Refactoring Multicore Applications Towards Energy Efficiency

Gustavo Pinto
Informatics Center
Federal University of Pernambuco
Recife, PE, Brazil
ghlp@cin.ufpe.br

Abstract

Great strides have been made to increase the energy efficiency of hardware, data center facilities, and network infrastructure. However, in any computer system, it is software that directs much of the activity of the hardware. Moreover, multicore processors have become ubiquitous, mainly because their multiple benefits, especially enhanced performance for multi-threaded and compute-intensive applications. Nonetheless, there are few studies addressing the topic of restructuring multicore applications to consume less energy and even fewer that leverage developer expertise to achieve that goal. In this thesis we present a brief background study for refactoring multicore applications in order to improve both performance and energy consumption. The idea consists in proposing a catalog of refactorings targeting some languages of the JVM platform.

Categories and Subject Descriptors D.2.7 [*Software Engineering*]: Restructuring, reverse engineering, and reengineering; D.1.3 [*Programming Techniques*]: Concurrent Programming, Parallel Programming

Keywords Refactoring, Concurrent/Parallel Programming, Energy Consumption

1. Motivation

In spite of advances in many areas, IT energy consumption keeps rising steeply [1], which indicates that rising demand is outpacing efficiency improvement. Nonetheless, for many years, research that connects computing and energy efficiency has concentrated on the hardware layer. These studies are motivated by the assumption that only hardware dissipates power, not software. However, there are studies that show that this assumption does not capture the whole picture [2, 8]. That would be analogous to postulating that only automobiles are responsible for burning gasoline, not the people who drive them and the way they are used. In any computer system, it is software that directs much of the activity of the hardware. Consequently, software can have a substantial impact on power consumption.

SPLASH '13, October 26–31, 2013, Indianapolis, Indiana, USA.
Copyright is held by the owner/author(s).
ACM 978-1-4503-1995-9/13/10.
http://dx.doi.org/10.1145/2508075.2514880

Software solutions for improving energy efficiency of computer systems can work at different levels, ranging from machine code level to end-user applications. Notwithstanding, concerns about energy usage were left for compiler writers, operating system designers and hardware engineers. Nonetheless, energy efficiency for higher layers of the software stack, in particular at the application level, is a subject that has been the target of only few studies [3, 9, 13].

Measuring the energy consumption of high level applications and understanding where the energy usage lies provides new opportunities for energy savings. In order to understand the complexities of this approach, we specifically look at multi-threaded applications, since multicore processors have become ubiquitous. The performance of the existing constructs for concurrenty[1] execution is reasonably well-understood [10, 14], but little is known about its energy efficiency. Furthermore, since concurrent/parallel programming enables programmers to run their applications faster, a common belief is that this application will also consume less energy [4]. Nonetheless, some researchers [7, 12] have shown that it is not necessarily true. This contradiction poses a challenge, creating the need of new research with deeper analysis.

In order to reduce the energy consumed, many communities can benefit from the results of this work: (i) large corporations which spend too much money on energy consumption derived from software applications; (ii) tool vendors could also benefit, because they can increase their products' value by implementing the refactorings described in this work; and (iii) indirectly, every software user could be benefit, since their favorite applications will be able to run faster, resulting in an overall decrease of energy, besides showing a better user-experience.

2. Problem

Considering the great number of concurrent applications currently in use, existing software systems should be refactored to consumes less energy. Hitherto, however, there are only few studies addressing the topic of restructuring existing concurrent high level applications to consume less energy [4], and even fewer that leverage developer expertise to achieve that goal. Fortunately, there is opportunity for substantial reductions in the energy consumption of existing applications.

Thus, the overall aim of this study is twofold: (i) To better understand the relationship between concurrent programming and energy consumption, and (ii) to derive a refactoring catalog of energy code smells for concurrent software. The first task is currently on development and preliminary results can be found in [7]. The general conclusion we can draw from it is that the trade-off between

[1] Throughout the paper, we often employ the terms "concurrent" and "parallel". Since the Java language does not have specific constructs for each abstraction, we use these terms interchangeably.

performance and energy consumption in multicore applications is not obvious, besides be very difficult to generalize. However, more experiments are planned to be conducted. The results of these experiments will be used as an input for a second study. Then, derive a catalog is currently the most important task and it presents the exact problem that the author will address.

The knowledge gained from these experiments will then be applied in the form of guidelines to programmers, in order to educate application programmers to safely restructure their applications to both improve performance and energy consumption.

2.1 Refactoring for Multicore and Energy Efficiency: Opportunities and Challenges

Refactoring an application to use concurrent constructs with the goal of improving performance does not necessarily imply on energy savings, even if there is a reduction in the execution time of the application [12]. Using the Java language as an example, programmers have at their disposal numerous constructs to create and manage concurrent/parallel applications.

