Interrelations between Software Quality Metrics, Performance and Energy Consumption in Embedded Applications

Lazaros Papadopoulos
School of ECE, National Technical University of Athens, Greece
lpapadop@microlab.ntua.gr

Charalampos Marantos
School of ECE, National Technical University of Athens, Greece
hmarantos@microlab.ntua.gr

Georgios Digkas
Department of Mathematics and Computer Science, University of Groningen, The Netherlands
g.digkas@rug.nl

Apostolos Ampatzoglou
Department of Applied Informatics, University of Macedonia, Greece
apostolos.ampatzoglou@gmail.com

Alexander Chatzigeorgiou
Department of Applied Informatics, University of Macedonia, Greece
achat@uom.gr

Dimitrios Soudris
School of ECE, National Technical University of Athens, Greece
dsoudris@microlab.ntua.gr

ABSTRACT

Source code refactorings and transformations are extensively used by embedded system developers to improve the quality of applications, often supported by various open source and proprietary tools. They either aim at improving the design time quality, such as the maintainability and reusability of software artifacts, or the runtime quality such as performance and energy efficiency. However, an inherent trade-off between design- and run-time qualities is often present posing challenges to embedded software development. This work is a first step towards the investigation of the impact of transformations for improving the performance and the energy efficiency on software quality metrics and the impact of refactorings for increasing the design time quality on the execution time, the memory and the energy consumption. Based on a set of embedded applications from widely used benchmark suites and typical transformations and refactorings, we identify interrelations and trade-offs between the aforementioned metrics.

1 INTRODUCTION

A wide variety of technologies and markets that experience rapid growth, such as the augmented reality, medical electronics, autonomous driving and wearable devices are enabled by low power, usually heterogeneous, embedded systems. In the world of Internet of Things (IoT), embedded systems act as low power edge devices in IoT networks with hard constraints in terms of performance and energy consumption. The evolution of these technologies and the requirements for increased processing capabilities along with power efficiency impose very high requirements on the embedded systems design and in embedded software development.

From the embedded systems design perspective, various kinds of heterogeneous computing architectures provide increased performance at constrained energy consumption. From the embedded software point of view, a wide variety of methodologies, techniques and tools have been proposed to efficiently manage the resources of heterogeneous architectures [4]. At application level, typical data management transformations are extensively used to improve the memory hierarchy utilization and to increase performance and energy efficiency [5]. Other techniques, such as the identification of expensive system calls and the efficient memory heap utilization may also be employed to reduce execution time and energy consumption [11]. Developers often leverage profiling tools, such as *Valgrind* and *perf* to identify suitable optimization techniques.

At the same time, the rapid evolution of the embedded systems market, the emergence of new hardware architectures and the requirements for long lifetime expectancy of embedded applications increase the demand for highly maintainable software products [3]. Poor design time quality may impose significant overhead in maintenance activities, often termed as *Technical Debt* (TD) [10]. However, the effects of source code transformations and optimizations that software developers apply to improve the performance and the energy consumption of embedded applications may affect the maintainability of software products. In other words, employing such techniques to improve the *runtime quality* of embedded applications (i.e. performance, memory requirements and the energy consumption) may have positive or negative impact on the *design time quality* of applications, such as the maintainability, reusability and testability. Similarly, refactorings that improve code quality may affect runtime quality. For example, by employing polymorphism to improve code quality, performance improves as well [7].

