Refactoring for Energy Efficiency: A Roadmap

(Keynote)

Gustavo Pinto, Francisco Soares-Neto, Fernando Castor
Federal University of Pernambuco
Informatics Center
{ghlp, fmssn, castor}@cin.ufpe.br

***Abstract*—Recent refactoring research introduced several innovations addressing diverse goals, such code extensibility, reusability, and testability. However, *energy consumption*, a critical property of any software system, remains unaddressed by refactoring research. In this paper, we provide an accounting of some of the recent and successful state-of-the-art research on software energy consumption. Through an investigation on premiere software engineering venues, we identify and discuss 14 contributions that can be further instantiated in refactoring tools used to improve software energy efficiency — and the challenges behind this process. The study serves as a call to action for refactoring researchers interested in software energy consumption issues.**

I. Introduction

Programs change. They might change because of changing in a requirement or because the context in which they exist changes. In order to update their programs, developers usually do so incrementally, by changing one single piece at a time. Each step can be seen as a behavior-preserving transformation, *i.e.*, a refactoring. Refactorings [10] are program transformations that change the structure of a program but not its behavior. Refactoring has several benefits, such as improving developer productivity by making it easier to maintain and understand a software. Agile advocates go further and claim that a lack of refactoring incurs technical debt [4].

Refactoring benefits are likely to go beyond understandability, covering different requirements such as extensibility, reusability, and testability. Also, recent research has succeed in applying refactoring in areas such as performance [34] and correctness [9]. Nonetheless, one grand challenge that has so far received much less attention is *energy consumption*. Despite their benefits, to the best of our knowledge, there is a lack of refactoring approaches focusing on improving the energy efficiency of a software system. This is unfortunate for at least two reasons: (1) energy is increasingly a first-order concern in any computer systems; (2) with the widespread use of mobile platforms, there is considerable evidence that battery usage is a key factor for evaluating and adopting mobile applications [50]. A robust refactoring tool used to improve software energy efficiency could thus be highly beneficial for programmers with immediate practical impact.

A first decision that needs to be made when deriving a refactoring is to determine the appropriate level of abstraction to which apply the refactoring. In this study we focus on the application level. This decision is based on the fact that while the strategy of leaving the energy consumption optimization problem to the lower-level layers has been successful, recent work showed that even better results can be achieved by empowering and encouraging software developers to participate in the process [1], [20], [26]. Thus, we believe that educating and empowering software developers with usable and useful tools can play a prominent role in reducing the energy consumption of the applications they write.

In order to derive new refactorings, it is indeed necessary to gain a deep understanding of the domain in which the refactoring will work. Notwithstanding, developing an energy efficient software is not an easy task. One of the fundamental problems in this task is to understand where energy is being consumed and how the code can be re-organized in order to reduce the energy consumed. Energy consumption estimation tools do exist (*e.g.,* [12], [24], [27]), but they do not solve this problem because (1) they require an in-depth knowledge of low-level implementation details and programmers under time pressure have little chance to learn how to use them; and (2) they do not provide direct guidance on energy optimization, *i.e.*, bridging the gap between understanding where energy is consumed and understanding how the code can be modified in order to reduce energy consumption. With no other option, programmers need to search for energy saving best practices on software development forums and blogs. Unfortunately, many of these guidelines are not supported by empirical evidences, anecdotal or even incorrect [38]. This brings us to our main research question.

RQ. What are the opportunities, and their inherent challenges, to derive new refactorings focusing on improving the energy efficiency of a software system?

This paper is aimed to provide answers to this timely but overlooked question. We mitigate this problem by investigating the state of the art of software energy consumption research and pointing out possible refactoring opportunities — and the challenges behind them. For this investigation, we have reviewed related literature of software energy consumption research published at top software engineering conferences such as ICSE and ESEC/FSE. In this mining process, we initially found a total of 20 research papers. However, after applying some filters, *e.g.,* if the paper presents an empirical study, we selected 16 of them (see Section II for details).

The main findings of this study are the following:

- We observe that software energy consumption is an emerging topic; the first research paper found is from 2012, and the number of accepted papers is increasing over the years.

- We identify 14 opportunities to refactoring for energy efficiency. These opportunities span a wide range of software characteristics, such as mobile applications, concurrent programming and software testing. Mobile applications is the topic with the greatest number of opportunities (6 out of 14).
- We identify at least one challenge for each one of the opportunities. We also found related research that succeed in overcoming similar challenges.

