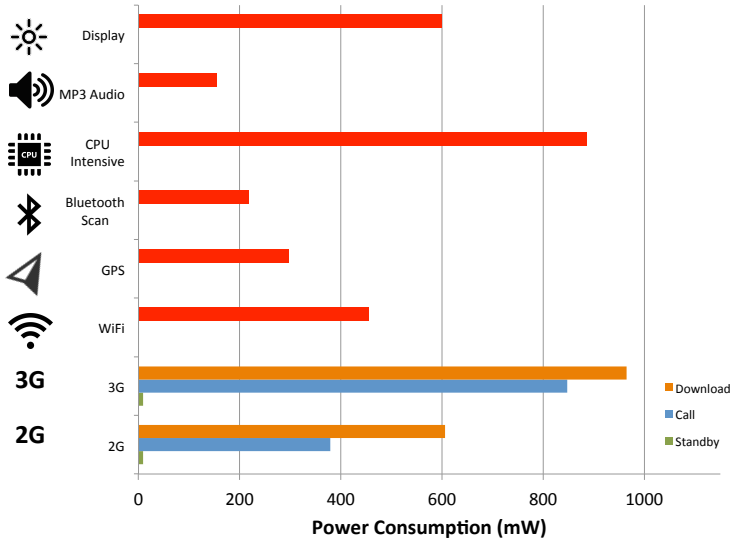

Fig. 2: Power Consumption of a Mobile Device in Different Usage Scenarios (data from [10])

This suggests two considerations: 1) these resources cannot be ignored by models and must be explicitly measured, 2) software developers need to be aware that decreasing the computational complexity of software applications is not enough to develop an energy-efficient application, but they need to adopt an holistic approach.

IV. Developing Green Software

A. Refactoring: adopt code level guidelines to improve energy efficiency

Predictive models embed the knowledge about both the resources (e.g., CPU) that consume power and the activities (e.g., disk transfers) that drive their consumption. As a consequence, the next step, from a developer perspective, is to identify those code patterns that imply high usage of those resources and activities. Taking inspiration from the well-known book of Fowler and Beck [11] we call them *Energy Code Smells* [9], i.e. implementation choices (at code, design or architectural level) that make the software execution less energy efficient.

Combining our experience together with the evidence provided by similar works ([12], [13], [14] and [15]), we derive a few lessons learned, which represent a set of code-level guidelines for developers aiming at greening their applications.

Clean up useless code and data.
As software evolves, many parts may become obsolete. Writing to never-read variables and other useless routines (e.g., repeated conditionals) may consume power purposelessly. Cleaning up these instructions might improve energy efficiency, as well as maintainability. Many static analysis tools are able to detect useless code.

Look for Immortals.
The lifecycle of software processes and threads must be carefully managed. The Immortality Energy Smells describes situations where a software service restarts after explicitly being killed by the user, continuing to drain energy. Sometimes, software immortals are created on purpose: in this cases, death and rebirth phases of the processes/threads should be as graceful as possible, in order to reduce the resource usage overhead and the consequent energy waste.

Monitor the appropriate resources. The physical responsible of energy usage is hardware. That being said, in modern computer systems there are many different hardware devices, with different power requirements. Understand first which hardware needs more energy (e.g., GPS transmitter, wireless antennas, sensors), then which software routines make use of it.

Scenario driven refactoring.
The software execution depends not only on its internal structure and host environment (e.g., operating system), but also on the input it receives. Thus, an energy refactoring operation may show its results only in specific situations. Focusing on common usage scenarios makes the improvements more perceivable by the end users and saves eventually more energy.

Focus on higher-level structures and complex routines
Like in performance optimization problems, improvements obtained at lower level might be hidden from higher level inefficiencies. This is especially true when there are many software layers or when software runs in a complex environment (e.g., virtualization, distributed systems). Start refactoring from higher level constructs: their impact on CPU and memory (and consequently, energy) is significantly higher compared to basic data types.

Do not trust loops. Loop constructs are powerful, but their contents must be carefully monitored. Loop smells happen when an application repeats the same activity on a loop, without achieving the intended results and uselessly consuming energy (e.g., polling an unreachable server). Detecting and refactoring such loops can save lot of energy, especially on battery powered devices.

