Performance-Oriented Software Architecture Engineering: an Experience Report

Chung-Horng Lung Anant Jalnapurkar Asham El-Rayess
SEAL - Software Engineering Analysis Lab
Nortel, Ottawa, Ontario, Canada
{lung, jalnapur, asham}@nortel.ca

Abstract

Current methods for software architecture analysis often fall short of providing objective and quantitative performance information. The paper describes how to bring together techniques in software performance engineering and software architecture analysis in order to support performance-oriented software architecture engineering. The paper presents a systematic approach derived from empirical case studies in real-time telecommunications applications. The approach has been successfully applied to these case studies to help product teams analyze and improve the performance and other quality factors of their systems.

1. Introduction

Many systems, especially real-time applications, fall short of meeting the performance goals set by the designers. Unfortunately, the problems are often discovered only late in the application life-cycle. At this stage, a lot of effort will be spent in tuning the performance or restructuring the architecture or design. This effort usually results in architecture erosion or drift. In other words, the implementation does not conform with the architecture, if there exists one. This problem further complicates the already time-consuming maintenance process for product evolution.

Software architecture analysis is an emerging field, promoted by the increasing complexity of software systems and the need to reduce maintenance costs for evolution. A software architecture analysis group was established in the Software Engineering Analysis Lab (SEAL) in 1995. Since then, we have worked with various Nortel product teams to evaluate their software architectures. Our initially adopted methodology was the Software Architecture Analysis Method (SAAM) [3]. Since many of the applications we worked with were real-time telecommunication applications, new challenges, especially recurring performance issues, have risen. These new challenges necessitated creating extensions to our approach.

Recently, we have been focussing more on the performance aspects during the evaluation process. Software Performance Engineering (SPE) [9] enforces a performance assessment step in the design stage before proceeding. SPE emphasizes the performance model construction and evaluation. However, SPE does not explicitly address the software architecture issues and how to re-engineer a system to improve performance. This paper describes a Performance-Oriented Software Architecture Engineering (POSAE) approach. POSAE integrates SPE and software architecture practices. We have used the approach to aid product teams in identifying potential bottlenecks and improving the performance of their systems. Furthermore, the approach also helped us look for opportunities to re-engineer legacy systems, identify reusable components, update documents, and build prototypes.

This paper reports empirical experiences and lessons learned from practical applications of the method. This paper can serve as a bridge between practitioners in software architectures and performance engineering. This paper supports practitioners in software architecture by demonstrating an explicit and systematic approach to conducting analysis and synthesis with performance as the most critical factor. The paper also brings to the attention of practitioners in performance engineering some of the techniques used in software architecture.

The paper is organized as follows. Section 2 introduces a set of architectural views adopted to support the analysis and synthesis. Section 3 illustrates POSAE, including the steps used and the types of artifacts produced. Some results and lessons learned from empirical studies are presented Section 4. Lastly, Section 5 provides concluding remark.

2. Architectural Views

The development of a complex software system always involves diverse stakeholders. Different stakeholders have different perspectives and needs. The description of an

WOSP98, Santa Fe, N.M. (c) 1998 ACM 1-58113-060-0/98/00... $5.00

architecture requires multiple viewpoints at various stages in the life cycle to meet the stakeholder's objectives.

SEAL has adopted various architectural views that are critical for POSAE. The set of views, as demonstrated in Figure 1 [5], includes a static view, a map view, a dynamic view, and a resource view. These architectural views are developed to support insightful understanding of the system and to facilitate communications among various stakeholders. The next section discusses how these views are applied to POSAE. Each view and some commonly used methods for the view are briefly described below.