II. Methodology

In this section we describe our empirical study. Since we are interested in refactoring opportunities that can be used on the application, we decided to use research papers published on top software engineering conferences as our dataset. We restrict our attention to eight of the most prominent software engineering publication venues:

- **ASE**: International Conference on Automated Software Engineering
- **ESEC/FSE**: International Symposium on the Foundations of Software Engineering
- **ICSE**: International Conference on Software Engineering
- **ICSME**: International Conference on Software Maintenance and Evolution
- **OOPSLA**: Object-Oriented Programming Systems, Languages, and Applications
- **ECOOP**: European Conference on Object-Oriented Programming
- **ISSTA**: International Symposium on Software Testing and Analysis
- **CSMR**: European Conference on Software Maintenance and Reengineering

We chose ESEC/FSE, ICSE, OOPSLA and ECOOP due their wide scope. On the other hand, we chose CSMR, ICSME, ASE and ISSTA due their narrow scope. Also, all such conferences are highly competitive [48]. Thus, studies published in such conferences tend to be more mature.

The data we have analysed was restricted to the main research track of each conference. For all considered conferences, we manually searched in the proceedings of the conference for "energy" and "power" keywords in the title and in the abstracts of the accepted papers. Since energy consumption of high-level applications is a new and emerging topic of research, our extracted data covers a period of 10 years (*i.e*, 2005 — 2014). After this extraction phase, a total of 20 papers were found. Table I describes the total of papers found per conference.

From this initial set, no one is older than 2012; this shows the emerging direction of the field. From them, 3 papers were published in 2012 (ASE: [33], CSMR [17], OOPSLA: [8]), 6 papers in 2013 (CSMR: [43], ICSE: [12], [3], OOPSLA: [19], ICSME: [20], ISSTA: [24]), and 11 papers in 2014 (ICSE: [29], [32], [26], ESEC/FSE: [2], OOPSLA: [30], [18], [39], [31], ICSME: [23], [41], ISSTA: [25]).

TABLE I
Papers per conference.

Conference	Selected	Papers
ICSE	5	[12], [3], [29], [32], [26]
ASE	1	[33]
ESEC/FSE	1	[2]
OOPSLA	6	[19], [8], [30], [18], [39], [31]
ECOOP	0	—
ICSME	3	[23], [20], [41]
ISSTA	2	[24], [25]
CSMR	2	[43], [17]

For each one of the selected papers, we thoroughly read and categorized them in terms of their main contributions for energy efficiency. We discard papers that focus only on energy consumption tooling and do not provide a comprehensive case study of application energy characteristics. Without such useful information, it is not possible to understand where energy is expended in an application, and thus derive new refactorings. Thus, we discarded [24]. Also, we discard runtime systems of which performs energy optimizations behind the scenes, without changing the source code. However, we kept studies that used runtime systems, and modified the application source code. Studies that do not modify the source code do not give us any insight for source to source transformations. Thus, we removed [29], [19]. Still, we do not consider papers that do not provide empirical evidences that the proposed approach is effectively energy saving. Thus, we discarded [17]. After this filtering process, 16 papers were selected.

III. Results

In this section we present the results for our empirical study. For each category found, we provide a brief description of the problem, the refactoring opportunity, and the challenges behind it.

A. Mobile applications

Mobile devices, especially smartphones and tablets, derive the energy required for their operation from batteries, which are limited in size and therefore capacity. This implies that managing energy consumption well is of great importance. This fact encourages mobile programmers to be employ energy-aware best practices. Also, as Carroll and colleagues [7] have pointed out, graphics, GSM and CPU core are some of the most energy consuming components on a smartphone. We now present some refactoring opportunities focusing on these components.

Opportunity (A.1): Li *et al.* [26] have showed an average of 40% reduction on display's power consumption. This happens because, according to the authors, "in OLED displays, darker colors, such as black, require less energy to display than lighter ones, such as white". The authors then create an automated rewriting technique for changing darker colors to lighter ones in page candidates. However, an automatic approach deals with the problem of not achieving the desirable interface.

Refactoring can mitigate this problem by providing step-by-step user interface transformations. Refactoring can succeed at different levels here. For instance, web designers should meet refactoring. Since most of the coding style activity is done in cascading style sheets (CSS) files, refactoring engines should be integrated with well-known CSS editing tools. Also, not only for web applications, this technique can also be used in native mobile applications. When developing a mobile user interface, a programmer should have the option to refactor to a more energy-efficient color. Thus, the refactoring engine should map each possible color to its energy efficient counterpart.

Challenges (A.1): There are several significant challenges in order to create such refactorings. One of them is to properly identify the colors used in a web application. Most modern web applications combine dynamically generated pages and cascading style sheets. The refactoring tool should analyze different, and scattered, dependency files to figure out where colors are defined. On mobile applications, on the other hand, taking an example of Android applications, although the interfaces editing files are well-defined, such files are xml-based. Complex and dynamic interfaces should require several file hierarchies, and the interaction between them is ruled by Java code. Refactoring engines should be smart enough to follow all these dependencies.