Reduce amount of data transferred.
In distributed and high-performance systems, or in battery powered devices using power-consuming radio transmission, data transfer might be a significant source of power drain. Data exchanged between software applications and/or databases (local or remote) can be optimized using data compression or data aggregation techniques. The energy impact of this optimization might be crucial, in data-intensive and Big Data applications.

The refactoring strategy mostly operates at code level, hence code-related metrics can be used to guide the application of the strategy. An example metric is Communication Energy Cost [16], that estimates the energy consumption induced by data transfers for each software component.

B. Self-adaptation: Green software by design

While guidelines are useful for developers who aim at *greening* their existing applications, self-adaptation techniques are a good solution when you can draw your software's architecture from scratch. The key idea is to provide different configurations

	Profile 1			Profile 2	
Software Sensor	State (ON/OFF)	Refresh After (s)	Software Sensor	State (ON/OFF)	Refresh After (s)
PhoneSensor	ON	3600	PhoneSensor	ON	600
LocationSensor	OFF	-	LocationSensor	ON + GPS ON	600
WiFiSensor	OFF	-	WiFiSensor	ON	600
BluetoothSensor	OFF	-	BluetoothSensor	ON	600
DeviceInfoSensor	ON	3600	DeviceInfoSensor	ON	3600
DeviceStatusSensor	ON	3600	DeviceStatusSensor	ON	600
DeviceSettings	ON	3600	DeviceSettings	ON	600
TerminalActivity	OFF	-	TerminalActivity	ON	600
DataSensor	ON	3600	DataSensor	ON	3600

TABLE I: Example of Configuration file for Self-Adapting Applications

of the same application that can be activated at different times in order to find the best trade-off between features provided and energy consumed. The main issue that arises in this approach is due to the need of getting instant power consumption data from devices; the hard part is getting the data with a limited overhead and without resorting to external equipment.

We can consider a device as composed of three layers: Hardware, Operating System, and Application.

Considering a single device, the data consumption flow should reach the upper layers (operating system and application) from the hardware layer in order to be read without introducing too much overhead. In this way the upper layers can be aware of the resource usage, and of their instant power consumption. Currently this data is typically computed by isolating the use of a given resource by a process, and multiplying it by the theoretical consumption value of that resource in that situation (see Section III). This method introduces an error due to the theoretical value and an overhead due to the computation. The use of built-in power meters installed in the hardware layer would help to send energy consumption data in real time to the upper levels with the advantage of getting the consumption data by simply reading a value.

Based on this architecture, the operating system:

- Receives consumption data in real time,
- Processes all the data received,
- Interacts with the application layer.

The application layer needs a suitable interface to exchange energy information with the operating system layer. Actions needed to implement self-adaptation depend on the context in which the application will be used. It is important to create usage scenarios in order to measure the ”energy cost” of a single functionality provided by the application. Based on the functionality energy cost, it is then possible to create different Energy Profiles, which will contain the full set or a subset of functionalities. This approach follows the ”scenario driven refactoring” guideline and it uses the energy consumption data provided by the OS layer. After that it is possible to work on the single functionality by deciding whether to include it or not in an ”Energy Profile”, or to change some parameters in order to save energy. In [3] we present a mobile application, which reconfigures itself based on the remaining battery level. We implemented self-adaptation working on the application functionalities, enabling or disabling modules and tuning some parameters, which changed the data granularity collected from the device and then sent to the server. Table I shows an example of different configurations of the same mobile application.

Results show improvements up to 30% compared to an equivalent, non reconfiguring application. Improvements depend on the scenario and on the level of trade-offs that developers are willing to reach.

Compared to the refactoring strategy, self-adaptation introduces a relevant set of changes to the software system. While refactoring operates at code level, self-adaptation is more of an architectural concern. Raibulet et al. [17] proposed a set of architectural metrics to evaluate the adaptivity of a software system. Although those metrics are not specific for energy-driven self adaptation, they can be adopted as a reference for developers that want to introduce self-adaptive mechanisms in their application. For example, the *MaAC (Minimum architectural Adaptive Cost)* expresses the fixed cost of adaptivity at architecture level.

V. Conclusions

There is significant evidence that a growing share of green IT will be addressed by green software engineering.