In this work, we investigate relations between design time and runtime quality of embedded applications. Although the issues of software quality in industrial embedded software have been investigated in the past [3], to the best of our knowledge, this is the first study of the effects of the performance/energy consumption optimizations at source code level to the design time quality for embedded applications and vice versa. By leveraging a set of embedded applications from widely used benchmark suites and tools that provide software quality and performance/energy consumption indications, we apply typical source code transformations/refactorings for improving various quality metrics, such as the cognitive

SCOPES '18, May 28–30, 2018, Sankt Goar, Germany

ACM ISBN 978-1-4503-5780-7/18/05…$15.00
https://doi.org/10.1145/3207719.3207736

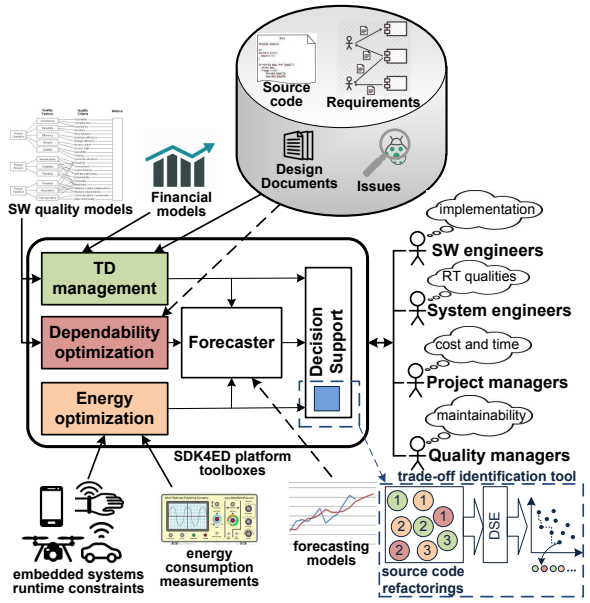


Figure 1: SDK4ED framework and trade-off identification.

complexity (for code quality) and the cache misses (for performance/energy). Thus, we investigate the interrelations between the design time quality and the runtime quality metrics and draw interesting conclusions.

2 TRANSFORMATIONS AND REFACTORINGS FOR DESIGN AND RUNTIME QUALITY IMPROVEMENT

The investigation of the interrelations between design and runtime quality at source code level will contribute to the design and the development of a tool that will enable the identification of trade-offs between their metrics. This tool will be a critical component of the SDK4ED framework, which will be developed in the context of the EU H2020 SDK4ED project [2] (Fig.1). In this Section we present a set of indicative source-to-source transformations for reducing execution time and energy consumption and a set of typical source code refactorings for improving the design time quality of applications.

Some of the typical source-to-source transformations for improving the performance and the energy efficiency of applications are listed in Table 1. The first transformation is the *removal of intermediate variables*, which are often used for temporary data storage and in cases such as in the example shown in Table 1, they can be eliminated. This transformation can potentially improve the execution time and the energy consumption. Also, if the variable is an array, the impact on memory efficiency may be significant. Finally, since it simplifies the source code, it may have positive impact on the comprehensibility. The second transformation can be used to avoid the *unnecessary reassignment of variables*. In the example of Table 1, array *arr* is accessed in each inner loop to assign the *arr[i]* value to *a*. The transformation eliminates the unnecessary memory accesses and it is expected to positively affect the execution time and the

Table 1: Indicative source-to-source transformations for improving performance and energy efficiency.

Before	After
Transformation 1: Intermediate variable removal	
`void foo(){ a = f(x); g(a); }`	`void foo(){ g(f(x)); }`
Improves performance/memory/energy and code quality	
Transformation 2: Avoid unnecessary variable reassignment	
`for (i=0;i<N;i++){ for (j=0;j<N;j++){ a = arr[i]; ... } }`	`for (i=0;i<N;i++){ a=arr[i]; for (j=0;j<N;j++){ ... } }`
Improves performance/energy	
Transformation 3: Loop interchange	
`for (i=0;i<N;i++){ for (j=0;j<N;j++){ ... } }`	`for (j=0;j<N;j++){ for (i=0;i<N;i++){ ... } }`
Improves performance/energy	
Transformation 4: Switch from dynamic to static allocation	
`a=(int*)malloc(SZ); if (a==NULL) { // error message. } ... free(a);`	`int a[ENTRIES];`
Improves performance/energy, code quality **May increase memory requirements.**	
Transformation 5: Switch from static to dynamic allocation (The opposite of Transformation 4) **May improve memory requirements.** **Increases execution time and energy consumption.**	

