Analysis of the impact of refactorings on the performance of embedded systems

Heitor B. dos Reis F.[1] Ulisses B. Corrêa [1], Lucio Mauro Duarte[1], Antônio C. S. Beck[1]
Instituto de Informática
[1]Universidade Federal do Rio Grande do Sul(UFRGS)
Porto Alegre, Brasil
{hbrfilho@inf.ufrgs.br},{caco@inf.ufrgs.br},{ ubcorrea@inf.ufrgs.br },{lmduarte@inf.ufrgs.br}

***Abstract*— In recent years, the large growth of embedded platforms available in the market brought a great challenge to software developers, since this kind of system offers limited hardware resources, which are lower than desktop computers. Therefore, these must be optimally used by programmers in order to meet expectations of end-users, such as performance, usability and battery life.**

In this context, this work aims to study the impact of code refactoring on the embedded devices' performance. This study is done by source code refactoring, aiming to improve readability and performance without altering the original functionality of the requirement. Three basic refactorings were evaluated and the Mult2Sim processor simulator was used.

***Keywords*— physical metrics; Performance microprocessors. .**

Introduction

The continuous growth of the use of embedded systems brings a great challenge to software developers, considering the limited number of hardware resources these systems offer. For this reason, these resources must be optimally used by programmers in order to meet end-users' expectations such as performance, usability, and battery life. Some examples of complex embedded system are: Smartphones, Tablets, smartwatches, smartTVs, etc. These devices have considerable processing capacity, with resources dedicated to the tasks they were designed for [14]. Some of the limitations that are observed in these devices are directly related to memory usage and battery consumption.

The evolution of software applications, as well as increasing its complexity, has led software engineers to enhance the technical analysis, software design and coding [1]. Thus, programming paradigms have been developed over time to improve productivity, making the programming activity more natural to the developers. However, the most popular programming paradigms are still structured and object-oriented programming.

There are some professional guidelines suggesting not employing the object-oriented paradigm due to its negative effect on the software performance (overhead) [3]. This can be related to the perception that, for example, applications developed with object-oriented languages have a larger memory footprint. However, this is only an assumption that has yet to be proven.

Thus, this work aims to present an initial study about how the programming best practices influence performance. More specifically, the technique of refactoring will be studied [3] as well as how this practice may influence the metrics such as: memory footprint, number of CPU cycles, energy consumption, power dissipation, etc. The refactoring was done using the features available in Eclipse IDE, and applied to source code written in C++. The physical metrics were obtained from Mult2sim [4], which emulates X86 processors.

This work is organized as follows: Section II presents some basic ideas about source code refactoring and a small refactoring catalog; Section III contains a description of our case study; in Section IV, we discuss our experimental results; finally, Section V presents our conclusions and some ideas for future work.

II. Refactoring

During the process of software development many failures are inserted into source code, and bring to light many types of errors, which can be minimized with a proper planning and execution of software tests. However, the quality of a particular application is not determined by the non-occurrence of errors only, but also how the application was built [5].

Traditional methodologies cope with the software developing process interactively and incrementally, most of times using a robust formalism throughout all the development stages, when refactorings are constantly used [1] [3]. According to [3] refactoring is a "transformation that preserves the behavior". However, many times this process may be somewhat impaired because of excessive rules that are involved in the whole process [6] or the need for faster development (shorter time-to-market). With the advent of agile methodologies, several new software development practices have been successfully applied in many different types of projects, in which refactoring has an important role.

Thus, the refactoring has as primary goal to increase the readability of the source code in order to facilitate the maintenance task in the future. However, this also allows the improvement of other features, such as of the software design, the process of searching for faults or even performance. The refactoring process may also involve tasks such as: removing duplicate code, simplification of conditional logic and so on.

The concept of refactoring can be applied in various stages of development. In the design phase, for instance, it is possible to apply refactoring patterns [7]. However, a given type of refactoring may have a greater impact than other depending on which point of the development stage it was applied. Examples of code refactoring [3] include Method Extracting, Method Inlining, Conditional Decomposing, Hierarchy Collapsing, Changing Unidirectional Association to Bidirectional, Changing References to Values, Pull Up Method, Renaming, etc. There are many other types of refactoring presented in Fowler's refactoring catalog [3], and many more types of refactorings have been introduced by other authors.