Opportunity (A.2): Kwon *et al.* [20] have described a technique to offload CPU intensive computations from a mobile device to the cloud, then reducing battery usage. However, not all possible CPU intensive computations can be offloaded, since offloading is not free. It does pay a toll on energy consumption, mainly due GSM and Wi-Fi power consumption for transmitting data over the network. The trade-off of when using this technique relies on the execution time of the computation; if it is small, it does not pay the network costs. However, when careful applied, this technique can reduce the overall energy consumption of a mobile application in up to 50%. However, to take advantage of the benefits of the cloud, developers face a high entry barrier. They need expertise on many topics: communication protocols, data storage, databases, and cloud infrastructure. Moreover, the manual set up of the cloud environment is tedious, error-prone and omission-prone. A refactoring engine can greatly lower the entry barrier to allow beginner developers to partition their mobile applications, so that the energy intensive functionality can be executed in the cloud.

Challenges (A.2): The refactoring engine has two main challenges here. The first one is to determine whether the computation is worth refactoring, that is, if offloading will not turn out to be more expensive than performing the computation locally. Refactoring engines can take advantage of runtime analysis and energy consumption estimation tools to help programmers to decide when to refactor. Second, if a programmer agree with the refactoring, the refactoring engine needs to set up all the environment to receive the computation in the cloud. While starting a virtual machine with default settings can be seem as trivial, set up a particular configuration to work with a particular piece of refactored code would require sophisticated source code analysis [52]. However, recent efforts have showed that such challenges can be overcome [13].

Opportunity (A.3): Li *et al.* [23] presented the first large scale study on the energy efficiency of mobile applications. Among the findings, they describe two remarkable ones: (1) a few set of APIs used in applications dominate non-idle energy consumption, and (2) HTTP request is the most energy consuming operation of the network. Likewise, Noureddine *et al.* [33] also observed that the highest power consumption methods on the Jetty Web Server came from classes that manage HTTP requests. Still, in Hao [12], the authors examined the energy hotspots of mobile applications and, for most of the target systems, HTTP usage consumed the most energy. We believe that refactoring engines can play a role here. For instance, a refactoring engine should be able to identify such energy consuming APIs, and replace them for energy-friendly ones.

Challenges (A.3): Although some energy-intensive APIs have an energy-friendly counterpart, *e.g.,* the power efficient work queue[1], most of them do not have. Refactoring tools should keep track of the cutting edge research on energy efficient APIs. For those APIs which do not have an energy efficient implementation, refactoring tools should favor "light-weight implementations". For instance, webservices can be implemented using at least two commonplace protocols: SOAP and REST, which greatly differ in their internal characteristics. While REST is more flexible and light-weight, SOAP is more detailed and heavy-weight. In the absence of an energy efficient implementation, refactoring tools should support the transition to more light-weight components.

Opportunity (A.4): Piracy is a real issue which greatly impact apps revenue. The most commonly used approach for preventing piracy is code obfuscation, that is, making the code of their applications more difficult to understand. However, according to Sahin *et al.* [41], obfuscations techniques used on mobile applications are likely to impact their energy usage negatively. The authors report an average energy increase of of 2.1%. Refactoring tools take advantage of this fact and implement more energy efficient code obfuscations techniques.

Challenges (A.4): Writing a novel energy efficient obfuscation technique is not an easy task. Taking in consideration the "spaghetti logic" example, described in the aforementioned study, it inserts branching and conditional instructions in the body of the methods. While additional instructions are likely to increase the absolute size of the program, and thus energy usage, they can also introduce logic bugs into the code. Using two well-known compiler optimizations, peephole and inline optimizations, we believe that refactoring engines improve the energy efficiency of the generated code. However, this requires empirical evidences.

Opportunities (A.5): In Banerjee *et al.* [2], similarly to Li *et al.* [23], the authors argued that I/O components utilities contribute significantly to the energy consumption of a mo-

[1]http://lwn.net/Articles/548281/

bile application. Among the findings, the authors observed a particular application of which the GPS module continue to run for a few seconds even after the application exits. Behind the scenes, a third-party advertisement module was responsible for keeping the GPU alive. Such advertisement module as running on the main thread, and any delay in loading the advertisement from the network prolongs the entire main thread. Refactoring tools can be useful here by putting features that are additional to the requirements of users (*e.g.* advertisements), in separate daemon asynchronous threads. Then, when the main application exits, all the other related threads would also exist, avoiding energy waste.

Challenges (A.5): In order to improve reuse, such additional features are usually released as binary code on external libraries, which prevents programmers to understand the inefficiencies behind them. Thus, the application becomes a black-box, and when one energy inefficient component is invoked, developers without appropriated tools can no longer realize that. If the documentation does not clearly provide a useful description of the used library, it will become difficult for a programmer to identify such energy-intensive computations. The refactoring tool, then, should have the ability to detect such energy intensive functions, in particular when the source code is not available. However, this would require an in-depth investigation of all used external libraries, which in turn can be time consuming for the refactoring engine be practical.

