Experiences of Performance Tuning Software Product Family Architectures Using a Scenario-Driven Approach

Christian Del Rosso
Nokia Research Center
Itämerenkatu 11-13, 00180
Helsinki, Finland
christian.del-rosso@nokia.com

Abstract

Performance is an important non functional quality attribute of a software system. The ability to deliver the expected performance objectives comes from a careful design and attention to detail. Unfortunately, performance is not always considered at the beginning. However, once built, software performance can still be improved by evaluating and tuning the software architecture. When analyzing the performance of a software product family, an understanding of its architectural properties is needed. A software product family architecture's strength is based on common assets, platforms and source code shared by its family members. Software product family design allows improved time-to-market, software quality and software reuse. At the same time, variability is the factor to instantiate different products and the handling of the variation points must be carefully managed. In this paper I present a scenario-driven approach for analyzing the performance of software product family architectures. The process of performance tuning has been applied to a Nokia software product family architecture and two case studies are presented. The evaluation process and the tradeoffs of evaluating software product family architectures are discussed.

Keywords: software product family, software performance, dynamic memory management, embedded real-time systems

1. INTRODUCTION

A software product family is a set of software-intensive systems having common assets and sharing architectural properties [3, 11, 5]. In a market where the demand for new and innovative products is increasing and where the complexity of the software never decreases, having a product family architecture is a competitive advantage. The concept of a software product family, or software product line, is derived from the hardware industry where new products were created by assembling and varying the existing products of the same line e.g a car manufacturing company can have lines like economic cars, luxurious cars. Products in the same family share the same reference architecture and part of the source code. Software reuse and a common reference architecture provide several advantages such as improved software quality and fast derivation of new products leading to improvements in time-to-market. At the same time, variation points are specified to differentiate the products. The managed set of variation points guarantees controlled diversification of the products of the same family. A variation point is a location at which change can occur in the software product line artifact [11]. Examples of variability mechanisms include inheritance (if the component is a class in an object oriented language) and configuration (by setting parameters at run-time or compile time) [3].

The characteristics of software product family architectures make a different approach necessary when performance analysis of the products is done. Tuning and improving the performance of a software product family implies enhancing the performance of the complete set of products or, at least, the products should not be penalized by improvements made to a specific model.

When dealing with a software product family, the analysis should consider specific issues of commonalities and variabilities in a software architecture for product family. Optimizations should not penalize products of a software family and should consider the implications of changes to the core architecture.

The main contribution of this paper is to present and discuss a scenario-driven approach to performance tuning a software product family architecture.

Key performance scenarios are selected and represent the starting point for the analysis. The scenario selection step is revised considering the characteristics of software product family architectures. The key performance scenarios will describe the most challenging and interesting scenarios that stress the system. Some of the scenarios will depend on specific functionalities implemented only in a subset of products, others will include functionalities common to all the products in the family. The performance analysts must be aware of the issue and must consider the implications for the analysis scope. Subsequently, in the tuning process, performance models or simulations are built and performance optimizations and improvements are proposed according to the results of the evaluation. However, the analysis phase takes into account the software product family and a further step is introduced to analyze the impact on and the tradeoffs against the software product family architecture.

I have validated the approach in the industrial realm using a Nokia software product family architecture. Two case studies are presented and analyzed using the proposed method. The remainder of this paper is organized as follows. In section 2 I present the method. In section 4 and 5, two case studies are presented; in the first case study I used the method to tune and improve the performance of the core features of the product family. In the second case, I used the method to analyze and tune the dynamic memory management of the software product family. I discuss the benefits and the tradeoffs of our approach in section 6, especially focusing on the use of scenario-based approach when tuning software product family architectures.

2. SOFTWARE PERFORMANCE TUNING FOR PRODUCT FAMILY

I use a scenario-driven approach to assess and tune the performance of a software product family. Performance tuning is done by first selecting key performance scenarios, then by developing performance models and simulation models, and subsequently, the analysis and the evaluation steps follow.

A scenario describes the actions performed by combining one or more features. A feature is defined here as a user visible functional requirement. Key performance scenarios are defined as the scenarios that influence the system the most in terms of performance speed and memory usage. The features most frequently used by users provide a good list for optimization and would benefit from improvements. At the same time, demanding features in terms of performance or memory usage must be included in the study. When analyzing performance of real-time systems, features with strict real-time constraints are important and must be considered.

