Performance Analysis Framework for Large Software-Intensive Systems with a Message Passing Paradigm

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

ABSTRACT

The launch of new features for mobile phones is increasing and the product life cycle symmetrically decreasing in duration as higher levels of sophistication are reached. Therefore, the optimization of resources is particularly important in embedded systems where CPU power and memory space are limited. In this context, performance engineers must be able to predict and analyze the performance of the software architecture in order to support its evolution and its new requirements. In this paper I describe a framework for the analysis of the performance of a software architecture where the architectural elements communicate using message based communication services. Using instrumentation traces I have extracted the run-time events I considered significant for the study. I have created a set of architectural views to reconstruct the dynamic and the static views of the architecture. Understanding the connections and the relationships between them guide the performance analyst to a clear comprehension of how the architecture works and subsequently how it can be optimized. The performance analysis framework described constitute an essential set. The next challenge is to enhance the integration of the tools and the synchronization of the views and to facilitate the entry barrier to novice performance engineers.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*Performance measures, product metrics*; D.4.8 [**Operating Systems**]: Performance—*Measurements*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*; D.2.5 [**Software Engineering**]: Testing and Debugging —*Testing tools, Tracing*; C.4 [**Computer Systems Organization**]: Performance of Systems—*Measurement techniques*

Keywords

Software performance engineering, dynamic analysis, performance tuning of embedded systems

1. INTRODUCTION

To predict or measure the performance of large software systems [6] is a challenging task. Software evolves and new features are incorporated. Mobile phones are shifting from simple devices used to communicate to complex software systems with the ability to make phone calls plus all the functionality of a handheld computer. New multimedia applications and high-speed data transmission protocols represent crucial features the mobile phone architecture must support.

The work of the performance analyst is to highlight hotspots and bottlenecks in the architecture in order to optimize it. In some cases, where real-time constraints are involved, the optimizations allow the architecture to evolve and respect strict forthcoming requirements. In embedded systems, limitations are even more severe in terms of CPU power and memory space and the economy of resources is a very important factor to consider when designing the architecture.

Different techniques exist to analyze a software system and they can vary depending on the status in the life cycle of the product. Modelling, simulation and prototyping can be applied from the early stages in the software development cycle. If the software has already been built, however, profiling using instrumentation constitutes a viable option. Additional source code (instrumentation code) is added to the software source code to examine the run-time behavior of the program. With the instrumentation in place different events can be monitored and the focus of the analysis will affect what traces will be enabled at run-time using some flags i.e. at compile time or in header files. As in every methodology, pros and cons of using instrumentation exist, for example adding traces affects the behavior of the system, and it requires access and knowledge of the source code [9], [14], but a discussion of this topic is beyond the scope of this paper.

Using instrumentation traces allows an off-line analysis: the events raised during the scenarios are collected in text files that will be analyzed using different analysis tools.

In this paper I will describe the performance analysis framework needed to investigate the behavior of an architecture using the message passing paradigm as an inter-process communication mean.

During the work as performance engineers in Nokia, I have developed a performance framework of tools that analyze the dynamic and static behavior of the system that focus on the CPU load. The framework creates different views that used together, can help to understand how and where a system can be tuned to improve the overall performance.

The framework developed constitute a starting point and has been used in Nokia. The project is progressing and improvements can be made by adding new tools and improving the existing ones.

2. PERFORMANCE TUNING OF EMBEDDED SYSTEMS

Performance as a non-functional quality attribute is not given the importance and the priority it deserves. In mobile phone software, performance is important for several reasons: real-time constraints must be respected, multimedia applications are now considered an essential part of the core software, memory requirements are increasing and power consumption must be limited.

The reduction of the consumption of resources such as CPU and memory must be carefully engineered, and strict requirements must correspond to specific architectural choices made at the architecture design phase. A correct design and implementation will allow the architecture to evolve.

However, even if the system is already built we can still enhance performance through a tuning process and a tuning framework created for that purpose.

3. THE PERFORMANCE ANALYSIS FRAMEWORK

The performance analysis framework allows different views of the architecture to be built. The framework allows a better understanding of the software system, and subsequently, several optimizations will follow from the augmented knowledge. In the sections below I will explain how I reconstructed the views; after that I will describe how to take advantage of the framework and explain the benefits and optimizations that can be obtained.