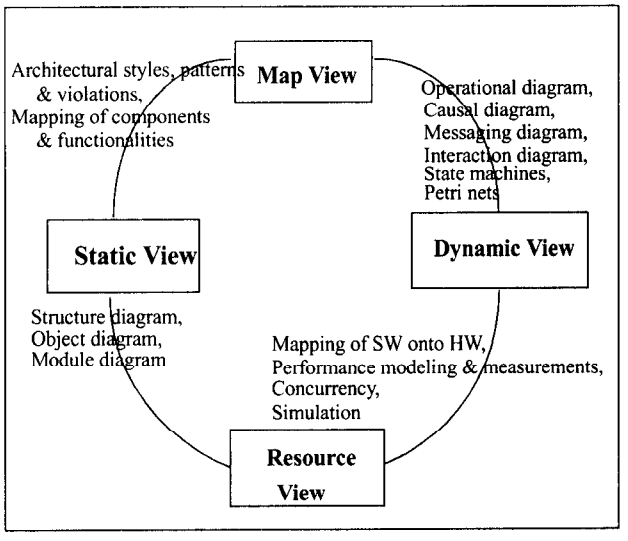

Figure 1. Software architectural views

- **Static view**. The static view shows the overall topology of system components and their interconnections. The methods that can be used for this view include logical diagram, structure diagram, object diagram, and module diagram. The static view can also be represented at various levels of detail, depending on the stakeholder's objectives.
- **Map view**. The map view identifies the architectural styles, patterns, and design violations. The identification of the styles or patterns helps us focus on the control and communication mechanisms. In addition, this view also identifies the mapping between components and functions or features. The mapping is used to cluster related components into cohesive modules.
- **Dynamic view**. The dynamic view addresses the behavioral aspects of a system. This view can be supported by functional diagram, causal diagram, messaging diagram or message sequence chart, object interaction diagram, state machines, and Petri nets.
- **Resource view**. The resource view deals with the utilization of system resources. Various techniques can be used, including the identification of mapping of software onto hardware, performance modeling, measurements, parallel or concurrent processing, and simulation.

The development of the views does not have to be carried out in a strict sequence. Rather, the process is iterative and incremental. Furthermore, not all views are needed for each evaluation and each view is not constrained by a particular method or notation. Selection of appropriate views and suitable methods depends on the specific application environment and values of the stakeholder.

3. Performance-Oriented Software Architecture Engineering

This section describes the POSAE approach. We have applied this approach to various applications in the telecommunications problem domain. This section describes the process and some approaches used. We also correlate the process with the architectural views discussed in the previous section. Note that the process is also iterative and incremental in nature. The process involves the following steps. More elaboration on the steps as well as some of the derived lessons follow.

1. Develop or capture a software architecture.
2. Identify most frequently used scenarios, with focus on real-time scenarios.
3. Identify execution paths for the scenarios.
4. Apply performance modeling, analysis, and measurements.
5. Perform architecture analysis based on the results of step 4.
6. Conduct trade-off analysis between performance and other quality attributes, and among design alternatives.
7. Build a prototype, based on the analysis, to improve the performance or other quality attributes.

1. Develop or capture the software architecture. This step corresponds to the static view shown in Figure 1. The main components and their interactions are captured. For complex systems, it is unlikely that people will continuously remember all the details. An explicit software architecture is a useful vehicle for communication among various stakeholders. In some cases, even the static view could display potential performance impact due to complex interconnections among components.

For legacy systems, reverse engineering of existing implementation is often necessary. A few commonly used approaches are adopted together in the process to extract the architecture. Those approaches are read documents, interview designers, use reverse engineering CASE tools, and walk through source code.

Reading documents and interviewing designers are good starting points. The approach gives a quick overview of the

system, but the approach usually only captures the high level view. Also, the documents may not be kept up-to-date. Using a reverse engineering tool facilitates the traversal of source code and capture of different views. However, for complex object-oriented software, existing tools may not generate complete or correct artifacts. Furthermore, human intervention and evaluation are often necessary to capture more semantic information and domain knowledge. Source code represents the actual implementation of the system. Walking through the code validates the captured architecture. These approaches are used iteratively. By applying these approaches, we often could develop artifacts of different levels of abstraction. Those artifacts are valuable information for the designers as most of them are only working on a certain specific area.

2. Identify most frequently used scenarios, with focus on real-time scenarios. Obviously, the most frequently used scenarios have direct impact on the overall system performance. For example, the basic phone call for two parties is the most frequently used scenario for the call processing system. Other features are actually built on top of it. Therefore, evaluation of the basic phone call scenario is a must for the call processing system. This step usually can be conducted in parallel with step 1.