Nevertheless, each one of them has its own advantages and disadvantages. For instance, the `Thread` class is the primary, and the simplest, construct to create a new thread [11]. If a programmer chooses to use these constructs, it will be necessary to worry about a number of tasks, such as (i) to manage thread creation and termination, and related operations; (ii) to choose the number of threads that should be initialized, which can vary with the number of available CPUs; (iii) to decide at what moment they should be initialized; (iv) to implement sophisticated mechanisms to reuse threads; and (v) to control access shared resources. As a consequence of having perform all the tasks, programmers often misuse concurrent constructs, which may result in wast of energy resources, and/or deterioration in the application's performance (i.e. code runs sequentially instead of concurrently [6]).

On the other hand, developers could choose enhanced threading constructs, such as those provided by the java.util.concurrent (j.u.c) library, available since version 1.5 of the language. Using high-level constructs instead of low-level threads has many benefits: not only are they less error-prone, but they also have better performance in some situations, at the expense of being less general. Moreover, recently was released the new version (1.7) of the j.u.c. library, with some improvements and new concurrent constructs, such as the `ForkJoin` framework [5]. Furthermore, the Java Virtual Machine also offer others concurrent-friendly programming languages, such as Scala. The Scala language is a highly productive programing language combining functional and object-oriented programming. It incorporates ground-breaking features supporting asynchronous programming.

To the best of our knowledge, there is no work in the literature that assesses the energy efficiency of the concurrent/parallel constructs of the Java Virtual Machine platform, considering more than one language. In addiction, to the best of our knowledge, there is an absence of any refactoring cookbook that supports developers in using different constructs, based on the performance and the power consumption requirements of an application.

3. Approach

This study aims to identify new methods, techniques and tools for refactoring programs in order to improve the energy consumption and still achieve a better performance on multi-core platforms. Its main expected outcome is a catalog of refactorings targeting two languages of the Java Virtual Machine platform: Java and Scala. More specifically, the set of refactorings that we intend to propose will support developers in employing different techniques to manage the execution of units of work that can be performed in parallel.

For example, a typical example of a refactoring to energy efficiency is between `Thread` and tasks from `ForkJoin` framework. It is well-known that Thread has high overhead (creating, scheduling, destroying, etc) which might outperform the useful computation. On the other hand, the `ForkJoin` is a lighter-weight thread-like entity, which could host a large number of tasks in a pool of small number of threads. Nonetheless, given the nature of divide-and-conquer algorithms, tasks that run in parallel should have the following characteristics: (i) are CPU-bound, not IO-bound; (ii) depending on the sequential threshold, and (iii) only need to synchronize when waiting for subtasks to complete. If the above conditions are satisfied, the refactoring could be applied.

Another, similar example is the refactoring between `Thread` and `Scala Actors`. However, as we reported elsewhere, a number of factors could impact these refactorings. We intend to investigate each case fully to determine if the refactoring will be beneficial.

Thus, to better understand the open problems of this area, that is, the refactoring approaches that could be derived, the author initially carried out a literature review to fully understand current issues within concurrent programming, as well as the energy consumption of high level application. The findings from this review are discussed in section 1 and 2 above, and highlight the importance of this topic.

An initial investigation was carried out to understand the various concurrent construct under several workloads. Previous results show that it is not as straightforward as it would seem. One reason to this, is because each language construct has its pros and cons. For instance, the `ForkJoin` framework, which is highly indicated to fine-grained parallelism, should be avoided in applications that make use of synchronized methods or blocks, or other blocking synchronization apart from joining other tasks and when performing blocking IO. To understanding which scenario a given transformation could achieve better results, for both performance and energy, is part of this work.

This study focus on two mainstream languages that are part of the Java Virtual machine platform: Java and Scala. The Java language was chosen because it is widespread in both academia and industry. Moreover, Java offers several high-level libraries that implement complex concurrent algorithms, besides the traditional model based on threads and shared data. On the other hand, Scala is a mixed object-functional language, which is very interesting to use to solve problems concurrently, since Scala can eliminate side-effects, and thus, problems like race conditions. Scala also has its own interesting properties for concurrent programming. One simple way to write concurrent programs is with Scala actors. The general approach is to create actors that represent the computation that you want to run asynchronously, and then start these computations by using one of the triggering functions. One key thing about the actors programming model is that it provides primitives that are scalable and you can be use for both CPU and I/O computations.

4. Evaluation Methodology

The author is currently in the process of assessment of the energy efficiency of concurrent constructs. A number of experiments have been done, and few more experiments should be conducted.

4.1 Hypothesis

This study has two hypotheses. The primary hypothesis is that it is possible to generalize the relationship between performance and energy consumption in concurrent programs. The secondary hypothesis is that programmers can safely refactor their applications in order to improve both performance and energy efficiency. Our benefits include: (i) a better understanding of the energy efficiency of concurrent constructs in the JVM; (ii) a set of factors that could also imply on the energy efficiency of a given application, and (iii) a catalog of refactorings that will help application programmers to better use energy resource.

4.2 Experiment Setup

The experiment setup of this study will be conducted by performing the following activities:

- Measurement: To assess the energy efficiency of the most common concurrent programming constructs. For example, in the Java language, `Thread` and `Runnable` classes are the most important concurrent constructs [11]. Initially, experiments to measure the energy efficiency of these constructs targeted programs from a set of benchmarks, since it provides a wide range of concurrent applications that employ varied constructs.

