The process of and the lessons learned from performance tuning of a product family software architecture for mobile phones

Christian Del Rosso
Nokia Research Center
P.O Box 407
FIN-00045 NOKIA GROUP (Finland)
christian.del-rosso@nokia.com

Abstract

Performance is an important non-functional quality attribute of a software system but not always is considered when a software is designed. Furthermore, software evolves and changes can negatively affect the performance. New requirements could introduce performance problems and the need for a different architecture design. Even if the architecture has been designed to be easy to extend and flexible enough to be modified to perform its function, a software component designed to be too general and flexible can slower the execution of the application. Performance tuning is a way to assess the characteristics of an existing software and highlight design flaws or inefficiencies. Periodical performance tuning inspections and architecture assessments can help to discover potential bottlenecks before it is too late especially when changes and requirements are added to the architecture design. In this paper a performance tuning experience of one Nokia product family architecture will be described. Assessing a product family architecture means also taking into account the performance of the entire line of products and optimizations must include or at least not penalize its members.

1. Introduction

The increase in the speed of time-to-market and the necessity of launching new products has made the concept of the *software product family* important. A software product family is a set of software products having common assets and sharing architectural properties.The products can have similarities in the hardware and common requirements and can share part of the code. The software architecture for a product family must address the variabilities and commonalities of the entire set of products [10],[3],[4].

Designing the architecture and writing the code does not end the software life-cycle. The software evolves and needs to be maintained in order to support new requirements and new hardware. It can be said that the major part of the work is done after the coding phase as described by Robert Glass in his book *Facts and Fallacies of Software Engineering* [8].

Software architecture assessments are performed to evaluate how the system fulfills quality attributes. Modifiability, evolvability, security, performance are called non-functional quality attributes and represent important properties the software should have. Depending what quality attribute is evaluated, qualitative or quantitative methods can be applied. Regardless the methods are not mutually exclusive and can be complemented. Qualitative methods can be based on scenarios, check-lists, questionnaires, or can be experience based and are used, for example, to estimate modifiability and evolvability [1],[2],[14],[12]. Quantitative methods include RMA (Rate Monotonic Analysis) for the estimation of worst-case response time, queuing models for the prediction of performance and Markov chains for reasoning about reliability.

Of non-functional quality attributes, performance is the one that is not always addressed and considered properly. Especially in mobile phone software in which the real time constraints are important and where multimedia applications now seem essential, performance must be considered carefully.

The strategy currently adopted is usually to build the system, and if performance problems arise, to try to optimize it. However, when the software architecture has been designed and the code already written then the modifications required would be too expensive. The preferable strategy would be to consider the performance starting from the requirements phase and subsequently transforming the requirements in specific software design choices. Design patterns can aid in the design since they generalize and conceptualize the knowledge that experienced software designer have already experimented [7].

When the system is already built, we can still improve its performance and we call this performance tuning. Tuning the system includes dynamic analysis and dynamic architecture reconstruction [9], [11].

In this paper I will report an experience in tuning the performance of an existing product family architecture used in Nokia mobile phones. I will describe the method used, then the specific experience is reported. I will conclude the section on the experience with the results obtained. The chapter with the lessons learned and future work will end the article.

This case study has been performed within and funded by Nokia Mobile Phones. The author of this paper works for the corporate research center of Nokia. This is a separate organizational entity and does not have product development responsibilities.

2. Software performance engineering tuning

Discovering the bottlenecks in a large software system is not an easy task. The fact that the software is simply a part of a family of products introduces more complexity: the optimizations must consider the commonalities and variabilities of the architecture. Optimizing one software product must not affect other members of the family negatively.

We have decided to adopt a modified version of the performance engineering method [15], see figure 1.

A good understanding of the architecture examined is a prerequisite.

The first step in the process is the collection of architecture documentation. It is important to keep the scope of the evaluation focused without becoming too general or too vague. The scope must be discussed and clarified with the representative sponsoring the work. Having a scope too wide can disperse the efforts and deliver results that are too fuzzy. At the end of this step, a document must be created describing and delimiting the study.

A request from the customer to “find the bottleneck of the system” is somewhat too vague and too open. The first problem to be addressed is what software bottlenecks mean in the specific context. A set of variables must be listed and described. For example, in a system like mobile phone software, it can be important to highlight the CPU load, the memory consumption and the data throughput.