In the following we enlist and explain some refactoring techniques.

A. *Method Extracting*

One of the main pitfalls that may occur during programming is code duplication. This kind of refactoring replaces duplicated code fragments by method calls. This allows better reuse of code, thereby improving the maintainability of the program. [3].

B. *Method Inlining*

This one is the opposite of the aforementioned refactoring. In this case, the source code contained in a method replaces the method call. This makes sense, for instance, if the method body is as clear as its name [3] or if the method is called one time only in the whole code.

C. *Renaming*

This refactoring consists of renaming identifiers in order to improve the understanding of the source code. This type of refactoring is usually quite simple and can be performed automatically by modern IDEs.

D. *Conditional Decomposing*

It is used when there are complex and hard to follow conditional statements, with many nested if-then-else conditionals. This kind of refactoring also helps making the code more readable [3].

E. *Hierarchy Collapsing*

Applied in cases when the superclass and the subclass do not have many differences (i.e., the specialization of the subclass is not significant). Therefore, it unifies both by grouping attributes and methods [3].

F. *Changing Unidirectional Association to Bidirectional*

When two classes are connected in one direction, and classes need to use resources contained in both of them, this refactoring is needed [6].

G. *Reference to Value Changing*

When there is a class with a reference to an object that is small, which does not change during execution and that is systematically used by the class, one can make this object a field of the same object.

H. *Pull up method*

The pull up refactoring method should be applied when there is a hierarchy of classes with duplicate methods with duplicated behavior. Even though two duplicate methods would correctly work, they would increase chances of mistakes in the future.

III. CASE STUDY

In this section we describe our case study and the methodology of ours experiments.

Initially the idea was to implement small benchmarks that allow assessing the performance before and after the implementation of changes in the code, but this approach showed to be ineffective. Thus to verify the influence of refactorings in the final product, we have chosen the Mpeg2Decoder benchmark, which is part of Mibench benchmark set [8].

This program has a large size and is part of a package to evaluate the performance of CPUs. The original program was written in C language and was converted to C++ using an analytical method for refactoring object-oriented code [9], which also allowed us to analyze the impact of this conversion.

A. *Multi2Sim*

The benchmarks were executed using the Multi2Sim [4] architectural simulator. Multi2Sim simulates the characteristics of many hardware structures of the Intel x86 Instruction Set Architecture (ISA) [10] [11], like number of pipeline stages, functional units and cache memories. Each component can be configured [9].

The **Figure 1** shows the simulation flow, from where we collect data from application execution, like number of CPU executed cycles, number of executed instructions, and values of instructions per cycle (IPC). In this figure we can see that a regular source file is compiled to generate an executable binary file that will be passed to five different processor models, simulated in Multi2Sim [4], to generate the physical metrics.

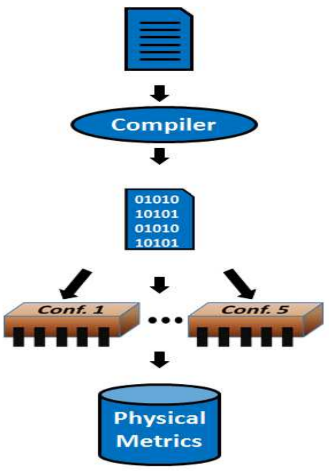

Figure 1- Steps to obtain physical metrics [12]**.**

B. *Methodology*

After choosing the benchmark (which will be described below), the source code was analyzed to find "bad smells" [3] [7]. All simulations were performed with the same CPU characteristics, but with different sizes for the L1 cache memory (32k, 16k, 8k, 4k, 2k). Initially, the original C benchmark was simulated. Then, the same code was converted to C++ and simulated again. The previous two simulations were done to evaluate the difference between using a structured language and an object-oriented language. Finally, the following refactorings were applied to the C++ code: renaming, Method Extracting and Method Inlining.

The first step to starting work was to create individual samples of source code refactorings to apply individually and verifying the individual influence of each one of them. We have also mixed refactorings, (i.e.: different refactorings were applied to the same code, such as renaming with method extracting and so on).