4. RECONSTRUCTION OF THE ARCHITECTURAL VIEWS

Data gathered from significant scenarios constitute the input for the dynamic analysis tools, the source code is the input for the static analysis tool, see figure 1. Collecting data is the first step in the process and the performance engineer must select a set of scenarios in cooperation with the software architect. The process of tuning was previously described in an experience paper [2]. I conform to the definitions of views given by the standard IEEE 1470 [6] and by Paul Clements and et.al. [1].

4.1 CPU Load view

To calculate the time spent by each process in the scenario I use the scheduling information contained in the log file: scheduling of the processes and time stamps. A table describing the percentage of the CPU time spent in each time slice was created and different visualization tools were used, e.g. an excel graph. The CPU load graph gives a high level of view of what is happening in the system. A stacked area graph provides the contribution trends in the load for each process over time. Peak load activities or high frequency activities can be easily located in the graph. The figure 2 shows an example of a CPU load graph where two processes called A and B are involved.

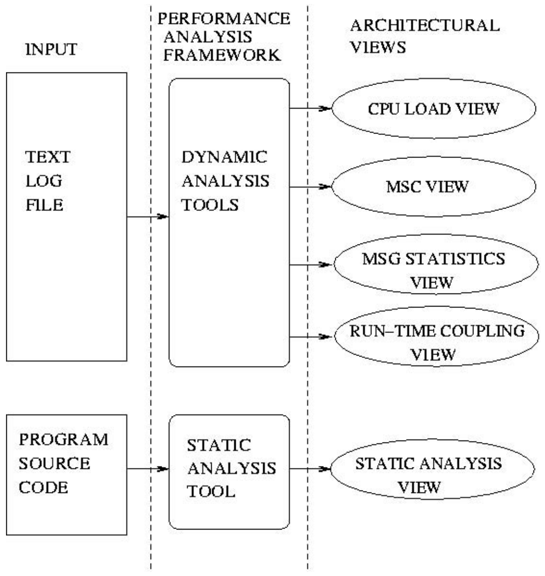

Figure 1: From the Data to the Views

4.2 Message Sequence Charts with Time Stamps

For the analysts, knowing which process contributes most to the peak load is not enough. I want to have a finer view of the system and know what the process is doing at that peak, and essentially what part of the source code is running at any given time. A MSC (message sequence chart) diagram showing the messages with time stamps can help.

I have enhanced a visualization tool showing MSC [13] with time stamps to provide information on each message exchanged,. The information used for the visualization is all contained in the log file and include the time stamp, sender, receiver, plus optional information about the message type. In figure 2 there is an example of MSC where A and B are exchanging messages m_i at the time t_i.

4.3 Message Statistics View

A message itself is not a ”time-consuming” activity but the information that it conveys is the source of the processing. The message could be a request for a particular resource, an acknowledgement or contain only data. The statistics tool developed gathers the statistics of the CPU time spent in processing different types of messages and can be used to unfold interesting activities. The assumption is that is possible to extract the begin and the end of the processing time for each message from the traces. An *End of Message Processing* trace instrumentation can be used to clearly mark the end point.

In a pure message passing system the statistics uncover the activities that have utilized more CPU time during the simulation. In a hybrid system, in which function calls are also used, the statistics are still valid and provide valuable data to the analyst.

4.4 Run-Time Coupling View

In a multi-tasking system, the processor must switch between different processes all the time. Every process has a private address space to allow multiple instances of the same program to run. If a message is sent between different processes, the process address space boundary must be crossed and a context switch must happen. Inter process communication has some overhead, and in case the communication involves a context switch, more processing time is implied.

I have used the number of messages exchanged between different processes as a metric of coupling even though different definitions of coupling exist, [3]. Highly coupled processes have an high rank arc in a directed weighted graph.

The coupling tool that was developed measures the number of messages exchanged between every single component in the system. The figure 2 show an example of the outcome when three processes, A, B and C are involved.

To limit context switches, highly coupled processes could be merged. This is an advisory and can aid in a decision to merge; other variables must be taken into account such as location of the development teams and architecture modularity.

At the same time, the coupling graph gives the network topology of the system and software anti-patterns can be discovered [4], for example the *God Class* anti-pattern described in [14], and found in an industrial experience report [2].

4.5 Static Analysis view

Dynamic analysis tools permit the visualization and finding of anomalous activities but, the more the analysis is refined, the more we are going to inspect lower level details that only the source code can express.

A static analysis tool is indeed needed and can aid in various circumstances. Dynamic analysis tools can reveal a high CPU load for some components in the running scenario but the code provides the final explanation for the load, e.g. a *for* loop.

