Evaluation of the impact of code refactoring on embedded software efficiency

Wellisson G. P. da Silva[1], Lisane Brisolara[1], Ulisses B. Corrêa[2], Luigi Carro[2]

[1]Universidade Federal de Pelotas (UFPEL) - Pelotas – RS – Brazil

[2]Universidade Federal do Rio Grande do Sul (UFRGS) - Porto Alegre – RS – Brazil

wguilhermino@gmail.com, lisane.brisolara@ufpel.edu.br,
ubcorrea@inf.ufrgs.br, carro@inf.ufrgs.br

Abstract. *The increasing complexity of embedded software and the hard time-to-market requirements, motivate to the use of object-oriented languages. However, this usage can negatively impact on energy consumption as well as on performance. Code refactoring are techniques that change the code in order to improve the software quality. This paper analyzes how the inline method refactoring, a software optimization technique, can impact on the performance and energy of embedded software written in Java. Three different applications are evaluated in order to discuss this impact.*

1. Introduction

Complex embedded systems have hard constraints regarding performance, memory, and energy and power consumption [1]. Although these physical properties are more closed to hardware, the way the software interacts with system resources has impact on power and energy consumption as well as on performance [2]. Moreover, the embedded system domain is driven by cost and time-to-market factors [3], which also influence the design decisions taken by embedded software developers.

To handle the increasing complexity of embedded software and the hard time-to-market requirements, the use of object-oriented languages became more important mainly due its modular and reusable code. However, object-oriented (OO) languages can introduce penalties to system power and energy consumption and performance [3]. In this scenario, embedded software designers should handle the software complexity and produce efficient software at the same time.

Refactoring is a software engineering technique that modifies the code to improve its readability and maintainability without changing its computation [4]. Method inline is a common refactoring method as well as a well-known code optimization technique. This work discusses the impact of the method inline on system performance and energy consumption when it is applied to Java codes. The objective is help designers to understand how OO practices affect these physical properties and evaluate if refactoring is a valid strategy to explore the embedded software design space.

The remaining of this paper is organized as follows. Section 2 gives the background. Section 3 presents the used methodology and target platform, while results are discussed in Section 4. Conclusions and future work are presented in Section 5.

2. Background

There are three main sources of power consumption in embedded systems [3], which are: processor power consumption, due to the processor activity, memory power consumption, for accessing data and instructions in memory, and the power consumption to connect the processor and memories. All these operations should be taken in account, when developing embedded software. Moreover, some software engineering practices can directly impact system power consumption and performance, like code refactoring.

Refactoring is a process of modifying the code to make it easier to understand and modify without changing its computation as defined in [4]. These code changes can also affect the system performance and energy consumption, since it modifies the instructions that will be used. Examples of changes that can be done are Extract Method – which creates a method to represent duplicate code – or Inline Method – which exchanges a method call for its body. Inline is not recommended by the software engineering best practices, because it can make the code hard to understand and so decrease code maintainability. However, the Inline Method is expected to increase performance since it removes the overhead of a method call – each method call in the Java Virtual Machine has a cost associated [5]. This work has as objective to analyze how method inline impacts performance and energy consumption for Java codes.

3. Methodology and Target Platform

Through dynamic profiling of the application code, information about the number of method calls are obtained then we apply the method inlines incrementally. In this way, the first most frequently called method is inlined in the first iteration (modifying original code and generating *Iteration1* version) and the second one is inlined generating the *Iteration2* version, and so on. Thus, the gains achieved by the reduction of method invocations can be increased. After that, several versions of the same application are generated.

In order to obtain performance and energy consumption for these several Java code versions, an estimation tool called DESEJOS was used [6]. As embedded platform, we adopted the FemtoJava [7] processor. This processor is a stack based Java Virtual Machine implementation, executing Java bytecodes natively. We chose FemtoJava and DESEJOS due to their use of Java, a language that is gaining attention on the embedded community.