Eclipse Kepler IDE version 3.8 was used, with its support for native refactorings for C ++. Renaming was done automatically by Eclipse, while the other two were made in a hybrid manner, (i.e. using the tool with some manual intervention).

C. *The Benchmark Mpeg2Decoder*

The benchmark chosen to test the refactorings was the Mpeg2Decoder [7], which was originally written in C and was converted to the C++ [9]. In **Figure 2** one can see the basic class diagram of the application. Not all associations were designed, and the structs have also been omitted in order to increase the readability of the figure as well as focusing on interesting aspects of the application.

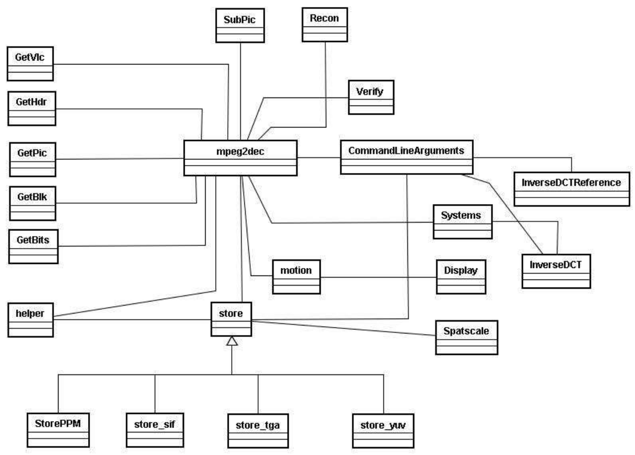

Figure 2 –Class diagram Mpeg2decoder

The **Figure 2** shows the set of the classes, in a total of 26 with 19664 lines of source code [8]. Several metrics for object orientation, using the Understand tool, were extracted from the respective software as shown in Table I [12]. The process of collecting the metrics of object orientation through the tool understand can be found in [12] for more details.

Table I - Metrics Object Orientation [12]

Metrics	Value
CountLine	19644
CountClassCoupled	84
CountDeclClass	26
CountInput	1260
CountOutput	922
Cyclomatic	877
PercentLackOfCohesion	26.35
MaxInheritanceTree	1
CountClassBase	4
CountClassDerived	4

In Table I it can be seen through the metric MaxInheritanceTree that the program does not explore inheritance, prioritizing compositions, which a wise policy design [13]. It also has a good cohesion and an average cyclomatic complexity.

The respective program uses the class mpeg2dec that contains the main () function to start the task of decoding. It receives all necessary parameters from the command line by using the CommandLineArguments class. After that, the file in MPEG2 format is loaded, and the various parts of the image are decoded by GetHdr, GetVlc, GetPic, GetBlk and GetBits classes.

The storePPM, store_sif, store_tga and store_yuv classes are responsible for generating the output file according to the parameters that were passed via the command line. For our simulations, we used the YUV output that generates three files without a header for each component: the component of light is stored in files with the extension Y, and chromatography components are stored in files with the U and V.

IV. CODE REFACTORING OVERHEAD ANALISYS

In this section the results of the simulations carried out in our research will be analyzed. In Table II and table III, data from simulations in the original code in C language and in C++ language are presented, respectively.

The metrics presented in tables II and III are described below:

Cycles: total number of cycles needed for the execution of the program.

Memory used: total occupancy of memory for program execution.

IPC: number of instructions per cycle.

Number of instructions: the number of instructions that were executed.

Accesses L1: all accesses to the L1 cache memory.

Hits L1: total number of Hits in cache memory L1.

Misses L1: total number of Misses in cache memory L1.

Accesses L2: all accesses to the L2 cache memory.

Hits L2: total number of Hits in cache memory L2.

Misses L2: total number of Misses in cache memory L2.