In addition, program comprehension cannot be achieved with the sole use of dynamic analysis tools and tuning cannot be achieved without a clear comprehension of the software inspected.

In this task I have used the reverse engineering tools developed by Claudio Riva and described in [12].

4.6 Using the Different Views Together

The CPU load graph and the message statistics views are the first views to begin the analysis with, then we can freely pass from one view to another.

The processes with the highest CPU load, the messages consuming most of the CPU time and the processes highly coupled are the first to be scrutinized and optimized.

As an example of interaction of the views, we can study from figure 2 the dependencies between the views following the behavior of the processes A and B and message type m_1 at time t_1. Processes A and B consume the most CPU time. At the peak, the MSC (message sequence chart view) reveals the reasons for the load, the m_1 at time t_1. The statistics view provides a more detailed information of the message m_1. The reverse engineering made by the static analysis tools finds the part of the code involved, the part of the source code where optimizations are possible and important. Finally, the coupling between processes A and B shows their

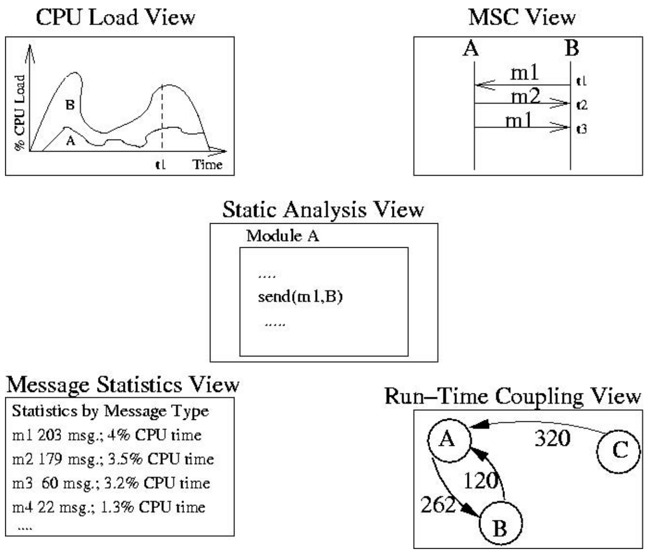

Figure 2: Architectural Views Reconstructed Using the Tools

tight connection and clarifies comprehension of system the run-time.

I discovered an architecture conformance problem during the industrial experience. The anomaly was evident in the message statistics view, where a specific message was consuming a large amount of CPU time. Then, the CPU load view showed the high load of the process involved. The break of the architecture rule was discovered using the coupling view, which showed that that connection was not allowed. The static analysis view permitted the location of the module and the source code.

Sometimes not all the views are used. I discovered a God Class antipattern [14] with the coupling view. A God Class is defined in this context as a large complex process (the God Class) that a large group of simple processes need to access and to communicate with. As a result, the bottleneck degrades the performance.

4.7 Benefits and Optimizations

There are several optimizations permitted by the analysis framework which can span different levels of abstraction. At the architecture design level, performance anti-patterns can be highlighted and the architecture subsequently refactored, [5], [4]. The coupling between the processes shows run-time dependencies and provides feedback for improvements to the architecture's structure. The improvements made with this view concern the architecture evolution and performance. A system highly coupled is a clear sign of bad architecture design, i.e. it is very difficult to add and subtract components; at the same time, performance is negatively affected e.g. God Class antipattern.

However, even an excellent design of the architecture does not assure that the performance objectives are achieved. Implementation algorithms are still very important. Message statistics and MSC might highlight bottlenecks in the handling of messages. Then the static analysis view will find the module and the part of the source code to improve.

A strict cooperation between static analysis and dynamic analysis tools is strongly advised. The dynamic analysis tools reconstruct the run-time view, while the static analysis tools complement them, aiming at a greater program comprehension.

In addition, the dynamic and the static analysis can reveal inconsistencies between the architecture as implemented and the architecture as designed (architecture check and conformance). This statement has also been proved though my industrial experience. With the reconstruction of the views using the performance framework developed I discovered a pattern where the messages were conforming to a previous design decision but not conforming to the new architecture design [2].

The target of optimizations is the heavy hitters. Not much benefit is found in improving an activity that consumes a small amount of resources. In additions, the benefit of the improvement has to be balanced with the amount of work and resources needed to make the change. An improvement that drastically changes the architecture must have a valid reason, i.e. be able to respect the real-time constraints or standards. The job of the performance engineer also consists of the ability to find the trade-offs between performance optimizations and gain.

