Information Needs in Performance Analysis of Telecommunication Software – a Case Study

Vesa Hirvisalo
Esko Nuutila

Helsinki University of Technology
Laboratory of Information Processing Science
Otakaari 1, FIN-02150, Espoo, Finland
Vesa.Hirvisalo@hut.fi, Esko.Nuutila@hut.fi

This paper discusses the information and the architectural views that are needed in performance analysis of software. Our discussion is based on a performance analysis that we did. The analyzed system was a subsystem of a large distributed telecommunication system, which has real-time requirements on its performance. During the analysis we found out that the typical software documents, e.g., specifications of software module structure, functionality and interfaces, lack information that is needed in performance analysis. In this paper, we describe what information we had and recovered or produced. Further, we discuss what information should be available to support performance analysis of large software systems.

1 Introduction

In this paper, we discuss the information and the architectural views that are needed in performance analysis of software. The discussion is based on a case study; the performance analysis of a telecommunication system software component. The goal of the study was to find out the response time of a device driver and to locate the bottlenecks in its execution. The analysis was done by modeling the component and measuring its performance by experimenting with a real system.

A software performance analysis by modeling and experimentation involves processing a large amount of analysis data. By analysis data we mean all the information that is needed in a performance analysis. Typically analysis data consists of the system documentation, the source code, performance models, monitoring and measurement data etc.

In our experience, the system documentation should include different views of the system under study. The typical software documents, e.g. specifications of software module structure, functionality and interfaces, lack information that is needed in performance analysis. They tell what the system does and how it was built, but they do not tell how the system operates.

To model and measure the performance of a software, the performance analyst should understand how its execution proceeds and why it is designed to proceed as it does. Without such understanding it is hard to decide, which features of the system operation should be modeled and measured, and which not. Such information can be found, for example, in execution architecture descriptions, control-flow diagrams, data-flow diagrams, and various scheduling diagrams. We call such documentation the *execution description* of a system.

During the performance analysis we found out that navigating in the system documentation is hard. Reading all the documentation and all the code of a large system is too laborious. The performance analyst should have an easy access to the data. For a given piece of code, the corresponding part of the documentation should be easily available, and vice versa.

We modeled the system using execution graphs and queueing networks [4]. We used the experimental measurement data to adjust and validate the performance model. The performance bottlenecks and ways to improve the performance could be easily seen from the final model.

The structure of this paper is the following. In Section 2, we introduce our analysis task case. We describe the analysis method that we used and the corresponding concepts. Section 3 describes the information needs in the performance analysis process step by step. Section 4 discusses the questions raised in Section 3.

2 Analysis task case

The system under study was a part of a distributed telecommunication system, which consists of a large communication control software and a number of various hardware devices. We analyzed the performance of a device driver embedded on a transmission control card.

The goal of the performance study was to find out the response time of the device driver and to locate the bottlenecks in its execution. Our study resulted in detailed information of the driver operation and a performance model of the driver. The model points out performance bottlenecks and can be used to predict performance effects of future improvements. Further, our study implied a modification to the driver, which improved the performance of the driver.

Such results cannot be achieved by using straight-forward methods, like simple structural profiling, which assigns each syntactic construct the time consumed by it. For instance, the improvement to the performance of the driver was not achieved by modifying the bottleneck, which consumed most of the CPU-time, but by modifying a part of the system that feeded data to the bottleneck.

Our method of analysis, adapted from [2, 4], consists of the following phases, which are discussed in more detail in the rest of this section.