Opportunity (A.6): Morden mobile applications are continuously-running, periodically sending and receiving data from servers. Such cumulatively behavior can greatly impact battery usage. Nikzad *et al.* [32] presents a technique that delays the execution of continuously-running power-hungry code fragments. To do so, developers must annotate these the places in the source code in which the execution should delayed and, at the appropriated time, the runtime system bulk these operations, reducing the cost of sending one at a time without sacrificing application integrity. The authors reported an energy savings of 63% when compared to the case when there is no coordination. Refactoring tools can ease the communication between programmers and the runtime system, as well as reducing the burden of writing such declarative annotation language.

Challenges (A.6): Due to the annotation language usage, the refactoring tool should be able to work with non-structured text files. Also, the refactoring tool should be integrated with the other existing refactorings. For instance, when a programmer is renaming a class, and if some of the methods of this class are already flagged to be used by the runtime system, the refactoring tool should be notified in order to update the configuration file to use the new fully qualified name.

B. Concurrent/Parallel programming

To better leverage multicore technology, applications must be concurrent, which poses a challenge, since it is well-known that concurrent programming is hard [44]. We found some papers studying the relationship between concurrent programming, performance and energy consumption [18], [39]. Even though no consensus has emerged from it, and despite the highly complex landscape, the authors identified some recurring patterns.

Opportunity (B.1): The ForkJoin framework [21], available since version 1.7 of the Java programming language, stands as a natural solution for parallel, fine-grained, divide-and-conquer algorithms, in particular, when recursion tasks are completely independent. However, Pinto *et al.* [39] observed that a great amount of energy consumption can be wasted if subtasks are copied instead of shared. The authors observed an energy saving amount of 15.38% when the shared approach is used.

Challenges (B.1): Refactoring concurrent programs is a daunting task because it inherits all the well-known refactoring problems, and it also comes with the cost of non-determinism. This can lead to the widely common concurrency bugs, in the form of unexpected new behavior, race conditions, deadlocks, or livelocks. Refactoring engines should take a particular care when applied for parallel programs. Another challenge is related to how to identify this pattern. Even though the programmer can make a copy using the `System.arraycopy()` method explicitly inside the parallel computation method, this is not the only possible scenario. Several utility classes in the `java.util` library make use of this method internally. One example is the `Arrays.copyOfRange()`. It can also be found in several collections method. Moreover, for modularity reasons, the programmer can use the `Arrays.copyOfRange()` method in another class/method, or it could be wrapped in a third-party library. The identification of such scenarios requires a sophisticated static analysis tool.

Opportunity (B.2): Kambadur and Kim [18] showed that well "parallelizable" problems can save a great amount of energy, up to 80% in an extreme case, when compared to the sequential version of the same program. Pinto *et al.* [39] also observed that a well-parallelizable problem is more energy-friendly than a sequential one. A refactoring engine can help programmers to identify and refactor such opportunity.

Challenges (B.2): First, not all kind of problems can be fully-parallelizable. Also, due to the natural shared-memory programming model present in high-level programming languages such as Java and C++, parallelism usually implies in coordination of shared locations, which in turn implies in synchronization. An automatic parallelization tool should analyze which places synchronization is needed, and apply them with extreme care, since synchronization can slow down the performance gained through parallelism. Also, the energy consumption of a multithreaded program is not an easy task to reason about. For instance, if a multi-threaded program receives a 2x speed-up but, at the same time, yields a fivefold increase in power consumption (as compared with a single core execution), energy consumption – the product of power consumption and execution time – and thus energy efficiency – the amount of work that can be achieved by consuming a certain amount of energy – degrades as the user embraces multi-core CPUs.

Opportunities (B.3): The use of Graphics Processing Units

(GPUs) for rendering is well-known, but their power for general parallel computation has only recently been explored. Scanniello *et al.* [43] have defined a strategy for transferring a CPU-intensive system to a Graphics Processing Unit (GPU) based architecture. Using this approach, the authors observed an improvement on energy consumption of over 60%. This refactoring is important because morden mainstream CPUs have only a few dozens of available cores. Conversely, GPUs offer the execution of many number of threads enabling to perform concurrently much more tasks.

Challenges (B.3): Again, the refactoring engine should detect the components that perform computationally intensive tasks. This component should be wrapped and integrated in a GPU system. Since GPU and CPU programming do not talk in the same language, the refactoring engine should be able to refactor the input system to a new language, which is not straightforward. Also, only a few programming languages have solid support for GPU programming, which reduces the options for the refactoring tool.

C. Preventing out-of-bounds errors

Since most of the established high-level programming languages do not prevent out-of-bound errors, that is, accessing an element outside of the array bounds, such errors become commonplace in the software development practice. Sophisticated techniques have been developed to mitigate this problem. When they succeed, however, the programmer pays a toll on both performance and energy consumption.