The performance analyst uses the list of user-visible features as input; however, the business importance of features can play a role. Software architects and programmers and eventually users with a stake in the system analyzed are involved in the process. The scenario selection process is iterative and brainstorming sessions can be considered to refine the list. The number of scenarios must be limited to avoid too wide a scope; however, too narrow a scope would limit performance analysis. The number of scenarios to be defined also depends on the time the analysis has to be completed. Experience of the performance analyst also plays an important role in the process.

When tuning a software product family architecture, the scenario selection process must be revised and the whole process must be adapted to suit the particularities involved. The performance tuning in this case has a wider scope, a family of products. Analysis can focus on product instantiation and its particular features or, on core features of the product family architecture and therefore part of all product instantiations.
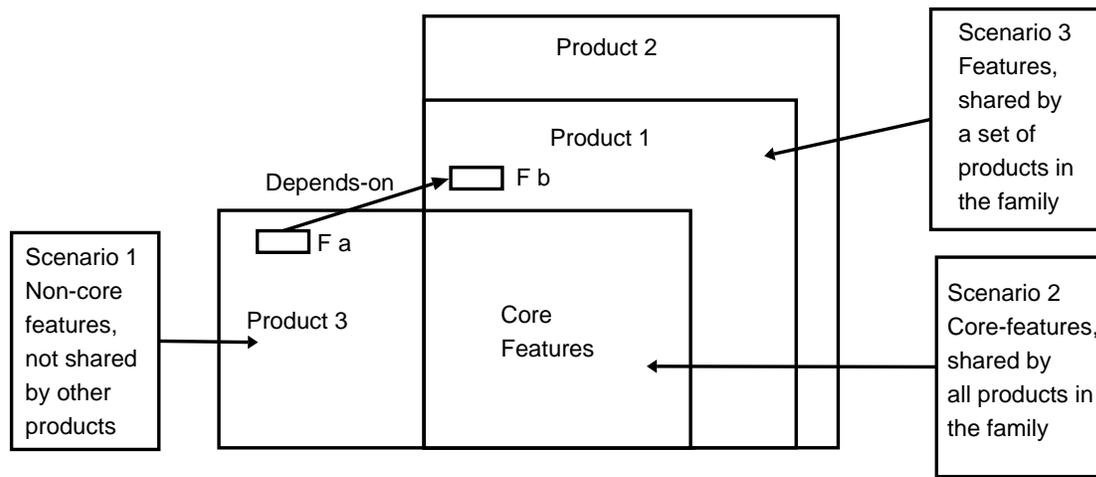
FIGURE 1: Scenario selection and features

Figure 1 represents how scenarios are mapped to specific products in the family. In figure 1, the blocks represent the products of the software product family and each product contains a set of features. Scenario 1 in the figure targets features that are implemented only in certain products of the family, the so-called optional features. An optional feature, a feature that is not part of the core features, can have particular requirements in terms of performance or resources. Multimedia features are examples of hungry resource consumers and they are becoming more and more common in handsets. In these cases the performance study can analyze the most resource-demanding features and optimize them by including these features in the key scenario set. On the other hand, the selection of core features for scenario selection assures an analysis focused on the entire range of products of the software product family as shown in scenario 2. In addition, depending on the analysis' scope, scenarios may include features shared by a subset of products in the family, and scenario 3 represents the concept in figure 1. In all these cases, the fact that performance tuning is done for a software product family is quite relevant and the whole tuning process must consider it.

After the key performance scenario definition phase, performance models and simulation techniques are used to study the performance of the system. One or more members of the product family are used as the representative set for the study, and modelling and simulations are based on their performance.

Evaluation of the simulations and performance models provides improvements using the key performance scenarios as reference and the simulation evaluation based on the specified products. However, a further step in the evaluation process has to be added to consider the software product family characteristics. Improvements and solutions found in the analysis must be analyzed versus the impact on the software product family architecture and its different members. The fact that the analysis focuses on features included only in certain products does not exclude other members from the impact analysis. Feature and architectural dependencies must be analyzed and well understood. Different features implemented in different products may have dependencies. Figure 1 shows how feature Fa instantiated only in product 3 depends on feature Fb instantiated in product 1. A feature dependency implies a dependency at architectural and at source code level. I will present the discussion of the impact analysis and the implications for the software product family in the discussion section of this paper in section 6.