The FemtoJava processor implements a stack machine compatible with the Java Virtual Machine (JVM) specification and that is able to execute Java code in hardware. Two different versions of the FemtoJava processor are available: multicycle and pipeline. In this experiment, we adopted the multicycle version that is targeted to low power embedded applications.

4. Experimental Results

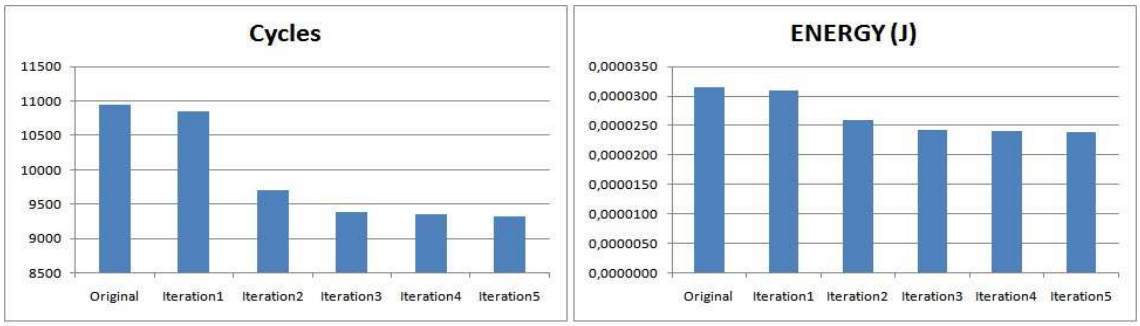
Three applications are used as case studies, an address book implemented with a hash table, a game called Sokoban, and also an Mpeg layer-3 audio decoder from the Spec jvm2008 benchmark set [8]. Table 1 presents values of metrics obtained by the Eclipse IDE Plug-in called Metrics [9] for each application. These metrics gives an idea about the complexity of the studied applications. According to these metrics, the MPEG

decoder is the used application with higher complexity, since it presents the higher number of classes, objects, and packages, besides the higher McCabe ciclomatic complexity, well-know measure for the complexity of a program.

Table 1. Complexity of the applications

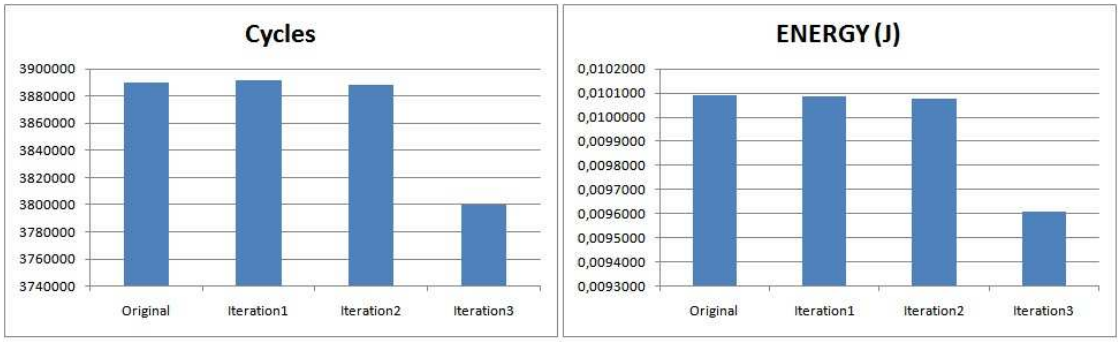
	Address Book	Sokoban	MPEG Decoder
Number of classes	2	7	51
Number of methods	31	49	184
Number of packages	1	2	80
McCabe Ciclomatic Complexity	37	109	120

Figure 1 (a) and (b) and Figure 2 (a) and (b) present performance (in cycles) and energy consumption (in Joule) for the different code versions, generated by inline refactoring, of the Address book and of the Sokoban, respectively. These results show that this refactoring method achieved improvements in performance and energy consumption for both applications.