After agreeing on the variables to be analyzed, a set of use cases can be created [5]. The objective of this phase is to identify key performance use cases and scenarios. The need to focus on representative use cases means that only a limited number of use cases can be considered and the priority of their selection must be clearly stated. It is important to consider scenarios that are executed more frequently. Scenarios that are executed less frequently but that have a great impact on the system can also be considered. Scenarios whose performance is critical even though they are not executed frequently are also important.

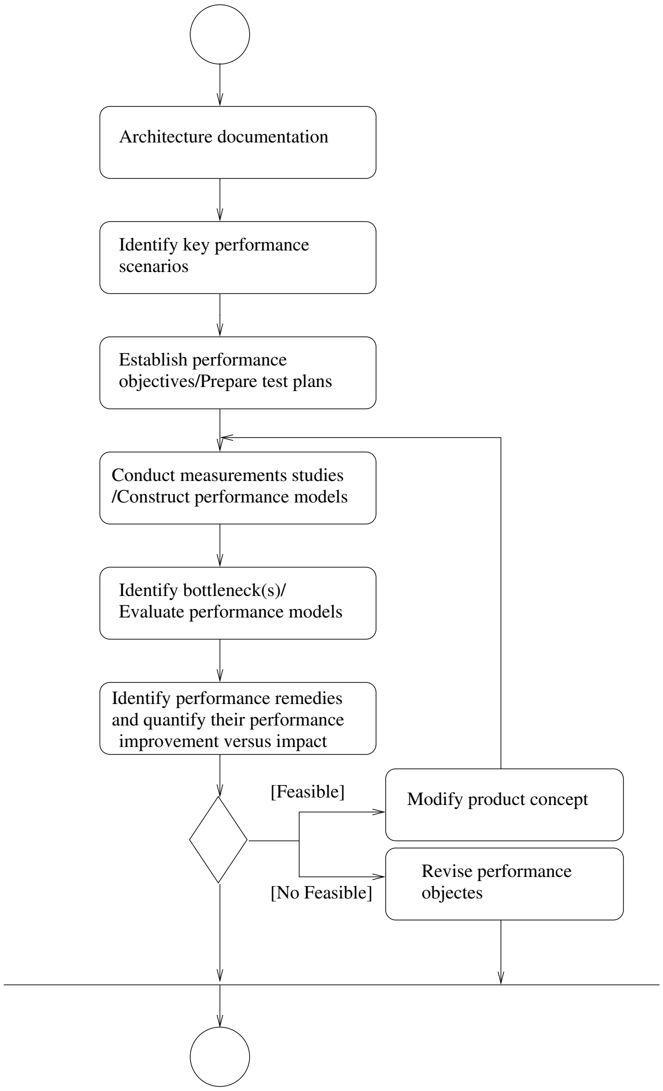
Architecture documentation
Identify key performance scenarios
Establish performance objectives/Prepare test plans
Conduct measurements studies /Construct performance models
Identify bottleneck(s)/ Evaluate performance models
Identify performance remedies and quantify their performance improvement versus impact
[Feasible]
[No Feasible]
Modify product concept
Revise performance objectes

Figure 1. Performance Tuning steps

Tuning implies that the system is already built therefore measurements of the run-time activities of the system can be taken. In this phase we conduct the measurements studies. A prerequisite for extracting the run-time activities of the system is that the code is instrumented (or can be instrumented), and the necessary tools to extract the activities (traces) exist. In our case instrumenting the software meant inserting when needed ”print” statements in the source code. Expertise and access to the tools and the equipment is necessary for this step.

With the logs of the activities, the analysis of the potential bottlenecks can start. The log files, which are events and their stamps, provide a snapshot of the system at run-time in the determined scenario. A dynamic reconstruction of the architecture is then possible. The reconstruction will be more or less complete depending on the variables traced. It must be considered that enabling the tracing of many variables influences the normal activity of the software, which can produce misleading outcomes. Instrumenting the software is an intrusive technique; enabling the tracing of many variables affects the measurements when a relevant part of CPU time is spent processing the instrumentation source code.

The analysis phase has been divided into two conceptual steps which are not necessarily sequential. First, the architecture is optimized at an higher level of abstraction i.e., searching for the design anti-patterns [13]. Subsequently specific parts of the source code are inspected and adjusted.