Opportunity (C.1): Nazaré *et al.* [31] have designed a static analysis tool technique for identifying out-of-bound error. This technique is implemented as an extension of a prior tool, AddressSanitizer, an industrial-quality tool. The main contribution of the authors is in eliminating several runtime checks used by AddressSanitizer to guard against out-of-bounds memory accesses, and thus improving the execution time and energy efficient in about 17% and 9%, respectively, when compared to the code originally produced by this tool.

Challenges (C.1): The technique implemented over the AddressSanitizer tool is complex and highly sophisticated. Additional layers can also introduce energy costs. In order to be useful, the refactoring should be seemly integrated with this tool.

D. Approximate programming

Many software applications offer the opportunity to tolerate occasional "soft errors", that is, errors that reduce the quality of service/solution. Such errors are welcome in certain applications, if they provide improvements on other system characteristics, such as an improvement on performance or a reduction in energy consumption. Many of these applications have one or more approximate computational kernels that consume the majority of the execution time. For instance, in a ray tracing implementation, the render method is likely to be the most time-consuming one.

Opportunities (D.1): Misailovic and colleagues [30] have introduced a framework for approximate programming namely Chisel. A valid Chisel program, is a program written in a high-level language such as C, and the instructions and variables stored in unreliable memories are written using Rely [6]. Chiesel then optimize the problem of selecting approximate instructions and variables allocated in approximate memories. Results have showed an energy savings from 8.7% to 19.8% in selected applications when compared to the original ones.

Challenges (D.1): This approach only works in emerging approximate hardware platforms. The refactoring tool should first verify if the current hardware has this support available. Also, the refactoring tool should provide means to developers understand the degree of error produced.

E. DVFS techniques

Dynamic Voltage and Frequency Scaling (DVFS) [37] is a common CPU feature where the operational frequency and the supply voltage of the CPU can be dynamically adjusted. It is one of the most effective power management strategy used in architecture research [22], [47]. In our selection of studies, two of them (*e.g.*, [8] and [3]) focus on using DVFS in a static way: providing as flags for runtime systems so that they can decide whether or not to scale the CPU frequency.

Opportunities (E.1): In the first study, Cohen *et al.* [8] have introduced a new type system to help reason about energy management. In this type system, the programmer is encouraged to think how CPU-intensive each program statement is. Take for example a system of which performs a long-running math calculation, but also performs some HTTP operations on the meantime. Using this new type system, a programmer can declare a partial order `phases { http <cpu math; }`, meaning "http is less CPU-intensive than math". This definition encourages programmers to contribute in their knowledge, so that DVFS calls are inserted automatically, and the decision of scaling down/up is conducted by the compiler based on the partial order. The authors reported an energy saving of 30%-50% in selected benchmarks. Refactoring tools can help programmers to update their code to use this type system.

Challenges (E.1): The introduced type system has two important language constructs: phases and modes. Such language constructs can be introduced at the class level, method level and variable level. Refactoring tools should take care of which one should be used, and when, which would require sophisticated def-use analysis.

Opportunities (E.2): In their study, Bartenstein *et al.* [3] propose a technique for reducing the energy consumption of a stream program. Stream programming is a general-purpose paradigm where software is composed as a stream graph, which is parallelism friendly. The proposed approach is based on a key insight of stream programming: a stream graph can operate more efficiently if the rates of streams are coordinated, so that, one filter may output a data item to a stream "just-in-time" for consumption by the next filter on the receiving end of the stream. Using this approach, the authors reported an average CPU energy saving of 28%.

Challenges (E.2): Existing refactoring tools cannot be reused to support this new refactoring, since most of the existing refactoring implementations encompass only control-flow-centric programming models (such as Java and C) while the stream programming is graph-centric model, implemented by the StreamIt programming language. Transformation from a control-flow-centric programming model to streams would require the refactoring to perform a great set of transformations, which can in turn weight in its applicability, due its time-consuming nature.

F. Software testing

As a means of ensuring the reliability of a software, software testing have become one fundamental activity during the software development process. Software engineers are often motivated to write several test cases to their programs. In nontrivial applications, however, executing test cases can be an extremely time-consuming due to the great number of them. Selecting the most important test cases to be executed, without losing testing coverage, is a topic of great interest that researchers have extensively worked on. This is a particular concern for embedded software, which has to routinely perform test cases on a live deployed system, which in turn has its life-time limited to battery power.

Opportunities (F.1): Li *et al.* [25] proposed a technique used to minimize the energy consumption of test suites. This technique selects test cases based not only on their coverage but also on their energy usage. Results revealed that the technique is effective at generating test suites that consume up to 95% less energy, while maintaining testing coverage requirements. Prior to deploy, a refactoring engine can be used to select only useful high-quality energy efficient test cases.