(a) Performance results of the **Address book** (b) Energy consumption results of the **Address book**

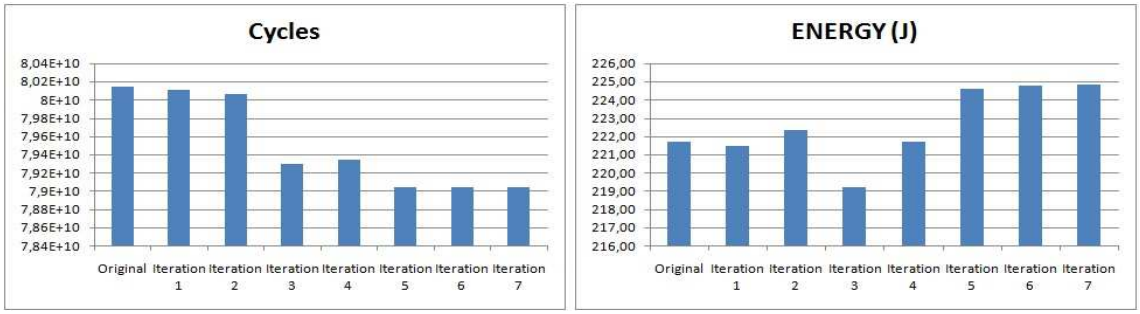
Figure 1: Performance and Energy consumption results for the Addrees Book



(a) Performance results of the **Sokoban** (b) Energy consumption results of the **Sokoban**

Figure 2: Performance and Energy consumption results for the Sokoban

Figure 3 (a) and (b) illustrates the results for performance (in cycles) and energy consumption (in Joule) achieved for the different code versions of the MPEG decoder. The performance results show that the cycles required by the application are reduced when the inline method is applied, as expected, since cycles used to create frames for each method call are eliminated. However, the energy results of the MPEG-Audio decoder were not as expected (see Fig. 3b). After the third refactoring iteration, which has a greater gain from its antecessors, all remaining versions have higher consumption and the last ones have a consumption worse than the original code (without any inline applied).



(a) Performance results of the **MPEG decoder** (b) Energy consumption of the **MPEG decoder**

Figure 3: Performance and Energy consumption results for the MPEG decoder

To better analyze the achieved results for energy consumption, we have analyzed the Java bytecodes and have obtained the instruction histogram for the execution of the different code versions in which the results are not the expected. Summarized results of these histograms can be observed in Table 2 and Table 3.

Table 2 shows the instructions that presented different number of calls between the *Iteration2* and *Iteration3*, whose versions present the lowest energy consumption (see Fig. 3b). A reduction in the number of *iload_1* instructions can be observed in the *Iteration3* results in Table 2, as well as an equivalent increase in the number of *iload* instructions. These instructions require one more access to memory than the *iload_<n>* instructions (like *iload_0*, …, *iload_3*). This extra memory access is used to load the index of which variable in the pool will be popped on the JVM operand stack, unlike the simpler *iload_<n>* instructions that has this address implicitly defined. Moreover, these results show reductions in the number of *invokevirtual* and *ireturn* instructions that represents an invocation of a class instance method in Java and the return instruction, respectively. These reductions were expected, since the inline removed method calls. Besides, a great reduction in the number of the *aload_0* instructions can be observed in Table 2.