3. WHY THE APPROACH WORKS

Designing and evaluating software architectures is a complex task. No two software architects will produce the same software design given the same set of requirements for the problem. Software design and evaluation is a human activity and therefore requires creativity. Great software is created by great programmers. Software patterns and anti-patterns have been catalogued to

summarize and to provide solutions to common design problems [10, 8, 4, 20]. However, software creation requires more than just applying good practices.

The performance assessment process presented is iterative and improvements to the outcome can be made in every single step of the process. First, the objectives and the focus of the analysis can be redefined and refined. Second, identification of key performance scenarios can be enhanced by including different stakeholders and enlarging or restricting the focus of the analysis. Third, additional performance models and simulations can be performed. Fourth, different architecture solutions may be investigated. The impact analysis will then consider the performance, implications and tradeoffs of the alternative solutions. And fifth, the end result of the proposed solutions must produce quantifiable evidence of the benefits of the assessment.

Evaluating software architectures is a complex task. Yet, software product family architectures add even more complexity to the picture since the evaluation includes a set of products. However, the scenario driven assessment approach presented provides an elegant, simple and effective way to improve the performance of such software systems. The method has been applied successfully to evaluate software product family architectures with approximately 2 million lines of source code in the industrial realm.

4. CASE STUDY 1, CPU LOAD

The scope of the first case study was the performance tuning of one of Nokia's mobile terminal product family software. The approach followed the process described in section 2.
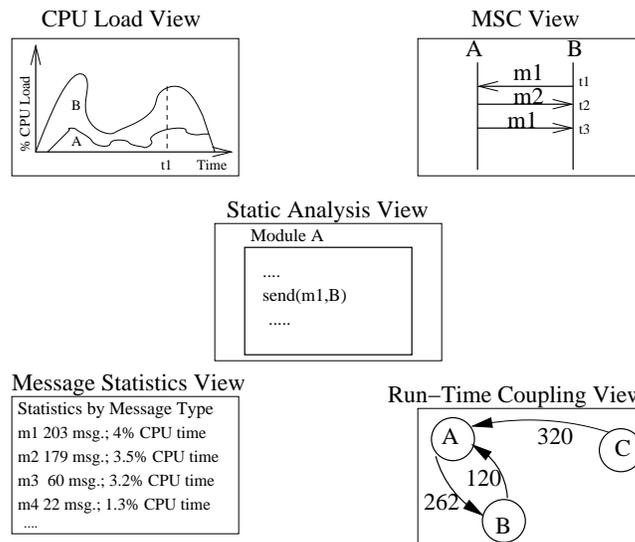
Key performance scenarios were defined after analyzing the handsets features. The selection process included brainstorming sessions with the stakeholders: architects, programmers and managers. During the selection phase process, a decision was made to focus on the optimizations for the reference architecture, and only core features were included in the scenarios. Examples of some of the scenarios analyzed were:

- Phone Start-Up
- Scrolling the Phone Book
- Incoming Call

Two handsets were selected as representatives for the study. The devices were from the same software product family but had different sets of features. One handset had a more complete and advanced features set, while the other had basic feature set. In addition, the handset with advanced features contained all the features included in the basic handset.

Performance tuning steps included a tracing phase using trace instrumentation. The data were extracted while running the selected scenarios in the handsets and text log files recorded scheduling information of the process and all information necessary for the analysis. After the tracing phase, the dynamic view of the architecture was reconstructed using a framework developed during the experiment aimed at the reconstruction of different architectural views of the run-time system.

The mobile terminal analyzed used a message passing mechanism as a communication paradigm. I used five different architectural views to optimize and analyze the performance: CPU load view, message statics view, run-time coupling view, message sequence chart view, and static analysis view, see figure 2. The CPU load view presented the CPU usage of the various processes. The message statistics view included the percentage of CPU time spent on handling the different message types during the scenarios. The run-time coupling view represented the run-time dependencies of the processes in the scenarios analyzed, and the weights in the arcs connecting the processes expressed, in our case, the number of messages exchanged. The message sequence charts view showed message sequence chart digrams with time stamps. The static analysis view was used to analyze the architectural dependencies of the system extracted from the source code, [17], and in this view were included component dependencies and function call dependencies.

CPU Load View

MSC View

Static Analysis View

Module A

....
send(m1,B)
.....