4.8 Considerations on the Performance Framework

The framework consists of different views. At the beginning of my work, only the CPU load view was used in the development environment. I conceived the other views to satisfy the need for a comprehensive study of the message passing system and the overhead created by its infrastructure. Even if the CPU load view is important for the location of hot spots and heavy hitters, it does not provide the answer to important questions on the root causes of the problem i.e. what it is happening at the specific time in the specific peak load.

The views considered are the most important from my point of view for the performance tuning of the architecture analyzed. The CPU load view and the message statistics view were the first to be used, however, full comprehension of the system could only be achieved using all the views together.

The performance framework does not constitute the definitive answer needed to tune a software architecture based on a message passing mechanism but in my experience it was an important instrument to tackle architecture performance. With this approach, the main bottlenecks can be found, in addition, I was able to extract a scenario and create the views in few hours.

However, to tune a software system, more aspects could be targeted and enhanced narrowing the scope of the tuning with specific analysis. The study of the scheduling algorithms and policies are still very important and Rate Monotonic Analysis is one of the methods used [7]. To improve the memory consumption and reduce the fragmentation, simulations of different allocation algorithms and strategies can be run and analyzed [15].

5. FUTURE WORK

Below is a list of optimizations and suggestions I believe important for the improvement of the current work:

- Visualization tools to better understand the structure of the software are needed. At the moment I am exploring the use of *self-organizing maps* to better visualize the coupling between the components in the system [8].
- Both static and dynamic analysis tools must be used when tuning the architecture. However, more synchronization of the views is needed in order to have a better and clear understanding of the dependencies between the two views. An interesting work of synchronization between message sequence charts and static views can be found in [13].
- The fact that the analysis tools are difficult to port to another environment or system constitute a barrier and a challenge to be faced.
- During the development of the analysis environment I found the importance of the use of scripting languages for prototyping and rapid development. The fact that regular expressions were extensively used also support the use of scripting languages [10]. In this case I used Python [11].
- The performance analysis tools constitute a useful set but new tools and improvements are necessary to automatize the process of performance tuning, to increase the level of details to be extracted and to facilitate the learning curve for novice performance engineers.
- I have tuned the analysis for a message passing framework. An interesting research area would be how to enhance the tools and the analysis for similar systems such as frameworks that use messages and function calls.

6. CONCLUSION

In this paper I have described a performance analysis environment for investigating and tuning the performance of a message passing system. The analysis tools allow the reconstruction of different views of the architecture. The different views can give a better understanding of large software systems and suggest improvements to the architecture structure without excluding code level details. Good results have been obtained in the realm of industrial experience.

The future work will concentrate on creating a better and integrated performance analysis framework with views synchronization and applying the knowledge acquired to different systems.

7. REFERENCES

[1] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2002.

[2] C. Del Rosso. The process of and the lessons learned from performance tuning of a product family software architecture for mobile phones. *Proceedings of the 8th European Conference on Software Maintenance and Reengineering*, Tampere, Finland, March 24-26 2004.

[3] T. Demarco. *Structured Analysis and System Specification*. Prentice-Hall, Yourdon Press Computing Series, 1978.

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

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

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

[7] M. Klein, T. Ralya, B. Pollak, R. Obenza, and M. Gonzales Harbour. *A Practitioners Handbook for Real-Time Analisys*. Kluwer Academic, 1993.

[8] T. Kohonen. *Self-Organizing Maps*. Springer Verlag; 3rd edition, http://www.cis.hut.fi/research/som-research/, 2000.

[9] E. Metz and R. Lencevicius. Efficient instrumentation for performance profiling. *Proceedings of the ICSE Workshop on Dynamic Analysis, (WODA)*, May 3-11 2003.

[10] J. K. Ousterhout. Scripting: Higher level programming for the 21st century. *Computer*, 31(3):23–30, March 1998.

[11] Python. *http://www.python.org*.

[12] C. Riva. Reverse architecting: an industrial experience report. *Proceeding of the 7th Working Conference on Reverse Engineering (WCRE2000)*, (Brisbane, Australia), November 23-25 2000.

[13] C. Riva and J. Vidal, Rodriguez. Combining static and dynamic views for architecture reconstruction. *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*, pages 47–55, March 11-13 2002.

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

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