From a management perspective, making software greener is a challenging task that involves complex tradeoffs among stakeholders.

From a technical perspective, several tools and good practices are available, although they are not yet well integrated in an organic framework able to provide the software developers and designers a unifying view.

We presented a conceptual framework that provides an high-level view over the possible operational strategies for developing greener software, by leveraging the information provided by power consumption models and power profiling tools. We identified and explained two such strategies, i.e. refactoring and self adaptation, although other strategies, at technological or process level, can be plugged in the framework, provided that they have a measurable impact on power consumption.

This fundamental condition is the natural consequence of one of the lesson learned in 45 years of software engineering: no improvement is possible without measurement.

REFERENCES

[1] A. Vetro', L. Ardito, M. Morisio, and G. Procaccianti, "Monitoring IT Power Consumption in a Research Center: Seven Facts," in *ENERGY 2011, The First International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies*, 2011. [Online]. Available: http://www.thinkmind.org/index.php?view=article&articleid=energy_2011_4_20_50078

[2] G. Procaccianti, A. Vetro', L. Ardito, and M. Morisio, "Profiling Power Consumption on Desktop Computer Systems," in *Information and Communication on Technology for the Fight against Global Warming*, ser. Lecture Notes in Computer Science, D. Kranzlmüller and A. Toja, Eds. Springer Berlin / Heidelberg, 2011, vol. 6868, pp. 110–123. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-23447-7_11

[3] L. Ardito, M. Torchiano, M. Marengo, and P. Falcarin, "glcb: an energy aware context broker," *Sustainable Computing: Informatics and Systems*, vol. 3, no. 1, pp. 18 – 26, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2210537912000522

[4] B. Penzenstadler, H. Femmer, and D. Richardson, "Who is the Advocate? Stakeholders for Sustainability," in *2nd International Workshop on Green and Sustainable Software (GREENS) at ICSE*, 2013.

[5] Microsoft Research, "Joulemeter," last visited: May 5th, 2014. [Online]. Available: http://research.microsoft.com/en-us/projects/joulemeter/

[6] AT&T, "Application resource optimizer (aro)," last visited: May 5th, 2014. [Online]. Available: https://developer.att.com/application-resource-optimizer

[7] Intel Open Source Technology Center, "Powertop," last visited: May 5th, 2014. [Online]. Available: https://01.org/powertop

[8] M. Gordon, L. Zhang, and B. Tiwana, "Powertutor," last visited: May 5th, 2014. [Online]. Available: http://ziyang.eecs.umich.edu/projects/powertutor/

[9] A. Vetro', L. Ardito, G. Procaccianti, and M. Morisio, "Definition, implementation and validation of energy code smells: an exploratory study on an embedded system," in *ENERGY 2013, The Third International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies*, 2013, pp. 34–39.

[10] L. Ardito, G. Procaccianti, M. Torchiano, and G. Migliore, "Profiling power consumption on mobile devices," in *ENERGY 2013, The Third International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies*, 2013, pp. 101–106.

[11] M. Fowler and K. Beck, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[12] M. Gottschalk, M. Josefiok, J. Jelschen, and A. Winter, "Removing energy code smells with reengineering services," in *GI-Jahrestagung*, 2012, pp. 441–455.

[13] A. Pathak, Y. C. Hu, and M. Zhang, "Bootstrapping energy debugging on smartphones: a first look at energy bugs in mobile devices," in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, ser. HotNets-X. New York, NY, USA: ACM, 2011, pp. 5:1–5:6. [Online]. Available: http://doi.acm.org/10.1145/2070562.2070567

[14] B. Steigerwald, R. Chabukswar, K. Krishnan, and J. Vega, "Creating energy efficient software," 2007.

[15] L. Curtis, "Environmentally sustainable infrastructure design," *The Architecture Journal*, vol. 18, pp. 2–8, 2008.

[16] C. Seo, S. Malek, and N. Medvidovic, "Component-level energy consumption estimation for distributed java-based software systems," in *Component-Based Software Engineering*. Springer, 2008, pp. 97–113.

[17] C. Raibulet and L. Masciadri, "Evaluation of dynamic adaptivity through metrics: an achievable target?" in *Software Architecture, 2009 & European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on*. IEEE, 2009, pp. 341–344.