While patterns illustrate *best practices*, performance anti-patterns illustrate what not to do and how to fix problems. Anti-patterns document common mistakes made during software design and that experienced software architects have found useful to document. This solution to anti-patterns is called refactoring. Refactoring the design of the architecture is a transformation improving the quality and preserving the semantic [6],[15].

To apply design anti-patterns a good and updated architecture documentation is necessary. In most of the cases, the architecture documentation does not match exactly the *actual* architecture of the system. The software is continuously evolving and a large and complex system is developed by many people at different sites; therefore, it follows that having an accurate and complete documentation is difficult if not impossible. As part of the process of performance tuning, the software architecture documentation should be compared with the one partially reconstructed from the dynamic analysis.

Performance improvements made at the level of architecture design have a great impact on responsiveness and scalability. With a poor architecture or design, it is unlikely that any amount of clever coding will allow performance objectives to be achieved. However, even though having a good architecture and design is essential, it does not guarantee that performance is achieved. It is not possible to ignore implementation choices and coding details. For example, the CPU load is examined to see what process is being executed and what the process is performing until the code level is reached.

Once identified the performance remedies we need to quantify the performance improvements versus their impact. A dramatic design change must have valid reasons and the trade-off between the cost of implementing it and the performance gained must be carefully considered. Unexpected requirements could lead to an architecture not suitable for the task unless a considerable amount of money and resources are spent. To avoid such situations we strongly suggest including assessment targeted to the evolution of the life cycle of the product [14], [12].

Performance improvements can be made when the problems are clearly identified. The impact of every improvement must be quantified. It is important to consider that optimizations in one scenario can introduce inefficiencies in another. For product family architectures, an optimization of a specific product should not have negative consequences for other members of the software product family. In addition, quality attributes do not exist in isolation; sometimes, optimizing one quality attribute comes at the expense of another. Improving the performance typically comes at the expense of modifiability, increasing reliability frequently come at the expense of performance, and increasing security typically reduces system interoperability.

Once the changes and enhancements are in place it must be verified that these changes have effectively solved the problems and not introduced others. The process iterates measuring the system performance and comparing the results obtained.

3. Case study: mobile software product family architecture

The scope of performance tuning included one of Nokia's mobile terminal product family software. The introduction of new multimedia features has raised the need for a performance tuning project in which inefficiencies can be discovered and the current architecture optimized. In the system studied, the architectural elements communicate using message based communication services. Further details are part of industrial assets and therefore confidential.

3.1. Activities

The assessment team and the sponsor of the project came from two different organizational units within Nokia and acquiring the documentation of the system was easily accomplished. Furthermore, the author already had good knowledge of the software architecture.

The architecture documentation was used as input for the next two phases: delimiting the scope of the research and defining use cases.

The tuning addressed the optimization of CPU time usage and memory consumption. Since the system is message based we also focused on the behavior of the messages. In this system, messages can be of different types and the CPU time spent handling these different types was considered to be an important factor. Other variables considered were throughput and the amount of messages exchanged between the processes.

It was agreed to focus only on the features most frequently used by a mobile phone user since we wanted to make the interaction between the user interface and the end-user more effective. A document specifying the set of use cases was created. During a brainstorming meeting the scenarios were reduced and we reached a number of three. They were:

- phone start-up
- scrolling the phone book
- incoming call

Traces were collected using an R&D version of the software and two different products of the product family were examined. Based on the variables measured, only specific tracing-flags were enabled in the code. As a general rule, only the strictly needed traces must be enabled to avoid that the tracing activity excessively influences the measurements. Setting up the equipment and the measurement environment required support and a learning phase with the software and hardware tools was necessary. The logs recorded the activities and the events of the system in which we were interested and the next phase was to analyze and interpret them.

State of the art tools were used at the beginning of the analysis phase to visualize the CPU load and the memory consumption. However, the existing tools did not provide the information needed and did not perform the analysis we wanted. In our case, no-one had done the analysis specifically targeted at message performance before and we had to develop new tools or enhance existing ones. We added time stamps to an existing tool visualizing message sequence diagrams. Two more scripts were created. The first aimed for a finer level of granularity of the CPU load and the second was created to summarize the CPU time spent by each different type of message and their throughput. In addition, the tools showed the coupling of the processes in the system with the number of messages exchanged between every process.