Table 2. Main instructions used for the Iteration 2 and 3

Instruction	Iteration 2	Iteration 3	Difference
iload	2256402932	2284795276	28392344
iload_1	117059899	88667555	-28392344
aload_0	1058245488	916283768	-141961720
istore	372594809	400987153	28392344
istore_1	28458885	66541	-28392344
ireturn	41305600	12913256	-28392344
invokevirtual	37981918	9589574	-28392344

To observe the increasing on energy provoked by inline, the difference between the third and fourth iterations is analyzed. Table 3 shows the number of used instructions for each code version (Iteration 3 and Iteration 4) and the difference between these numbers. The results show a great increase in the number of *iload* instructions and also a decrease in *iload_2* instructions, and yet an increase in the number of *getfield* instructions. Moreover, Table 3 shows the effect of the inline method, where the number of method calls is decreased, provoking a reduction in the number of *invokevirtual* and *return* instructions. In addition to that, the inline affected

the number of method´s local variables causing a changing of position in the variable pool. This changing can be noted in the variation of *istore* and *istore_1* instructions. With the method scope increasing the variable that occupied the second position in the variable pool went to a new position after the fourth position, generating the necessity of an *istore* instruction. The increase of *iload* is a consequence of the inline method, which inserted more local variables to the method and the instructions *iload_0, iload_1, iload_2* and *iload_3* cannot be used.

Table 3. Main instructions used for the Iteration 3 and 4

Instruction	Iteration 3	Iteration 4	Difference
iload	2284795276	2383364176	98568900
iload_2	289277504	190708604	-98568900
aload_0	916283768	939206768	22923000
aload_1	203437622	178986422	-24451200
istore	400987153	401751253	764100
istore_2	16634671	15870571	-764100
return	6336716	5572616	-764100
getfield	1169270767	1241096167	71825400
invokevirtual	9589574	8825474	-764100

The experiments show that for more complex application with more complex methods, as the MPEG decoder (the most complex application from Table 1), the inline method cannot achieve the expected improvements for energy or performance. Loss in energetic efficiency occurs when after inlining, the new method body/scope increase too much then simpler instructions have to be changed by more complex instructions. Moreover, indiscriminate inlining generates an increasing in the number of instructions related with Object Oriented Programming (like *getfield*).

5. Conclusions and future work

This paper studies the impact on the embedded system performance and energy consumption when the method inline refactoring is applied to Java codes. The Inline Method was expected to increase performance and decrease energy consumption since it would reduce the number of cycles to create a frame in the Java Virtual Machine for every method call. The expected results were achieved for the Address Book and Sokoban applications, but not for the MPEG decoder, which is the most complex analyzed application. Moreover, the results have shown that when applying inline in a method, the complexity of the method and whole application should be taken in account. It is because the reduction of a method call does not result in a gain when the method has many variables to be addressed in the variable pool.

As future work, we plan to explore other refactoring methods and case studies.

References

[1] Graaf, B.; Lormans, M.; Toetenel, H. “Embedded Software Engineering: the State of the Practice”. IEEE Software, v. 20, n. 6, p. 61- 69, Nov. – Dec. 2003.

[2] Saxe, E. “Power Efficient Software”. Communication of the ACM. v. 53, n. 02, Feb. 2010.

[3] Chatzigeorgiou, A. and Stephanides, G. “Evaluating Performance and Power of Object-Oriented vs. Procedural Programming in Embedded Processors”. Proc. of International Conference on Reliable Software Technologies, Ada-Europe 2002, Vienna, Austria, June 17-21, 2002.

[4] Fowler, M. “Refactoring: Improving the Design of Existing Code”, Addison Wesley, 14th printing, 2004.

[5] Sun Microsystems, Inc. “The JavaTM Virtual Machine Specification”, http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html, March, 1999.

[6] Mattos, J.C.B., Carro, L. “Object and Method Exploration for Embedded Systems Applications”. Proc. of Symposium on Integrated Circuits and Systems Design, SBCCI 2007, Rio de Janeiro, Brazil, 2007.

[7] Ito, S. A., Carro, L., & Jacobi, R. P. “Making java work for microcontroller applications”. IEEE Design & Test of Computers, v.18, n. 5, p.100-110, 2001.

[8] SPECjvm2008 (Java Virtual Machine Benchmark), http://www.spec.org/jvm2008.

[9] Metrics Eclipse Plug-in, http://metrics.sourceforge.net/.