Message Statistics View

Statistics by Message Type
m1 203 msg.; 4% CPU time
m2 179 msg.; 3.5% CPU time
m3 60 msg.; 3.2% CPU time
m4 22 msg.; 1.3% CPU time
....

Run–Time Coupling View

FIGURE 2: Architectural Views Reconstructed

In this paper I focus on the discussion on the scenario-based approach for product family architectures; for a comprehensive discussion of the views see our paper [7].

The analysis of the dynamic view revealed architectural consistency problems that also caused performance overhead to the system. The main findings were the discovery of a break of an architectural rule during system startup and the discovery of the God class antipattern [20] in the product family architecture. The dynamic views and the analysis highlighted an anomalous activity that was consuming CPU cycles during the start-up phase. An in-depth analysis revealed that an architectural rule had been broken and that legacy code was consuming CPU cycles. The legacy code was part of the previous architecture design but it was not removed from the software source code. The fault did not create any visible failures such as system crashes, however, software performance was affected. In addition, I discovered a God class antipattern in the software product family architecture at run time. A God class is one that performs the majority of the work in the system and other processes are relegated to supporting and minor roles. The God class antipattern represents a poor architectural choice and has negative consequences on the performance since it constitutes a bottleneck and affects the maintenability and evolution of the software.

The core features analyzed are shared by all the products in the family and implied a shared architecture and source code between the products. As a consequence, the architectural problems were in both products analyzed. Managing the software product family evolution and performance is not an easy task; the system analyzed contained more than one million lines of source code and development teams worked in different parts of the world and in different time zones. Nevertheless, the tuning process proved to be helpful.

Software product family architecture has demonstrated its own benefits and advantages. As part of our process, the scenarios selected included only core features and therefore, performance improvements affected the entire set of products. In addition, the analysis demonstrated that the improvements did not influence the performance of other products negatively. This was verified by analyzing the feature dependency diagrams and by iterating the performance analysis process with the improvements in place.

5. CASE STUDY 2, MEMORY USAGE

The second case study was aimed at the performance tuning of the same Nokia software product family of case study one, but it focused on memory usage optimization [22].

The study targeted the dynamic memory usage of real-time embedded systems. When looking at performance optimizations, dynamic memory usage and CPU load are the most important issues. The cost of a product and its performance are affected by the dynamic memory management system. In embedded systems, the amount of dynamic memory is limited and consequently, optimizations are important. No compaction of memory was done in the system analyzed and requests were served using contiguous blocks of free memory. Problems in dynamic memory allocations arise when, because of fragmentation, a free contiguous block does not exist. In addition to memory usage efficiency, real-time embedded systems have time constraints that must be analyzed. Memory allocation and deallocation must respect real-time deadlines.

I have used the scenario-driven approach described in section 2, using key performance scenarios to drive the analysis. One handset was used for the simulations. The handset was part of the Nokia software product family, had multimedia features and included a super set of the mobile phone features in the family. Key performance scenarios were selected using not only core set features but also optional multimedia features. Examples of some of the key performance scenarios selected included phone start-up, browsing the web, downloading and playing java games and sending and receiving MMS (Multi Media Message).

By running the scenarios in the handset I extracted the data of allocations and deallocations. The data collected from the real-time embedded system through trace instrumentation was given as input to a simulation environment created for the experiment. The process data flow is described in Figure 3.
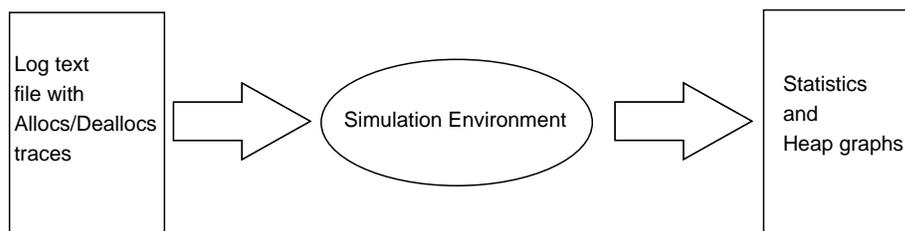
FIGURE 3: Data flow