A high number of messages exchanged between two processes highlighted a high coupling and joining them may be advised. A message between processes means the need for a context switch and regrouping the processes can significantly improve performance. On the other hand, joining many processes together could diminish the modularity of the system, diminish the flexibility of the architecture to evolve and have negative consequences on the development life-cycle considering that different modules can be developed in different sites.

Using the tools developed, a dynamic view of the architecture was reconstructed. At this point, we had an updated picture of the system which could be compared with the original architecture documentation. Looking at the dynamic view of the architecture, performance design anti-patterns can be found.

While trying to optimize the architecture design we also tried to deal with potential optimizations at a lower level of abstraction. The CPU load graphs and the memory consumption graphs showed the processes involved when performing a specific scenario. The peaks in the CPU load graphs clearly revealed the processes involved in the use cases. A high resource consuming activity was the first to be addressed and investigated. When a process was chosen as a *CPU hitter*, a deeper analysis was needed. It is not enough to be able to list the process involved, but is also important to understand the actual part of code involved in the processing time. This phase required a deeper knowledge of the system and a close contact with the developers. This analysis can reach and find the specific function call and access to the source code is required.

All the optimizations had to be considered against the different trade-offs listed in last three paragraphs of the previous section. The outcome of the analysis affected the architecture of the system and the performance tuning process followed an iterative path aimed to test enhancements against their impact on the architecture. At the time of this writing the first iteration has been performed and we plan to have more iterations. In the first iteration we had no major design changes, and in addition, a proposal was made for in-depth investigations targeting a specific area that the analysis had revealed as a potential bottleneck.

3.2. Results

The performance tuning analysis included a tracing phase and a subsequent partial architecture reconstruction of the dynamic view of the system.

The dynamic view and the message statistics allowed us to discover one anomalous activity that was consuming CPU cycles during the start-up phase. Through an in-depth analysis it was discovered to be part of legacy code no longer needed. This code was part of the previous product family architecture design but in the updated version the code was no longer needed; however, it was still there and was wasting CPU cycles. In this case, the system did not produce errors but the performance was affected.

We noticed frequent communication between some of the processes. In addition, an instance of the *GOD class* pattern was found [15]. We classify a GOD class in this specific context as a process performing all the work surrounded by simple processes. Processes communicating with a single common process deteriorate the performance by generating excessive message traffic. A better distribution of intelligence was desirable. Because of the limited amount of time and the exigency to study the issue in more detail follow-up study focusing on the optimization of the processes structure was recommended.

The optimizations were not only centered at the architecture design level. The analysis of the CPU time evolved from graphs showing the CPU load of the processes to the examination of specific parts of the code. Some information could not be obtained from the traces and it was necessary to instrument the code to extract the values of the specific variables. The granularity achieved permitted the identification of the source code involved; the analysis results will be part of the second iteration of the tuning phase and they are not included in this article.

4. Lessons Learned

4.1 Evaluation team not part of the development unit

The fact that the evaluation team was not part of the development unit permitted unbiased research. Developers and architects may have strong opinions which can lead them to assume that a certain design decision is optimal and to resist changes; vice versa, they could tend to be pessimistic and to see a solution as building the system from scratch.

The evaluation team can provide a neutral viewpoint since it does not have a stake in the development unit organization. In addition, new perspectives can be considered.

The reverse of the medal is that a good knowledge of the system is needed. The external evaluation team must spend time in learning the new system and its architecture.

4.2 The focus must be well defined

The evaluation must be well focused and agreed upon with the representatives commissioning the work. Too wide a scope does not allow the work to be clearly defined and makes it difficult to properly and accurately analyze the data collected. On the other hand too narrow a study will not provide enough useful information.

Experience in the architecture domain is important when creating the use cases and when the significant variables are considered. The activities consuming more resources need to be considered first. If the target is to optimize the memory usage, memory allocations and de-allocation must be considered, and in case that the tuning is concentrated in the CPU cycles, the processes loading the CPU are the ones to be considered; frequency and usage time are both important and the project objectives establish the priorities.

An optimization of the activities most frequently performed can significantly boost performance. Activities considered important or crucial for the functioning of the system can also have a high priority even though they are not executed often. when the system performs poorly in a high load situation, then research must be focused on the use case in which this high load condition happens.

Many different aspects can be observed and optimized and a document should specify the priorities of the study and with it, the reasons and the rationales.