Table II - Results of simulations of the source code in C language

	Cache Size				
	32k	16k	8k	4k	2k
Cicles	3034249982	6610641842	16051574634	16053978604	25117684403
Memory used	9318400	9318400	9318400	9318400	9318400
IPC	0,1752	0,08043	0,03313	0,03312	0.02117
Number of Instructions	531725054	531725054	531725054	531725054	531725054
Acessess L1	158087599	168019769	258729996	258775791	346435491
Hits L1	146476855	140563679	81289881	81259091	50479726
Misses L1	11610744	27456090	177440115	177516700	295955765
Acessess L2	38626415	74326847	370122932	370235493	538103103
Hits L2	27746199	41273808	236588873	236615962	321107951
Misses L2	10880216	33053039	133534059	133619531	216995152

Analyzing the results one can observe that there is a loss of 9% in the C ++ version in relation to the number of cycles for all simulations. In the aspect of memory footprint, there was an increase of 5% for the C + + language. The metric number of instructions increased 7% with the use of the C + + language.

The number of Misses in the cache access is quite significant with respect to the use of two languages, especially when the cache size decreases.

Table III - Results of simulations of the source code in C ++ language

	Cache Size				
	32k	16k	8k	4k	2k
Cicles	3310120173	6747575343	10306245313	18194940790	27753374586
Memory used	9854976	9854976	9854976	9854976	9854976
IPC	0,1725	0,08461	0,0554	0,03138	0,02057
Number of Instructions	570934486	570934486	570934486	570934486	570934486
Acessess L1	160835064	164605905	201007120	260854586	351214773
Hits L1	146365795	129874665	105737124	78999830	46221810
Misses L1	14469269	34731240	95269996	181854756	304992963
Acessess L2	42742567	91063812	205043483	386514867	564369567
Hits L2	30015192	55866615	134331638	244125197	333498872
Misses L2	12727375	35197197	70711845	142389670	230870695

The **Figure 3** shows the increased cache miss rate as the size of the cache decreases. Analyzing the graph it is observed that the refactoring conversion from C to C + + is not worth for embedded systems that have a minimal amount of cache memory available, since the number of cache misses in C language for this benchmark is lower.

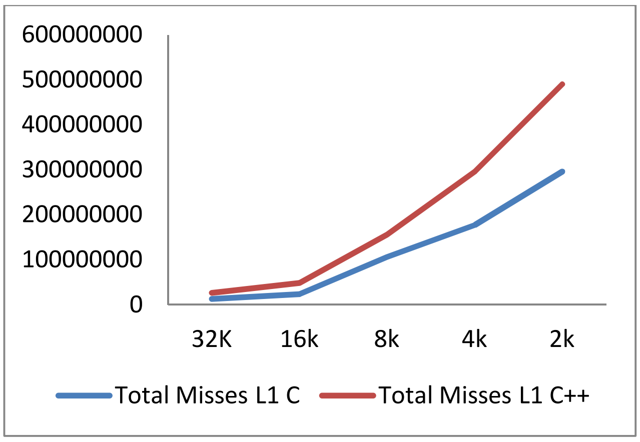

Figure 3 – Misses distribution function of the variation in the cache memory.

In the table IV, data are presented after the refactorings inline method, rename e Extract method were applied on the classes store_YUV, GetPic, GetBits e Mpeg2Dec. The results obtained were not significant, because the variation in data was very small. In comparison with the results of the conversion from C to C ++, one can observe (in table III) that there was a reduction of only 2.2% for the number of cycles with 32k, while this reduction was of less than 1% for a cache with 2k. There was no changes in the IPC, as well as when one considers the memory footprint. The change in the number of instructions is also negligible(less than 1%).

Table IV - Results of the simulations after applying the inline method refactoring, rename and extract method.

	Cache Size				
	32k	16k	8k	4k	2k
Cicles	3238213016	6744141109	10535869153	18359735140	27753348309
Memory used	9854976	9854976	9854976	9854976	9854976
IPC	0,1763	0,08466	0,05419	0,0311	0,02057
Number of Instructions	570939459	570939459	570939459	570939459	570939459
Acessess L1	159211048	165216733	221442148	265101618	353483137
Hits L1	146026091	136135181	106290764	79324863	46265809
Misses L1	13184957	29081552	115151384	185776755	307217328
Acessess L2	41823140	80650931	230609262	387585830	566024171
Hits L2	29395751	45311852	159420148	245184770	334576069
Misses L2	12427389	35339079	71189114	142401060	231448102

Regarding the effect on accesses to the cache memory, there was also no significant change, as can be seen in **Figure 4**.