The simulation environment was created to test different strategies and configuration parameters; the memory allocation requests were extracted by running the scenarios in the handset. For a critical review on how to design a simulator for dynamic memory management see [6]. Following the performance tuning process and using the simulation environment, I have evaluated six different allocation algorithms and simulated two heap layouts. A heap layout in our experiment could consist of a single contiguous block of memory, or a heap plus segregated free lists [22, 16]. Segregated free lists are preallocated sets of bins with fixed-size chunks of memory. The Doug Lea algorithm, used also in Linux, is an example of a dynamic memory management system which uses segregated free lists [14]. The 12 possible combinations have been analyzed following a set of metrics established at the beginning of the performance analysis, at the test plan phase. The metrics used were performance, external fragmentation and internal fragmentation of the algorithms analyzed. The algorithm simulated included the first-fit algortihm, the best-fit algorithm and various combinations of these algorithms using different heuristics. Every algorithm was analyzed using the two layouts presented, heap or heap with segregated free lists.

Algorithms and strategies can be targeted and exploit specific patterns of memory usage of the applications. For example, programs that use custom allocators are the Apache web server [1], the C++ Standard Template library, [19], and the GCC compiler [9]. Segregated free lists layout is advantageous to patterns with small memory allocations. However, requests for large blocks are not served efficiently by segregated free lists since preallocating large chunks of memory implies a possible waste of memory for unused large chunks.

Table 1 presents the data characteristics of a few scenarios analyzed. To understand the product family domain, an additional column has been added to the table to list the products implementing the feature and to specify if the feature is optional or part of the core features. In table 1 the

Scenarios defined	Number of Obj Allocated	Average Obj Size (Bytes)	Largest Block Size (Bytes)	Core/Optional Products with the feature
Sending and Receiving 5 MMS of 100 KB each	781563	92.67	96793	Optional: P1, P2
Downloading and playing a Java Midlet	119932	83.87	46363	Optional: P1, P2, P3
Browsing the Web	201744	70.90	38859	Optional: P1, P2, P3
Phone Start-Up	26811	50.02	18400	Core Feature

TABLE 1: Characteristics of the scenarios

products listed serve as a demonstrative example, the data characteristics of the scenarios are real. The last column in table 1 connects a feature to patterns of memory usage and to products implementing the feature.

The main results of the simulations have shown that the distribution of the blocks size has a strong impact on the performance of a dynamic memory management system. Core features presented patterns of small allocations while multimedia features allocated large blocks of memory. New features such as video call and streaming, will require larger contiguous blocks to be allocated in the dynamic memory.

Scientific literature has thoroghly investigated and analyzed dynamic memory management systems given determined patterns of allocations [12, 22]. An entire book describes how to manage memory in systems with limited memory [16] and dynamic memory management systems have been specifically designed for multithreaded applications [2].

Instead of focusing on the best algorithm to use, in this paper I will focus the discussion on the product family domain and on the implications of dynamic memory management in product family architectures. In the context of a software product family, I analyzed the tradeoffs of a dynamic memory management system for a set of products. The dynamic memory management system was part of the software product family architecture and was instantiated without changes in all the products. However, the simulation demonstrated that multimedia features, not part of the core features, needed large contiguous memory blocks in the heap. Instead, the core features of the product family analyzed were better served by layouts and algorithms that advantage small block allocations like segregated free lists. The issue was not obvious before I performed our study and before I highlighted the measurements and analyzed the memory usage. Given the different features' requirements, the analysis considered the tradeoff of custom dynamic memory management systems for different mobile phones against a common dynamic memory management system that could work well enough for the situation. Analyzing the tradeoffs between memory efficiency and software product family evolution, I chose a common dynamic memory management system which avoided the need for a variation point in the architecture. A common heap layout and algorithm simplifies product family management; it works with performance compromises but it was demonstated to be satisfactory for the system studied at that point in time.

6. DISCUSSION

Performance tuning is the process by which software is optimized. Software performance improvements can be included in different stages of the software life-cycle; however, performance is not always considered when software is designed. Furthermore, software evolution and changes can negatively affect performance. New requirements could introduce performance problems and the need for a different architecture design. Due to the reasons above, a performance tuning process is advised especially considering software evolution. Sometimes, even though performance objectives are respected we may want to investigate further improvements and better alternatives.