For some applications, it is also important to identify scenarios that address non real-time impact on real-time processes. These scenarios may add unexpected performance overheads, but are often neglected in performance evaluation. We have used a systematic approach to generate scenarios based on the Objective/Scenario/Metric paradigm [5,6]. For each objective, a tree-like structure of scenarios are developed. Each node in the tree is either a specific or a generic scenario class which in turn is a collection of scenarios or scenario classes. This approach is useful in determining scenario coverage and balance, and in deciding when to stop generating scenarios.

3. Identify execution paths for the scenarios identified in step 2. This step corresponds to the dynamic view shown in Figure 1. Once the architecture is developed or captured, and the critical scenarios are identified, we traverse the architecture with the scenarios. The concept is similar to the SPE approach.

Various approaches have been used to model the dynamic aspects of a system. Among these, we have used message sequence charts, causal structures, and Use Case Maps (UCMs) [1]. There are advantages and limitations for each approach. Detailed comparisons of these appraises are beyond the scope the this paper. We have also used an approach that is similar to UCMs. The UCM notation facilitates the visual representation of execution paths, but the approach does not include rich semantics. This can be remedied with causal structures [2,5]. Figure 2 and 3 show a simplified example of how these two approaches are used together to represent dynamic behaviors and execution paths. Each block in Figure 3 could present information on object (in *italics*), method, or data that are involved. The combination of these two approaches is useful to show component interactions and execution paths, especially early in the preliminary analysis process.

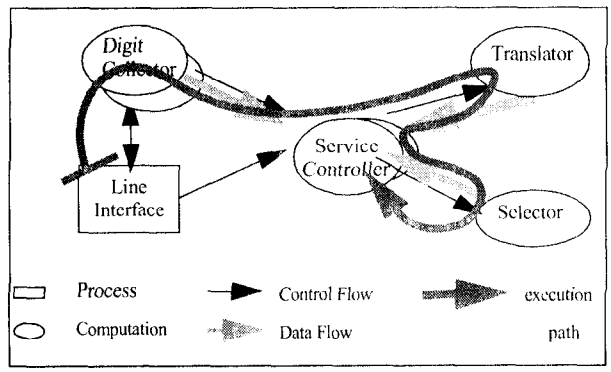

Figure 2. Example of Architecture walk-through

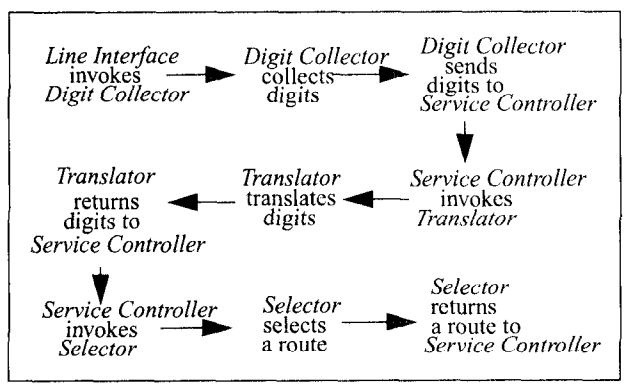

Figure 3. Causal representation of architecture walk-through

4. Apply performance modeling, analysis, and measurement. This step is related to the resource view depicted in Figure 1. The key idea is to examine the utilization of resources to improve or maximize system performance. This step provides quantitative information for system performance at various levels.

The application of SPE approach, including modeling, analysis, and measurement, to the software design cycle has been hindered by a variety of factors:

Time constraints often force project managers to focus merely on functionality. Performance engineering in such projects is restricted to code-level optimization. Code-level optimization is necessary but not sufficient for complex systems. The existence of software bottlenecks in such systems may result in disastrous performance results.

SPE requires special skills in a domain that might be different from the project domain. The integration of SPE into the project requires knowledge of both domains.