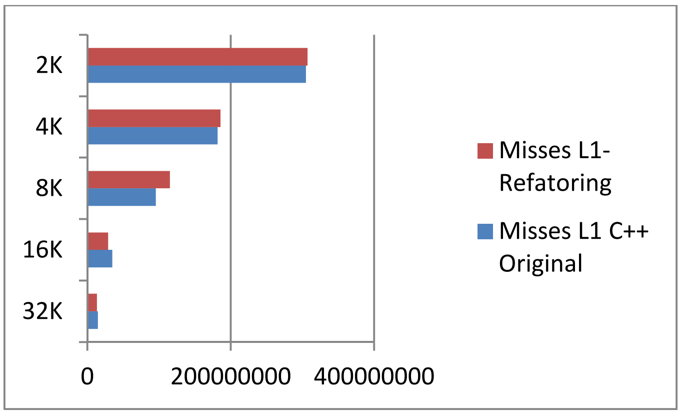

Figure 4 – Misses distribution to Original C++ and Refactoring program.

V. CONCLUSIONS AND FUTURE WORK

The conversion applied to the original code in C with its implementation of design patterns [9] had some impact, which is still acceptable if the machine does not have a small amount of cache memory.

Our experiments demonstrated that there was no variation in the data analyzed for the Rename refactoring. Therefore, it application should be encouraged, since it does not affect the performance but facilitates the understanding of the code. Moreover, it can be automatically applied using a refactoring tool such as the Eclipse IDE.

As future work, we intend to apply the same type of experiment to other benchmarks. They should preferably include inheritance and polymorphism so that we could also analyze other refactorings. We plan to evaluate the impact of refactorings on physical metrics considering other architectures and compilers, so that we can analyze whether the effect of refactorings in software performance is different for different configurations. This would give us a better understanding of the relation between code refactoring and physical metrics and, ultimately, lead us to determine in which situations the use of refactoring is recommended and which it is not. This information could help developers to weigh the benefits and disadvantages of applying refactorings depending, especially, on their non-functional requirements regarding performance and decide whether it is worth or not to improve readability and maintainability in exchange for a possible decrease of performance.

References

[1] R. S. Pressman, Software Engineering, United States of America: McGraw-hill, 2009.

[2] Google Company, "Performance Tips," 2013. [Online]. Available: http://developer.android.com/training/articles/perf-tips.html#PackageInner. [Accessed 28 08 2013].

[3] M. Fowler, Refactoring: Improving The Design of Existing Code, United States of America: Addison Wesley Professional, 2000.

[4] Mult2Sim, “Mult2Sim,” 30 06 2014. [Online]. Available: https://www.multi2sim.org/. [Acesso em 30 06 2014].

[5] S. Ian, Software Engineering, 9th edition, New York: Pearson Addison-Wesley, 2010.

[6] K. Beck e C. Andres, Extreme Programming Explained, Boston: Addison-Wesley, 2004.

[7] K. Joshua, Refactoring to Patterns, United States of America: Pearson Higher Education , 2004.

[8] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge e R. B. Brown, “MiBench: A free, commercially representative embedded benchmark suite,” 2001.

[9] S. Pissanetzky, “An Analytical Method for Refactoring Object-Oriented Code,” 28 08 2006.

[10] J. L. Henessy e D. A. Patterson, Computer Organization and Design - The Hardware/Software Interface, United States of America: The Morgan Kaufmann, 1998.

[11] D. F. Toledo e F. W. C. d. Oliveira, “Avaliação do Desempenho de uma Arquitetura heterogênea simulada alterando a latência da memória cache L2 da GPU,” *Enacomp,* 2013.

[12] U. B. Corrêa, A. C. S. F. Luís F.G. Millani e L. Carro, “Quality Impact on Software Performance,” *SBESC* , 2013.

[13] C. Larman, Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition), Massachusetts: Pearson Education, 2009.

[14] Beck A.C.S., Lisbôa C.A.L., Carro L., Adaptable Embedded Systems, New York, Springer, 2013.