The ability to optimize a software system implies also cost saving. By using better dynamic memory management systems, memory fragmentation can be contained, with the consequence that less memory is needed to run the same scenarios. By re-factoring the software architecture using performance patterns and eliminating the bad design decisions by finding performance

antipatterns, we optimize the architecture design [20, 8]. Optimized software means that the software can run in less powerful hardware, meet its performance objectives and respect real-time deadlines. Software performance is a competitive advantage and favors the company's image: poor performance in the user interface or in running applications will lead to customer dissatisfaction.

When dealing with a software product family, performance optimizations must cover the wider scope of performance for a set of related products. In the software performance engineering method (SPE) [20], a set of performance scenarios are chosen and performance improvements are evaluated against the impact on the whole system. However, a new dimension to the problem is added when performance tuning is targeted to a software product family.

A software product family architecture provides the reference architecture for a set of products. The products part of a family share a common reference architecture, part of the modules and hardware platforms. Therefore, it is evident that improvements in one product will affect a whole set of products. However, what is not evident is how to proceed in the evaluation of the performance of software product family architectures and description of the process is missing in the literature.

In this paper I have proposed a method for performance tuning a software product family and I have described two cases studies. The method has been adapted from the SPE method but I have enhanced it to consider software product family architectures. When tuning software, a set of features are chosen as significant for the analysis. In a software product family domain, improving the set of products means improving the set of core features. The core features are the set of features shared by all the members and improvements to common features imply improvements to the common assets. Therefore, when tuning is aimed at the common denominator, during the scenarios selection phase, the core features are included and analyzed.

Performance tuning is not necessarily targeted at the core features set. Software evolves and the process of software evolution includes the enrichment of the architecture with new features. Commonly, forthcoming features demand more in terms of hardware, memory and throughput (if the devices have connectivity). In order for the architecture to evolve, it should also be able to both maintain performance characteristics and satisfy future requirements. In mobile phones, multimedia features require an architecture able to handle large flows of data and also require particularly demanding time constraints. These requirements constitute challenges to performance since they did not exist when the architecture was created.

The ability to incorporate new features into the architecture is very important for the evolution of a software product family [21, 15, 18]. New features are usually included as optional (non-core) and they will be instantiated in some leading products of the family. When the tuning is targeted to the new features the key performance scenarios will include the optional features. In this case, the focus of the analysis will be on a sub-set of the products of the software product family.

The features described in the scenarios are evaluated and improvements and changes are proposed. Evaluation of the performance analysis includes the analysis of tradeoffs. Performance, as a non-functional quality attribute, has implications for other software quality attributes. For example, software security, software flexibility and fault-tolerance usually come at the expense of software performance. Increasing performance can sometimes mean an increase in hardware capability, i.e. a more powerful CPU, more memory. However, the cost of a product has to be considered and the right balance must be found.

The evaluation process includes iterations after the improvements are applied. Iterations are needed to determine if improvements have unexpected drawbacks. For example, changes in a module can lead to unexpected consequences in other modules. An additional step is the consideration of the product family architecture dimension. The improvements must also consider the implications to the entire family of products. When an improvement is considered, the impact on the product family must be evaluated.

Impact analysis follows the links and connections in the software architecture and in the feature model, see figure 4. In the feature model, a feature has dependency links to other features. One example of feature model analysis is the FODA method [13]. In addition, every feature has dependency connections to the software architecture, source code, its modules, classes and libraries. Determining the implications of the changes is not an easy task but is an important and fundamental step required for understanding the software analyzed and for performance analysis evaluation. The features model and the software product family architecture with their description of the commonalities and variabilities are the models we use for impact analysis.
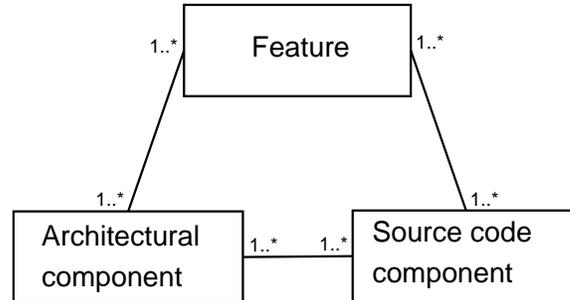
FIGURE 4: Dependency between features, architecture design and source code level

The consequences of improvements can lead to revisions to the reference architecture of the software product family. For example, changing an immutable core feature to optional or moving a functionality from the core to the optional set of features and vice versa. Scenarios based on core features focus the analysis on the reference architecture. On the other hand, when the focus is not on core features but on optional and particularly demanding features (in terms of resources and time constraints) the analyst should not discard the implications for the product family reference architecture and its own common assets.