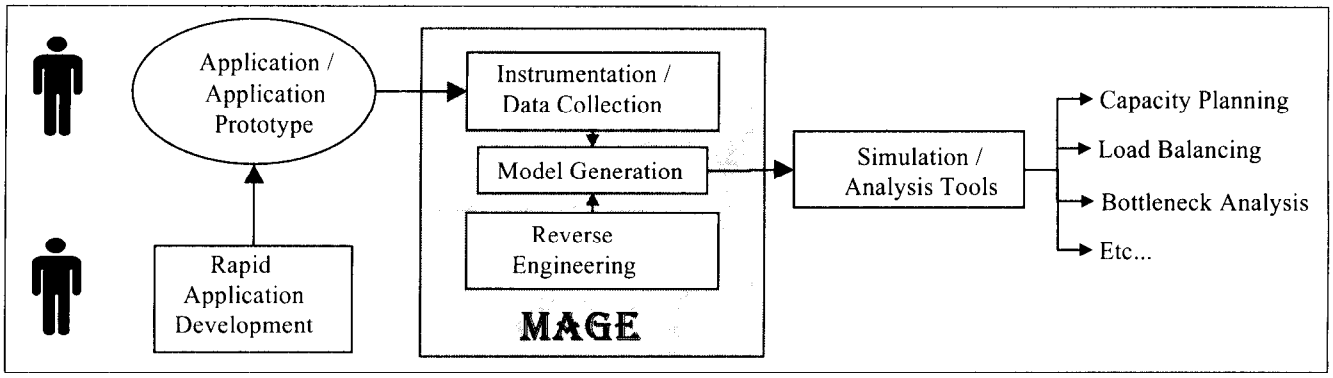

Figure 4. Overview of the Model Automatic Generation Environment

Our approach to solving these hurdles is to automate parts of the SPE process. We are working on the development of a Model Automatic Generation Environment (MAGE). MAGE (Figure 4) is a tool to be used for automatic generation of simulation and/or analysis models from existing applications or application prototypes.

The tool user (software programmer) embeds a set of data collection APIs at specific points inside the application code. These APIs measure, capture, and gather the architectural and performance characteristics of the application. The tool includes a model generator that can make use of the data to automatically build performance models for a variety of simulation and analysis tools. The models are used by the software performance engineer to perform bottleneck analysis, capacity planning, load balancing, and other performance related tasks. Software designers can also use Rapid Application Development tools to produce a prototype that can be similarly used to generate initial performance models.

By automating the process of constructing performance models, we de-skill and reduce the time needed for the integration of SPE methodologies into software development.

Depending on the types of applications, different modeling techniques could be selected. We have applied the Layered Queuing Network (LQN) model [7,10] to client-server architectures in order to identify potential software bottlenecks, investigate the impact of size of critical sections on performance and the optimal number of concurrent processes for maximal performance for multiprocessing systems. Detailed discussion of the application is described in [8].

5. Perform architecture analysis based on the results of step 4. The main purpose of this step is to evaluate the overall software architecture. Some critical areas of focus include elimination of redundant operations, reduction of execution frequency of some paths or components, and avoidance of levels of indirection. For examples, the initialization operation may be invoked repeatedly, redundant memory copy operations may be performed, or a module may be a contention point or over-executed. All these operations have impact on performance.

To achieve the goals of this step both top-down and bottom-up approaches are used. We start at the architecture level but validate from lower level artifacts, if available. The execution paths identified in step 3 are also refined in this step. In other words, more detailed analyses are conducted. We also identify architectural styles or design patterns. The objective is to focus on the features of the styles or patterns. The primary features consist of the control mechanism, communication mechanism, registration mechanism, and other quality factors. Violations of the styles and the rationale are also explicitly captured. Some violations may be potential areas for improvement.

The information could support the trade-off analysis, which is described next. In addition, for some cases, we analyze if the components could be better partitioned or clustered from the cohesion and coupling point of view. Figure 5 shows an example of improvement of modulization from (a) to (b) by simply regrouping components [4]. Systems with de-coupled modules can greatly improve evolution and could also improve performance as well. This step is closely related to the map view shown in Figure 1.

6. Conduct trade-off analysis between performance and other quality attributes, or among design alternatives. Performance usually is not the only concern of stakeholders. We may need to consider other attributes such as maintainability, scalability, and time-to-market. Some of these attributes may contradict with the performance goals. The development of a set of related and concrete scenarios, with different views, helps the trade-off analysis.

Trade-off analysis could also be applied to cases where multiple design alternatives exist. Lung and Kalaichelvan [6] propose an approach to support an objective and repeatable evaluation. The idea is to break down the evaluation into more fine grained levels and identify the sensitivity of the attributes or design alternatives with respect to the key stakeholder objectives.