runtime consumption. Memory requirements and software quality are not affected. *Loop interchange* is a typical data reuse transformation that, when properly applied, reduces execution time and energy consumption by improving memory hierarchy utilization. The fourth transformation is the *switching from dynamic memory allocation to static*. Although this transformation may increase the memory requirements, performance and energy are expected to improve, due to the elimination of the overhead imposed by the dynamic memory allocators of embedded systems. Additionally, software metrics pertaining to comprehensibility, testability and maintainability are expected to improve, due to the fact the static memory allocation is simple and straightforward. Finally, switching

Table 2: Indicative refactorings for improving code quality.

Before	After
Refactoring 1: Extract Method	

`void foo() {`
`  ...`
`  // fibonacci:`
`  for(i=1;i<n;i++){`
`  next_term=t1+t2;`
`  t1=t2;`
`  t2=next_term;`
`  }`
`}`
	`void foo() {`
	`  ...`
	`  fibonacci(n)`
	`}`
	`void fibonacci(n){`
	`  for(i=1;i<n;i++){`
	`  next_term=t1+t2;`
	`  t1=t2;`
	`  t2=next_term;`
	`  }`
	`}`

Improves understandability, complexity, cohesion **Increases execution time**	
Refactoring 2: Consolidate Duplicate Conditional Fragments	

`if (a>0){`
`  ...`
`  foo();`
`}`
`else{`
`  ...`
`  foo();`
`}`
	`if (a>0){`
	`  ...`
	`}`
	`else{`
	`  ...`
	`}`
	`foo();`

Improves comprehensibility, maintainability, and code size	
Refactoring 3: Replace Conditional with Polymorphism	

`class Movie{`
`  ...`
`  double getCharge(){`
`  switch(priceCode){`
`  case REGULAR:`
`  // regular price`
`  case CHILDREN:`
`  // for children`
`  }`
`  }`
`}`
	`abstract class Movie{`
	`  abstract getCharge();`
	`}`
	`class RegularMovie`
	`  extends Movie{`
	`  double getCharge(){`
	`  // regular price`
	`  }`
	`class ChildrensMovie`
	`  extends Movie{`
	`  double getCharge(){`
	`  // for children`
	`  }`
	`}`

Improves maintainability memory and execution time	

from static to dynamic memory allocation (Transformation 5) has the exact opposite effects of Transformation 4.

Some of the most representative refactorings [8] aiming at improving software design time qualities, such as maintainability and reusability, are listed in Table 2. *Extract method* refactoring targets the Long method code smell [8], that is, methods which are long, complex and non-cohesive. Such methods violate the *Single Responsibility Principle* according to which any module should take over a single responsibility so that it has only one reason to change. Extracting a cohesive set of statements to a separate method, renders the resulting methods less complex, smaller in size and more cohesive, facilitating their maintenance. However, this refactoring negatively impacts execution time as an additional method invocation is needed and also increases the total code size affecting memory footprint.

Code cloning is one of the most frequent and debated symptoms in software maintenance. Code clones are considered harmful because: a) duplicates of code generally increase maintenance costs and b) inconsistent changes to clones may lead to incorrect program behavior. In general, removing clones can have a mixed effect on design- and run-time qualities. However, the *consolidation of duplicate conditional fragments*, i.e. moving clones outside the branches of a conditional, has a positive effect on program comprehensibility, maintainability and code size without affecting execution time.

The third refactoring refers to the application of polymorphism in order to eliminate state-checking, which manifests itself as conditionals that select an execution path by comparing the value of a variable representing the state of an object. The corresponding refactoring consists in the introduction of an appropriate hierarchy of types along with the use of a polymorphic method call, drastically improving maintainability. However, code size is significantly increased impacting memory requirements while polymorphic method calls have a negative effect on execution time.