Challenges (F.1): First, the refactoring engine should be seemly integrate with the proposed approach. Since the proposed approach needs to modify the existing test suite, this should be done automatically by the refactoring tool. Second, the energy savings of the proposed approach is based on the quality of the test cases. If the test suite doe not have enough quality, the refactoring can impact negatively on the testing coverage. Such analysis should be performed by the refactoring engine. Finally, since test suites written in an ad hoc manner are not supported by the proposed approach, the refactoring engine should also be capable to refactor such test cases to use a high-level framework, such as JUnit or TestNG.

IV. Related Work

In this section we describe the studies overlapping with the scope of our work.

The most established energy management approaches are focused on the hardware level (*e.g.*, [16]) and the OS level (*e.g.*, [51]). Tiwari *et al.* [45] correlated energy consumption with CPU instructions. Vijaykrishnan *et al.* [49] performed an early study on the energy consumption of the JVM. Within the programming language community, it is an active area of research to design energy-aware programming languages, with examples such as EnerJ [42], Energy Types [8], and LAB [19]. Existing research that dealt with the trade-off of comparing individual characteristics of an application and energy consumption has covered a wide spectrum of applications. These characteristics vary from data structures [29], VM services [5], cloud offloading [20], and code obfuscation [41].

The mobile arena is also an important topic of research. Hindle [14] investigated the relationship between software changes and power consumption on Mozilla Firefox. The author observed that intentional performance optimization introduced a steady reduction in power consumption. More recently, Hindle *et al.* [15] proposed an energy consumption framework to be used in mobile devices. The authors suggest that this framework is more accurate than real meters in measuring energy consumption of smartphones because it does not take the battery usage in consideration. Pathak *et al.* [36] presented an in-depth investigation in order to understand which is the root cause for energy consumption problems in mobile applications. Like the study of Pinto *et al.* [38], they also observed that advertisement plays an important role, consuming up to 75% of energy consumption in free apps.

The energy consumption of concurrent programs is gaining attention over the years. Park *et al.* [35] developed several synchronization-aware runtime techniques to balance the trade-off between energy and performance. Gautham *et al.* [11] studied the relative energy efficiency of synchronization implementation techniques (such as spin locks and transactions). A recent short paper [28] called for energy management based on different synchronization patterns. Trefethen and Thiyagalingam [46] surveyed energy-aware software, including multi-threaded programs with different workload settings. Bartenstein and Liu [3] designed a data-centric approach to improve energy efficiency for multi-threaded stream programs. Ribic and Liu [40] designed an algorithm to improve the energy efficiency of the work-stealing runtime of Intel Cilk Plus by managing the relative speed of threads.

V. Threats to Validity

First, in presenting our analysis on opportunities and challenges for refactoring for energy efficiency, we attempted to select and cite relevant papers published on premier software engineering venues. However, the conducted survey is not particularly exhaustive; we make no pretense of having selected and cited all possible energy consumption related studies. We leave this task to surveys and systematic studies. Instead, the presented work consists in highlighting refactoring opportunities to improve existing systems. Also, by focus only on premiere software engineering conferences, and not on workshops, we believe we can select more mature and stablished research. Second, even though we have searched in 8 software engineering conferences, this paper does not mean to make general conclusions about the entire software energy consumption community. Rather, results should only be viewed in the context of the papers in those 10 years of chosen conferences. Nevertheless, we truly hope that the approach we followed helped this paper better reflect the views of researchers in the software energy consumption community. Finally, another threat is related to the procedure we use for select and classify our themes for opportunities and challenges. Since we did not use a qualitative analysis in order to record themes within data, our

VI. Conclusions

Our goal in this paper was to present and discuss some of the opportunities to make refactoring green, allowing programmers to write energy efficient software. To do that, we have analyzed recent software energy consumption literature. For each one of the selected papers, we describe how the main contribution can be reincarnated in a refactoring engine, and what are the challenges to do so. In a nutshell, Table II reviews the main opportunities.

TABLE II
Summary of identified refactoring opportunities.

Opportunity	Supported By
User interfaces	[26]
Cloud computing	[20]
HTTP requests	[23], [33], [12]
Software Piracy	[41]
I/O operations	[2], [23]
Continuously running app	[32]
Excessive copy chains	[39]
Embrace parallelism	[18], [39]
GPU programming	[43]
Out-of-bounds errors	[31]
Approximate programming	[6]
DVFS techniques	[8], [3]
Software testing	[25]

For future work, we plan to effectively implement some of the aforementioned opportunities in a refactoring engine. In order to evaluate the effectiveness of the refactoring tool, we also plan to conduct controlled experiments with practitioners, so that we can observe if the refactoring tool is not only efficient in improving the energy consumption, but also usable.