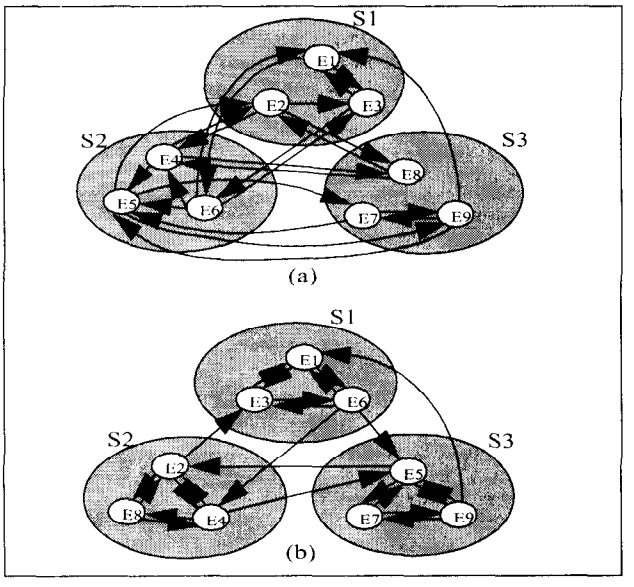

Figure 5. Regrouping of components to reduce coupling: an example

7. Build a prototype, based on the analysis, to improve performance or other quality attributes. Identifying the performance bottleneck is an important step. However, the actual value lies in the removal of the bottleneck. Based on the evaluation of the software architecture and performance, we build prototypes to demonstrate how to eliminate certain problems or to improve performance and other system qualities. The approach has been used in one instance to synthesize an architecture based on design patterns for the problem area. The new design greatly reduced coupling between some system components and, at the same time, improved the overall performance.

4. Results and Lessons Learned

We have applied the above approach at Nortel's SEAL in order to perform end-to-end analysis on various real-time telecommunication products. One specific example is the Nortel Service Control Point (SCP) product. We conducted performance analyses for the messaging system, run-time environment, application framework, and the high-level services and applications. The performance is defined as system throughput under quality of service (QoS) constraints. We also make concrete recommendations to the design teams, and demonstrate feasibility and added values.

Some results and values of this approach include:

- Capturing various views at different levels of abstraction enables better understanding of the architecture by the design teams
- Developing and perusing a variety of scenarios helps identify risk areas
- Identification of synchronization points and potential bottlenecks in a concurrent environment.
- Performance improvement of the run-time environment, application framework, and services and applications.

In short, we helped the design teams increase performance and improve software quality. In one instance, through use of the above approach, we managed to improve the throughput of a certain system by 25%. In another instance, the throughput of a certain application has improved by 500%.

A 25% gain of system throughput was achieved for the real-time toll-free telephone number look-up (1-800 service). Based on the existing architecture, we first used the techniques described in step 2 of POSAE to identify critical scenarios as query and update to the real-time database. Analysis of the scenarios, as described in step 3, showed that the main application process remains in a critical section for approximately 25% of the time required for processing a query. A 4-processor system was used to implement the product and there were four such application processes. Modeling techniques were then used [8] to determine the optimal length of the critical section. Based on the critical section analysis results, appropriate design changes were suggested to reduce the length of the critical section. This resulted in 25% improvement in the system throughput without affecting the quality of service.

A 500% gain of performance was achieved on an intelligent network application implemented in a unit-processor environment. Analysis of key scenarios in this application revealed a critical interaction among query, update, and OS scheduling mechanism. Update processing, in this case, was 50 times more expensive compared to query processing. The application process performing updates did not relinquish the CPU for the entire quanta (1 second). This resulted in loss of queries. The update processing algorithm was optimized with the help of the design team and it was run at a lower priority. This resulted in 500% improvement of the system throughput.

We have also learned many lessons from the actual field experiences. These lessons include:

Software architecture is a critical asset for the designers. Quite often the designers' focus is on one specific area of the system. With an architecture in place, the designers can appreciate the overall system components and how they work together. An explicit software architecture helps designers understand the system and facilitates communication among the various design teams. On the other hand, current software architecture analysis practices often stop at the reverse engineering and problem identification stages. Design teams often react better to explicit remedial

recommendations. These recommendations, as such, have a better chance influencing the system. In addition to pure analysis, software architecture should support an engineering discipline.