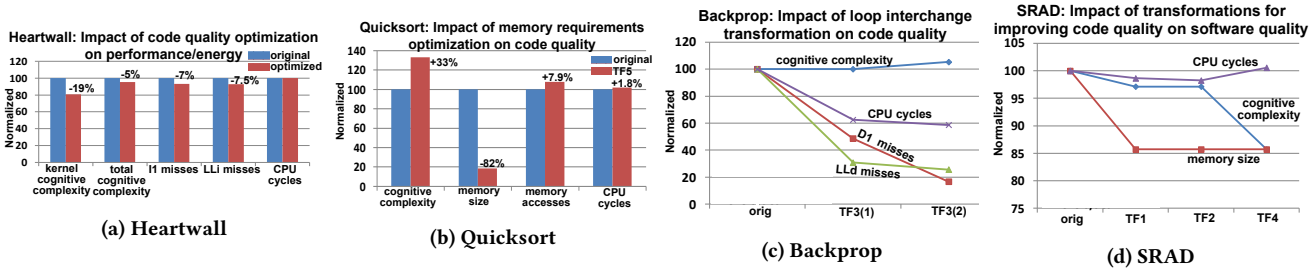
In the following section we will investigate the impact of the above transformations/refactorings for improving design time quality on runtime quality and vice versa.

3 INTERRELATIONS BETWEEN DESIGN AND RUNTIME QUALITIES

To investigate the interrelation between design and runtime quality metrics, we examined source code quality and performance/energy issues in the following embedded applications: i) *HeartWall* ii) *SRAD* iii) *Backprop* from *Rodinia* [6] and iv) *QSort* from the *MiBench* [9] embedded benchmark suite. For measuring design time quality, SonarQube [1] was selected and tools from the Valgrind suite were used for measuring the runtime quality of the applications under evaluation. The design time quality metric we selected is the cognitive complexity (as measured in SonarQube). Cognitive Complexity[1] measures the maintainability of the code. In other words, it shows how difficult is to understand and maintain a method. The difference with the Cyclomatic Complexity is that the latter measures the testability of the code. The runtime quality metrics we selected and which are indications for performance/energy and memory consumption, are the cache misses, the memory accesses, the memory footprint and the CPU cycles. Valgrind v.3.13 and gcc 5.4 with default optimization were used.

In Heartwall, which is an image processing application, we applied Refactorings 1 and 2, as proposed by SonarQube, to lower the cognitive complexity of a specific function by 19%, as shown in Fig.2a. The proposed refactorings were mainly the removal of

[1] https://blog.sonarsource.com/cognitive-complexity-because-testability-understandability



(a) Heartwall (b) Quicksort (c) Backprop (d) SRAD

Figure 2: Experimental results that indicate the interrelations between software quality and runtime quality metrics.

unused variables, the removal of duplicate code and the simplification of statements that improve source code understandability. By applying the refactorings, the cognitive complexity of the whole application improved by 5%, while the effects on the runtime quality metrics were minor. Although cache misses were slightly reduced, no significant impact on the CPU cycles was observed.

In Quicksort, a widely used sorting algorithm from the MiBench suite, we applied Transformation 5, so that the application input to be handled dynamically. Thus, as shown in Fig.2b, memory requirements reduced by 82%. However, the total number of memory accesses slightly increased (by 7.9%) due to the overhead imposed by the dynamic allocation, which slightly affected the CPU cycles. Dynamic allocation overhead affects cognitive complexity as well (e.g. checking the input size, detecting *malloc()* failure), which increased by 33%.