VII. Acknowledgments

We would like to thank the anonymous reviewers for their helpful comments. Gustavo is supported by CAPES/Brazil, and Fernando is supported by CNPq/Brazil (306619/2011-3, 487549/2012-0 and 477139/2013-2), FACEPE/Brazil (APQ-1367-1.03/12) and INES (CNPq 573964/ 2008-4 and FACEPE APQ-1037-1.03/08). Any opinions expressed here are from the authors and do not necessarily reflect the views of the sponsors.

References

[1] L. Ardito, G. Procaccianti, M. Torchiano, and A. Vetrò. Understanding green software development: A conceptual framework. *IT PROFESSIONAL*, pages 1–6, 2015.

[2] A. Banerjee, L. K. Chong, S. Chattopadhyay, and A. Roychoudhury. Detecting energy bugs and hotspots in mobile apps. In *FSE*, pages 588–598, 2014.

[3] T. W. Bartenstein and Y. D. Liu. Green streams for data-intensive software. In *ICSE*, pages 532–541, 2013.

[4] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change (2Nd Edition)*. Addison-Wesley Professional, 2004.

[5] T. Cao, S. M. Blackburn, T. Gao, and K. S. McKinley. The yin and yang of power and performance for asymmetric hardware and managed software. In *ISCA*, pages 225–236, 2012.

[6] M. Carbin, S. Misailovic, and M. C. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In *OOPSLA*, pages 33–52, 2013.

[7] A. Carroll and G. Heiser. An analysis of power consumption in a smartphone. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[8] M. Cohen, H. S. Zhu, E. E. Senem, and Y. D. Liu. Energy types. In *OOPSLA*, pages 831–850, 2012.

[9] D. Dig, J. Marrero, and M. D. Ernst. Refactoring sequential java code for concurrency via concurrent libraries. In *ICSE*, pages 397–407, 2009.

[10] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[11] A. Gautham, K. Korgaonkar, P. Slpsk, S. Balachandran, and K. Veezhinathan. The implications of shared data synchronization techniques on multi-core energy efficiency. In *Proceedings of the*, HotPower'12, 2012.

[12] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan. Estimating mobile application energy consumption using program analysis. In *ICSE*, pages 92–101, 2013.

[13] M. Hilton, A. Christi, D. Dig, M. Moskal, S. Burckhardt, and N. Tillmann. Refactoring local to cloud data types for mobile apps. In *Proceedings of the 1st International Conference on Mobile Software Engineering and Systems*, MOBILESoft 2014, pages 83–92, 2014.

[14] A. Hindle. Green mining: A methodology of relating software change to power consumption. In *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, pages 78–87, 2012.

[15] A. Hindle, A. Wilson, K. Rasmussen, E. J. Barlow, J. C. Campbell, and S. Romansky. Greenminer: A hardware based mining software repositories software energy consumption framework. In *MSR*, pages 12–21, 2014.

[16] M. Horowitz, T. Indermaur, and R. Gonzalez. Low-power digital design. In *Low Power Electronics, 1994. IEEE Symposium*, 1994.

[17] J. Jelschen, M. Gottschalk, M. Josefiok, C. Pitu, and A. Winter. Towards applying reengineering services to energy-efficient applications. In *CSMR*, pages 353–358, 2012.

[18] M. Kambadur and M. A. Kim. An experimental survey of energy management across the stack. In *OOPSLA*, pages 329–344, 2014.

[19] A. Kansal, T. S. Saponas, A. J. B. Brush, K. S. McKinley, T. Mytkowicz, and R. Ziola. The latency, accuracy, and battery (LAB) abstraction: programmer productivity and energy efficiency for continuous mobile context sensing. In *OOPSLA*, pages 661–676, 2013.

[20] Y. Kwon and E. Tilevich. Reducing the energy consumption of mobile applications behind the scenes. In *ICSM*, pages 170–179, 2013.

[21] D. Lea. A java fork/join framework. In *Proceedings of the ACM 2000 Conference on Java Grande*, JAVA '00, pages 36–43, 2000.

[22] H. Lebreton and P. Vivet. Power modeling in systemc at transaction level, application to a dvfs architecture. In *Proceedings of the 2008 IEEE Computer Society Annual Symposium on VLSI*, ISVLSI '08, pages 463–466, 2008.

[23] D. Li, S. Hao, J. Gui, and W. G. J. Halfond. An empirical study of the energy consumption of android applications. In *ICSME*, pages 121–130, 2014.

[24] D. Li, S. Hao, W. G. J. Halfond, and R. Govindan. Calculating source line level energy information for android applications. In *ISSTA*, pages 78–89, 2013.

[25] D. Li, Y. Jin, C. Sahin, J. Clause, and W. G. J. Halfond. Integrated energy-directed test suite optimization. In *ISSTA*, pages 339–350, 2014.