- phase 1: overview the analysis task
  - overview the system and the software
  - overview the execution
- phase 2: model the performance
  - build an execution model
  - estimate the resource requirements
  - build a performance model
- phase 3: experiment with the real system
  - design and run the experiments
- phase 4: analysis
  - analyze and interpret data
  - validate and verify

To overview the system and the software, we read the software code and related documentation and consulted the system designers. To understand the system operation, we tried to build *execution descriptions*. An execution description is any formal or informal description that expresses, how the execution of a system proceeds and why the system is designed to execute as it does. Thus, it answers questions like: what typically happens in the execution, what can happen in the execution, why the implementation is as it is. A system can have several execution descriptions. The following is a very short informal execution description, which expresses the performance analyst the ideas and assumptions, which the system designer had:

Example: *All incoming messages handled by the system are received by process A. The incoming messages can be of any type, but type X messages are the most common and most critical with respect to the overall performance of the system. Therefore, the system is tuned to handle type X messages. A type X message is processed in the following way. Process A parses the incoming message and sends the parsed data to process B, which accesses the database and constructs the outgoing message. The system scheduling does not allow process A to be interrupted, but process B can be interrupted by processes A and C. However, process B is typically able to construct the outgoing message before A receives the next one.*

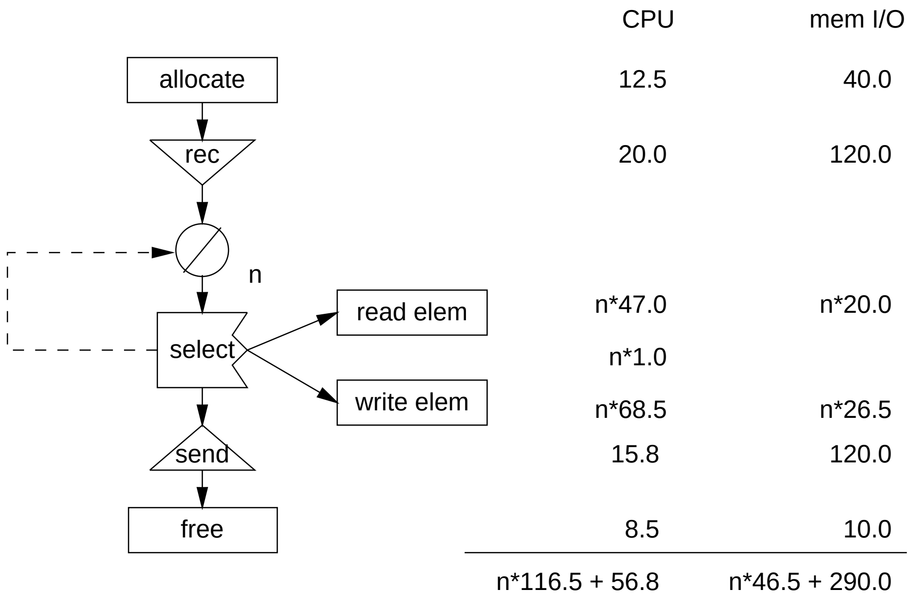

Figure 1: Example of an execution graph.

We modeled the system performance by using *execution graphs* and *queueing networks*. An execution graph [1] consists of vertices, edges, and *resource usage estimates* (typically time). An execution graph identifies the software components that execute, the order of execution, component repetition and conditional execution.

Figure 1 gives an example of an execution graph. It is a model of a message handler that first allocates memory, then receives a message, processes the fields of the message, sends a message, and frees the allocated memory. For each component of execution the CPU and I/O resource usage estimates are given.

A queueing network [3] consists of a set of servers and flow of jobs between the servers (see Figure 2). The servers can have a queue of incoming jobs.

Using the initial model of the system we designed and run the experiments. We used both hardware and software monitors. The hardware monitors measured the external and internal communication of the system. The software monitor printed data on the software execution. It consisted of a user interface and a number of small fragments of instrumentation code embedded in the transmission hardware

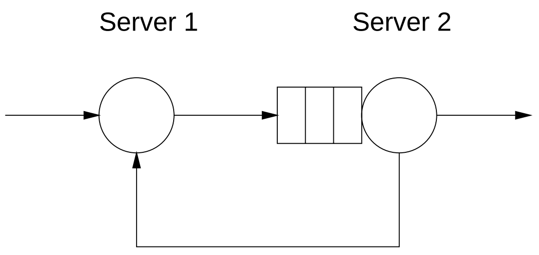

Figure 2: Example of an queueing network.

driver. The metric data emitted from the software monitor was stored for analysis.