4.3 Checking the dynamic view of the architecture

Assessing the performance of the system includes the dynamic analysis of the architecture.

A *snapshot* of every scenario was stored in a log text file created by extracting the events and the time stamps from the mobile phone. From the information collected it was possible to understand how the components interacted. The dynamic view of the architecture was reconstructed and the run-time activities inspected looking for hot spots and bottlenecks.

In a large and fast evolving system the architecture documentation does not perfectly match the actual architecture. The reconstruction can help to spot legacy code or behaviors not conformed to the specifications. Even if the performance tuning is not really focused on checking the specifications, sometimes, as we have experienced in our work, legacy code that utilizes resources unnecessarily can be found.

4.4 Performance improvements at different abstraction layers

We have followed a top-down approach. At a higher level of abstraction we considered architecture design optimizations.

Performance design anti-patterns can be found and the necessary transformations applied. Refactoring modifies the architecture design and trade-offs between the cost of implementing the change and the gain obtained must carefully be considered. In addition, in product family architecture particular care must be given to the impact of the changes on the entire line of the products.

After inspecting the system design the software is investigated and optimized at a lower level of abstraction. In our case, this also meant developing new tools. The necessity of understanding the system in more detail became clear. In a large software system it is usually impossible to know every detail and it is impossible to be an *expert-in-everything*. Architects have a high level of knowledge of the system design, while the developers have a high level of knowledge of the specific source-code module. The performance analyst must focus on the anomalies and look for the hot spots. With the list of potential problems, the experts can be contacted and the situation investigated further.

4.5 Performance improvements for the product family

As the products in a product family architecture share similar features and part of the source code, improvements done to the common architecture enhance the performance of the whole set of products. In our case, we used two different mobile phone models with a different set of features and hardware but belonging to the same product family. The major improvements were found in the product family architecture design and in the parts of the source code that was common to both. The improvements benefited the entire line of products.

5. Future work

We are continuing the performance analysis of the product family architecture described in this paper. New traces and enhancement of tools will allow a higher level of granularity and will permit a better understanding of the software architecture.

In addition, our future work will concentrate on applying this method to different domains and comparing different experiences. We advocate the continuation of research in the field, as well as the publication of more experience reports.

References

[1] G. Abowd, L. Bass, L. Clements, R. Kazman, L. Northrop, and A. Zaremski. Recommended best industrial practice for software architecture evaluation. *Technical Report CMU/SEI-96-TR-025*, http://www.sei.cmu.edu, January 1997.

[2] G. Booch. Conducting a software architecture assessment. *Rational White Paper*, http://www.rational.com/products/whitepapers/391.jsp.

[3] J. Bosch. *Design and Use of Sofware Architectures*. Addison Wesley, 2000.

[4] P. Clements and L. Northrop. *Software Product Lines*. Addison Wesley, 2002.

[5] A. Cockburn. *Writing Effective Use Cases*. Addison Wesley, 2000.

[6] M. Fowler. *Refactoring, improving the Design of the existing code*. Addison Wesley, 1999.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, 1995.

[8] R. L. Glass. *Facts and Fallacies of Software Engineering*. Addison Wesley, 2003.

[9] IEEE 2000. Recommended practice for architectural description of software-intensive systems. *IEEE Standard No. 1471-2000*, http://shop.ieee.org/store/.

[10] M. Jazayeri, F. Van Der Linden, and A. Ran. *Software Architecture for Product Families*. Addison Wesley, 2000.

[11] P. B. Kruchten. The 4+1 view model of software architecture. *IEEE Software*, Vol 12(issue 6), November 1995.

[12] A. Maccari. Experiences in assessing product family architecture for evolution. *Proceedings of the 23rd International conference on Software Engineering (ICSE)*, May 2002.

[13] I. Pyarali, C. O'Ryan, D. C. Schmidt, N. Wang, V. Kachroo, and A. Gokhale. Using principle patterns to optimize realtime orbs. *Concurrency, IEEE (see also IEEE Parallel & Distributed Technology)*, 8(1):16–25, Jan-March 2000.

[14] C. Riva and C. Del Rosso. Experiences with software product family evolution. *Proceedings of the sixth International workshop on principles of software evolution*, (SE-5 2), 2003.

[15] C. U. Smith and L. G. Williams. *Performance Solutions*. Addison Wesley, 1995.