[26] D. Li, A. H. Tran, and W. G. J. Halfond. Making web applications more energy efficient for oled smartphones. In *ICSE*, pages 527–538, 2014.

[27] K. Liu, G. Pinto, and D. Liu. Data-oriented characterization of application-level energy optimization. In *FASE*, 2015.

[28] Y. D. Liu. Energy-efficient synchronization through program patterns. In *First International Workshop on Green and Sustainable Software, GREENS 2012, Zurich, Switzerland, June 3, 2012*, pages 35–40, 2012.

[29] I. Manotas, L. Pollock, and J. Clause. Seeds: A software engineer's energy-optimization decision support framework. In *ICSE*, pages 503–514, 2014.

[30] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard. Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. In *OOPSLA*, pages 309–328, 2014.

[31] H. Nazaré, I. Maffra, W. Santos, L. Barbosa, L. Gonnord, and F. M. Quintão Pereira. Validation of memory accesses through symbolic analyses. In *OOPSLA*, pages 791–809, 2014.

[32] N. Nikzad, O. Chipara, and W. G. Griswold. Ape: An annotation language and middleware for energy-efficient mobile application development. In *ICSE*, pages 515–526, 2014.

[33] A. Noureddine, A. Bourdon, R. Rouvoy, and L. Seinturier. Runtime monitoring of software energy hotspots. In *ASE*, pages 160–169, 2012.

[34] J. Overbey, S. Xanthos, R. Johnson, and B. Foote. Refactorings for Fortran and High-Performance Computing. In *Second International Workshop on Software Engineering for High Performance Computing System Applications*, May 2005.

[35] S. Park, W. Jiang, Y. Zhou, and S. V. Adve. Managing energy-performance tradeoffs for multithreaded applications on multiprocessor architectures. In *SIGMETRICS*, pages 169–180, 2007.

[36] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with eprof. In *EuroSys*, pages 29–42, 2012.

[37] T. Pering, T. Burd, and R. Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proceedings of the 1998 International Symposium on Low Power Electronics and Design*, ISLPED '98, pages 76–81, 1998.

[38] G. Pinto, F. Castor, and Y. D. Liu. Mining questions about software energy consumption. In *MSR*, pages 22–31, 2014.

[39] G. Pinto, F. Castor, and Y. D. Liu. Understanding energy behaviors of thread management constructs. In *OOPSLA*, pages 345–360, 2014.

[40] H. Ribic and Y. D. Liu. Energy-efficient work-stealing language runtimes. In *ASPLOS*, pages 513–528, 2014.

[41] C. Sahin, P. Tornquist, R. Mckenna, Z. Pearson, and J. Clause. How does code obfuscation impact energy usage? In *ICSME*, pages 131–140, 2014.

[42] 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 *PLDI*, pages 164–174, 2011.

[43] G. Scanniello, U. Erra, G. Caggianese, and C. Gravino. Using the gpu

[43] G. Scanniello, U. Erra, G. Caggianese, and C. Gravino. Using the gpu to green an intensive and massive computation system. In *CSMR*, pages 384–387, 2013.

[44] H. Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, Sept. 2005.

[45] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: A first step towards software power minimization. *IEEE Transactions on VLSI Systems*, 2:437–445, 1994.

[46] A. Trefethen and J. Thiyagalingam. Energy-aware software: Challenges, opportunities and strategies. *Journal of Computational Science*, 1(0):–, 2013.

[47] S.-Y. Tseng and M.-W. Chang. Dvfs aware techniques on parallel architecture core (pac) platform. In *Proceedings of the 2008 International Conference on Embedded Software and Systems Symposia*, ICESSSYMPOSIA '08, pages 79–84, 2008.

[48] B. Vasilescu, A. Serebrenik, T. Mens, M. G. J. van den Brand, and E. Pek. How healthy are software engineering conferences? *Sci. Comput. Program.*, 89:251–272, 2014.

[49] N. Vijaykrishnan, S. Kim, S. Tomar, A. Sivasubramaniam, and M. J. Irwin. Energy behavior of java applications from the memory perspective. In *in Usenix Java Virtual Machine Research and Technology Symposium (JVM01*, pages 207–220, 2001.

[50] C. Wilke, S. Richly, S. Gotz, C. Piechnick, and U. Assmann. Energy consumption and efficiency in mobile applications: A user feedback study. In *GreenCom*, pages 134–141, 2013.

[51] W. Yuan and K. Nahrstedt. Energy-efficient soft real-time cpu scheduling for mobile multimedia systems. In *SOSP*, pages 149–163, 2003.

[52] Y. Zhang, G. Huang, X. Liu, W. Zhang, H. Mei, and S. Yang. Refactoring android java code for on-demand computation offloading. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 233–248, 2012.