  Preliminary experiments have been executed and show that it is possible to switch from one technique to another in order to consume less energy [7]. Nonetheless, we also conclude that it is very hard to identify which technique is the better for a given scenario. Moreover, our experiments show that factors such as the nature of the problem to be solved, the technique used to manage concurrent execution, the CPU clock frequency, and the JVM implementation can create variations. For example, for a CPU-bound problem, the `ForkJoin` framework can perform better while using less energy. At the same time for a heavily IO-bound problem, it can consume about 10% more energy than a sequential implementation, although still 30% faster. We also noticed that the choice of the JVM implementation can increase the energy consumption in more than 10%.
- Study Design: The refactorings from the catalogue are accordingly organized in three groups, each focusing on one of the refactoring phases. Their phases are (i) identify code which actually runs concurrently; (ii) identify which constructs could be replaced in order to improve performance and energy efficiency; (iii) extract code into the new concurrent construct.
- Evaluation by specialists: To assess the generality of the refactoring catalog, the refactorings will be applied to an existing system by a specialist. We will document the scenarios where the refactorings can not be applied and use this information to analyze whether they need to be more general and, if so, how that goal can be achieved.
- Evaluation by controlled experiments: Controlled experiments with graduate and undergraduate students. These studies will compare the performance of students explicitly instructed in the use of the transformations against students performing adhoc restructuring. Performance will be measured in terms of the time to complete the assignment and the number and nature of the bugs in the code, if any.

5. Future Work

As future work, we are planning to implement a tool in order to to aim programmer to refactoring their application automatically. This tool should be integrated with well-known IDE, such as Eclipse and InteliJ. Moreover, we need to adapt our approach to use other JVM-languages. Thus, this allows us to go beyond the traditional refactoring approach in order to propose refactorings between languages. We also intend to investigate how this approach can be used to improve the battery life of Android phones.

6. Acknowledgments

I would like to thank Professor Fernando Castor (my supervisor) for the fruitful discussions we had about the ideas of my PhD thesis proposal, Francisco Neto-Soares for the valuable reviews, and the OOPSLA anonymous reviewers for their helpful comments. This work is supported by Brazilian funding agencies. Gustavo is supported by CAPES and Fernando is supported by CNPq (306619/2011- 3), FACEPE (APQ-1367-1.03/12), and by INES (CNPq 573964/2008-4 and FACEPE APQ-1037- 1.03/08).

References

[1] J. Asafu-Adjaye. The relationship between energy consumption, energy prices and economic growth: time series evidence from asian developing countries. *Energy Economics*, 22(6):615 – 625, 2000. ISSN 0140-9883. .

[2] M. Cohen, H. S. Zhu, E. E. Senem, and Y. D. Liu. Energy types. In *Proceedings of the 13th OOPSLA*, pages 831–850, 2012.

[3] M. G., M. J., J. J., and A. W. Removing energy code smells with reengineering services. In *Beitragsband der 42. Jahrestagung der Gesellschaft fr Informatik e.V. (GI)*, volume 208, pages 441–455. Bonner Kllen Verlag, 2012.

[4] 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.

[5] D. Lea. A java fork/join framework. In *Java Grande*, pages 36–43, 2000.

[6] S. Okur and D. Dig. How do developers use parallel libraries. In *Proceedings of the 21st ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2012.

[7] G. Pinto and F. Castor. On the implications of language constructs for concurrent execution in the energy efficiency of multicore applications. In *OOPSLA Companion*, 2013. to appear.

[8] A. S., W. D., E. F., D. G., L. C., and D. G. Enerj: approximate data types for safe and general low-power computation. In *Proceedings of the 32nd*, PLDI '11, pages 164–174, 2011. ISBN 978-1-4503-0663-8.

[9] C. Sahin, F. Cayci, I. Gutiérrez, J. Clause, F. Kiamilev, L. Pollock, and K. Winbladh. Initial explorations on design pattern energy usage. In *GREENS*, pages 55–61, 2012.

[10] L. A. Smith, J. M. Bull, and J. Obdrzálek. A parallel java grande benchmark suite. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '01, pages 8–8, New York, NY, USA, 2001. ACM. ISBN 1-58113-293-X.

[11] W. Torres, G. Pinto, B. Fernandes, J. a. P. Oliveira, F. A. Ximenes, and F. Castor. Are java programmers transitioning to multicore?: a large scale study of java floss. In *Proceedings of the Transitioning to multicore (TMC'11)*, SPLASH '11, pages 123–128. ACM, 2011.

[12] A. Trefethen and J. Thiyagalingam. Energy-aware software: Challenges, opportunities and strategies. *Journal of Computational Science*, 1(0):–, 2013. ISSN 1877-7503. .

[13] Y. Zhang, G. Huang, X. Liu, W. Zhang, H. Mei, and S. Yang. Refactoring android java code for on-demand computation offloading. In *OOPSLA*, pages 233–248, 2012.

[14] W. Zhu, J. del Cuvillo, and G. R. Gao. Performance characteristics of openmp language constructs on a many-core-on-a-chip architecture. In *IWOMP*, pages 230–241, 2006.