In case study two, we have seen that multimedia features demand a dynamic memory management system optimized for large chunks of contiguous memory. On the other hand, low-end mobile phones need to be optimized for small amounts of memory and small memory block allocations. The decision to find a compromise in the dynamic memory management architecture was made considering the tradeoffs in the variability of the software product family architecture against its performance. When a component is moved out from the core asset or is made parameterized, a cost is paid in terms of evolvability and maintenability. The strength of a product family architecture is based on the commonalities of the assets. Increasing the variability comes at the expense of the commonality of the architecture while the benefits of the software product family architecture diminish. Refactoring the architecture and increasing the number of core features is a good engineering practice. In some cases, however, the split into a new branch with the creation of another software product family architecture can be necessary and it is part of the software product family evolution: the new requirements have created the need to refactor the architecture.

7. CONCLUSIONS

A software product family architecture is an architecture designed for a set of related products. The products in the family share a common reference architecture and common assets. At the same time however, variation points exist and variability permits the differentiation of the products.

The new dimension introduced by the software product family representing a set of products constitutes the main challenge when analyzing performance. Performance improvements must consider and analyze the implications to the entire family of products.

I have presented a scenario-based approach to tackle the performance of software product family architectures and I have used two case studies to illustrate the application of the method. The focus of the analysis is established by the scenarios selected and the key performance scenarios

are the outcome of the scenarios selection process. Key performance scenarios can include core features or optional features of the software product family. By selecting core features, the analysis targets the core assets. By selecting the optional features, the analysis focuses on functionalities instantiated only in a few products. However, in both cases the analysis must verify how the enhancements affect the entire software product family architecture. The evaluation step considers the tradeoffs and the improvements against the entire set of products. The analysis of the tradeoffs is described in section 6.

The process described permitted an evaluation and optimization of a product family architecture in Nokia. The scope of this paper was aimed at the understanding of software performance in the context of a software product family. Future work will further investigate the analysis of the performance of software product family architectures during the entire software life-cycle.

REFERENCES

[1] Apache Foundation. Apache web server: http://www.apache.org.

[2] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, 28,34(5,5):117–128, November 2000.

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

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, and P. Sommerlad. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, 1996.

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

[6] C.-T. Dan Lo, W. Srisa-an, and J. M. Chang. The design and analysis of a quantitative simulator for dynamic memory management. *The Journal of Systems and Software*, 72(3):443–453, August 2004.

[7] C. Del Rosso. Performance analysis framework for large software intensive systems with a message passing paradigm. *Proceedings of 20th Annual ACM Symposium on Applied Computing, track Embedded Systems: Applications, Solutions, and Techniques (EMBS), Santa Fe, New Mexico, March 13 -17, 2005*, November 2005.

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

[9] Free Software Foundation. Gcc home page: http://gcc.gnu.org/.

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

[11] H. Gomaa. *Designing Software Product Lines with UML*. Addison Wesley, 2004.

[12] D. Grunwald and B. Zorn. Customalloc: Efficient synthesized memory allocators. *Journal Software: Practice and Experience*, 23(8):851–869, August 1993.

[13] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson, A. Feature-oriented domain analysis (foda). *Software Engineering Institute, Carnegie Mellon University technical report*, (CMU/SEI-90-TR-021), 1990.

[14] D. Lea. A memory allocator. *http://gee.cs.oswego.edu/dl/html/malloc.html*.

[15] M. M. Mattsson and J. Bosch. Stability assessment of evolving industrial object-oriented frameworks. *Journal of Software Maintenance: Research and Practice*, 12(2):79–102, 1999.

[16] J. Noble and C. Weir. *Small Memory Software: Patterns for system with limited memory*. Addison Wesley, 2001.

[17] C. Riva. View-based software architecture reconstruction. *PhD disseration, Vienna University of Technology*, October 2004.

[18] 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.

[19] SGI. Standard template library: http://www.sgi.com/tech/stl/allocators.html.

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

[21] M. Svahnberg and J. Bosch. Evolution in software product lines: Two cases. *Journal of Software Maintenance: Research and Practice*, 11(6):391–422, Dec 1999.

[22] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. *Proc. Int. Workshop on Memory Management*, Kinross, Scotland, UK, September 1995.