3 Information needed

In this section, we describe the information content of the performance analysis process. Our analysis consisted of four major phases: overview phase, modeling phase, experimenting phase, and analysis phase. We shall discuss the information content of each major phase in its own subsection.

3.1 Phase 1: Overview

During the overview phase, the major question that we tried to answer was the following: In what way we should model the system, i.e., how should we decompose it into parts and what kind of a structure should we build. We answered the question by finding out what kind of structures existed and were documented.

We had an overall description of the system and for each module in the system, we had the following documentation available:

- an interface specification describing the interface of the module
- a behavioral specification describing the internal operation of the module
- a list of requirements that the module has to fulfill

We tried to understand how the system was decomposed into modules and what kind of coding conventions were used. Learning the overall software structure and coding conventions was easy. Understanding the interconnections between the software modules and the overall system functionality was very hard, i.e., we could understand how something was implemented, but we did not know why.

Next we tried to get execution descriptions. This information was not readily available; so we tried to recover it, but we did not properly succeed. It seemed that the various restrictions in the system, e.g., priorities of different tasks, implied that the ways how the execution can proceed are restricted. Because of the complexity of the system, we could not state the restrictions. It seems that the execution descriptions are very hard to recover once they have been lost.

Based on the different views of the system that we had, we decided to build a two-level model. The top level was the process structure and the second level was a functional decomposition. We could also have used the modular or calling structure, but they seemed inferior with respect to the goal of the analysis. Using other views of the system would have been too laborious, because they were not readily available.

For a discussion of architectural views that can be typically found in industrial software see, for example, [5].

3.2 Phase 2: Modeling

The major question that we tried to answer during the modeling phase was the following: which tasks on the flow of control have significant effect on the performance of the system. Thus we tried to decompose the execution into reasonable-size parts that included everything that seemed relevant and excluded everything that was obviously irrelevant with respect to the performance.

We started modeling the system by building execution graphs describing its operation. Our work was based on the source code, the input data, the output data, and international standards. We created the execution graphs by reading the code and annotating it. The modeling of the system was based on the understanding how the system operates and why it operates as it does.

We did not have proper execution descriptions. We could understand how the execution typically proceeds, but we could not understand what are the options and why the system executes as it does. The reasons for the selected design tell the performance analyst what are the central tasks. They guide the performance analyst to decide, which concrete tasks in an implementation should be merged to form an abstract task in its model. They also guide to select the level of detail needed in various parts of the model.

3.3 Phase 3: Experimenting

The major task in the experimenting phase was designing the experiments. Running the experiments was rather easy and actually involved no thinking. We followed our plan and saved the resulting measurement data. During this phase, we faced no problems related to the information available.

3.4 Phase 4: Analysis

After experimenting, we analyzed our data and wrote the final document of our performance analysis. The question that we tried to answer during this phase was the following: Does our model describe the performance of the system under study.

The work that we did was basically simple; we analyzed our measured data statistically, added the resulting values to the performance model, and modified the

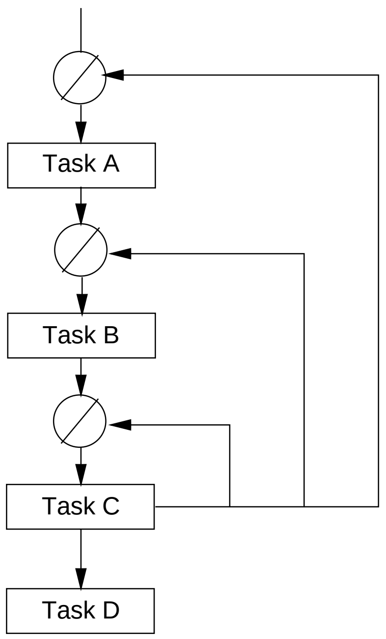

Figure 3: A part of the initial model.

model accordingly. The result was a verified performance model that pointed out the performance bottlenecks.