To benefit the most, SPE must be closely tied to software architecture. Performance and other quality attributes may be constrained by a software architecture. Traditional SPE methodologies promote the construction of a performance model early in the design cycle. Efforts are undergoing for automatic generation of performance models from high-level design descriptions such as UCMs. These models help provide crude estimates that can be useful, for example, in identifying critical application paths.

In order to refine and validate the constructed models, application response time and resource usage information should be also provided. In simple centralized systems estimates based on expert intuition were often sufficient for constructing useful models. We have found that this is not often true for complex distributed systems. For this purpose we advocate the rapid construction of a functional skeletal prototype using the technology of choice for the application. The prototype is instrumented using MAGE instrumentation APIs (See Step 4 in Section 3). MAGE automatically generates a performance model that can be used for architecture assessment, feasibility analysis, and capacity planning.

Prototype development is also useful in demonstrating design alternatives and showing values. The design teams are usually tied up with deliverable products. Assessing a system and identifying hot spots or bottlenecks may not be sufficient to solve the problem or have a great impact. Often, we need to provide the designers with partial solutions to evaluate concrete impacts before they actually modify the design.

Domain knowledge plays a critical role in re-architecting or re-engineering a system. There is no replacement for it. We usually need to spend a lot of time understanding the problem domain to make contributions. We may also need to spend a lot of time building the performance model and conducting analysis. This process needs to be shortened to provide prompt feedback to the designers.

5. Conclusion

In this paper, we described an approach called POSAE. The main contribution of this paper is to put together some existing ideas in software architecture and software performance engineering. We presented a systematic approach to the performance assessment of software architectures either at an early stage in the design cycle or at the re-engineering stage. POSAE also emphasized the role of re-architecting or re-engineering rather than pure analysis or simply reverse engineering.

There are still other challenges that lie ahead in this area. One challenge is to develop robust tools to construct precise call graphs for object-oriented software and to generate reliable performance measurements. Tools are essential to automate the reverse engineering process to capture the current architecture, especially software architecture evolves over time. The results generated by existing commercial tools are often incomplete or incorrect. We also found that different tools generate different results. Similar results are also observed for performance measurement tools.

The ultimate goal of this approach is to aid design teams to reduce time-to-market and to meet the stakeholders objectives. We are building tools that automate the process of capturing performance and architectural characteristics of systems, making it easier to integrate the SPE methodology into software development. We are also investigating and characterizing the performance of design patterns. Patterns provide opportunities to produce high-quality designs in shorter time periods. However, they are still immature and more information is needed to guide developers to select the appropriate patterns. Adequate training is also required to support the process.

References

[1] R. Buhr and R.S. Casselman, *Use Case Maps for Object-Oriented Systems*, Prentice Hall, 1996.

[2] G.J. Holzmann, D.A. Peled, and M.H. Redberg, "Design tools for requirements engineering", *Bell Labs Tech. J.*, pp. 86-95, 1997.

[3] R. Kazman, G. Abowd, L. Bass, and P. Clements, "Scenario-based analysis of software architecture", *IEEE Software*, Nov 1996.

[4] C.-H. Lung, "Effective software partitioning through clustering techniques", *Nortel Design Forum*, Oct. 1997.

[5] C.-H. Lung, S. Bot, K. Kalaichelvan, and R. Kazman, "An approach to software architecture analysis for evolution and reusability", *Proc. of CASCON*, pp. 144-154, 1997.

[6] C.-H. Lung and K. Kalaichelvan, "An approach to quantitative software architecture sensitivity analysis", *to appear in Proc. of the Int'l Conf. on Software Eng. and Knowledge Eng.*, 1998.

[7] J.A. Rolia and K.C. Sevcik, "The method of layers", *IEEE Trans. on SE*, 21 (8), pp. 689-700, 1995

[8] D. C. Petriu, C. Shousha, A. Jalnapurkar, and K. Ngo, "Applying performance modeling to a telecommunication system", To appear in *Proc. of Int'l Workshop on Software and Performance*, 1998.

[9] C. Smith, *Performance Engineering of Software Systems*, Addison-Wesley, 1990.

[10] J.E. Neilson, C.M. Woodside, D. C. Petriu, and S. Majumdar, "Software bottlenecking in client-server systems and rendezvous networks", *IEEE Trans. on SE*, 21 (9), pp. 776-782, Sept. 1995.