The impact of Transformation 3 on design and runtime quality is evaluated in the Backprop application (Fig.2c). We applied loop interchange in two different loops (TF3(1) and TF3(2)). TF3(1) significantly reduced cache misses and CPU cycles, up to 71% and 37.5% respectively. For applying the same transformation in another loop TF3(2) extra checks and utilization of extra variables were required. Therefore, although TF3(2) reduced cache misses by up to 86% and CPU cycles by 42%, cognitive complexity increased by 5%.

Three different transformations for improving performance, memory utilization and energy efficiency were applied in the SRAD application, depicted in Fig.2d. By removing intermediate variables, and more specifically an array used for temporary storage, memory requirements reduced by 15%. By optimizing loops and removing unnecessary memory accesses (TF2), CPU cycles reduced slightly. Finally, by switching from dynamic to static allocation, CPU cycles increased by 2%, however cognitive complexity significantly lowered by 15%. The latter, demonstrates improvement in design time quality with minor negative impact on performance.

Finally, based on the experimental results, we identify relations and trade-offs between the design and the runtime quality metrics.

Observation 1: The impact of Refactorings 1 and 2 for improving source code quality on performance and energy consumption is minor (Heartwall). However, as stated earlier, removing duplicate code (e.g. by developing new functions) may result in higher execution times when the number of function calls increases significantly. Nevertheless, no such overhead was observed in Heartwall.

Observation2: When applying Transformation 5 to switch from static to dynamic memory allocation, there can be a trade-off between memory requirements and cognitive complexity, as shown in QSort, since dynamic memory allocation results in more complicated source code.

Observation 3: When applying Transformation 3 to perform loop interchange and reduce the cache misses, the extra checks and variables that may be needed to retain the application functionality may increase cognitive complexity (Backprop). Therefore, trade-off between performance/energy consumption and cognitive complexity may be observed.

Observation 4: When applying Transformation 4, trade-off between performance/energy and cognitive complexity may be experienced. Indeed, as shown in SRAD, static memory allocation may improve performance/energy, however it significantly reduces cognitive complexity.

4 CONCLUSIONS

This work is a first step towards the investigation of the impact of transformations for improving runtime quality on design time quality metrics and vice versa. The identification of interrelations and trade-offs between them, as demonstrated in this work, underlines the need for further investigation that will finally lead to the design and development of tools that will assist embedded developers to perform optimizations considering both design and runtime quality aspects.

Acknowledgements This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 780572 SDK4ED (www.sdk4ed.eu).

REFERENCES

[1] 2008–2018. SonarQube website. https://www.sonarqube.org
[2] 2018. SDK4ED H2020 project. http://www.sdk4ed.eu
[3] A. Ampatzoglou and et al.. 2016. The Perception of Technical Debt in the Embedded Systems Domain: An Industrial Case Study. In *Proc. 2016 IEEE 8th International Workshop on Managing Technical Debt (MTD)*. pp. 9–16.
[4] Christos Baloukas and et al.. 2010. Mapping Embedded Applications on MPSoCs: The MNEMEE Approach. *Proc. ISVLSI'10* (2010), pp. 512–517.
[5] F Catthoor and et al.. 2013. *Data access and storage management for embedded programmable processors.* Springer Science & Business Media.
[6] S. Che and et al.. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proc. IISWC'09.* pp. 44–54.
[7] Serge Demeyer. 2003. Maintainability versus Performance: What's the Effect of Introducing Polymorphism?. In *Proc. ICSE'2003.*
[8] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Boston, MA, USA.
[9] M. R. Guthaus and et al.. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. IISWC'01.* pp. 3–14.
[10] P. Kruchten, R. L. Nord, and I. Ozkaya. 2012. Technical Debt: From Metaphor to Theory and Practice. *IEEE Software* 29, 6 (2012), pp. 18–21.
[11] S. Xydis, A. Bartzas, I. Anagnostopoulos, D. Soudris, and K. Pekmestzi. 2010. Custom multi-threaded Dynamic Memory Management for Multiprocessor System-on-Chip platforms. In *Proc. ICSAMOS'10.* pp. 102–109.