The modifications that we made to our original model were significant. Thus, our original understanding of the execution, which was based on the system documentation, lacked many facts. We illustrate this by an example. In Figure 3 is a part of our original model. We believed that this part is a performance bottleneck, but it was not. From the system documentation we did not understand that the outer loops are executed only once in the situation that we studied. Further, the tasks A and B have practically no effect on the overall performance. In our final model, the part was reduced to the form of Figure 4.

After the experiments we had about 10Mb of analysis data containing different versions of source code (with various versions of instrumentation), different versions of execution models, measurement data from different measurement sessions and

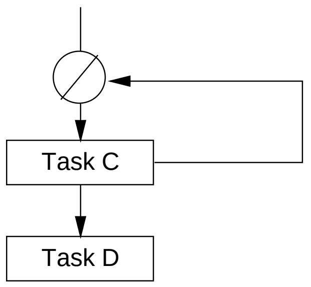

Figure 4: The corresponding part of the final model.

monitors, etc. Although all data was carefully placed into files, we had major difficulties in organizing the data and understanding all the dependencies.

To check if a detail was relevant or not, we had to find the corresponding data. Typically, this meant finding a piece of source code, a part of the system documentation, an instrumentation that we had done, and the resulting metric values that corresponded to a part of our performance model.

4 Discussion

When we had finished the analysis task, we evaluated the analysis work itself. We tried to understand what could be learned and how doing such analysis could be improved. In the following, we first discuss the need for different views and especially for execution descriptions. Then, we discuss the data access and management problems.

In our experience, the analysis data should include different views of the system under study [6]. The typical software documents, e.g., specifications of software module structure, functionality and interfaces, lack information that is needed in performance analysis. They tell what the system does and how it was build, but they do not tell how the system operates.

Example: *A subsystem can be modeled as a thread of control in an execution graph or as a node in a queueing network. Having multiple architectural views of the system guides to select the best modeling mechanism. A single view easily leads to a biased modeling decision.*

To model and measure the performance of a software, the performance analyst should understand how its execution proceeds. Without such understanding it is hard to decide, which features of the system operation should and which should not be modeled and measured.

Example: *The performance analyst can model a group of three subroutines as three vertices in an execution graph and have simple resource requirements for them. Another way to model the subroutines is to have a single vertex and a complex resource requirement for it. The knowledge of reasons for the current implementation guides in deciding how to model the subroutines.*

During the performance analysis we found out that navigating in the system documentation is hard. Reading all the documentation and all the code of a large system is too laborious. The performance analyst should have an easy access to the data.

We spend a lot of time while trying to find the documentation of particular pieces of code. Understanding the code was usually easy after finding relevant documentation or standard. Also the reverse access should be fast: after reading a document, the analyst should easily find the corresponding piece of code.

Example: *The code has a statement* `size = 128`*, which causes the messages always to have 128 elements contained. You wonder why, because a smaller size seems more appropriate. Should you model how different message sizes affect the performance or not? After reading hundreds of pages of documentation you find out that the size is an international standard.*

It is quite obvious that these interconnection should be explicitly present, e.g., hypertext-like links should point to the corresponding parts of system documentation and selected parts of international standards.

Significant part of our work consisted of managing the data that we had gathered, recovered, or produced. This was especially true for the analysis phase, where we consumed a lot of time trying to find and organize the data. Although most of the system remained unchanged during the study, the data evolved as the analysis proceeded. Maintaining the various dependencies in the data made the analysis task significantly harder to do.

We feel that specialized data management tools and software documentation containing multiple views is needed. Particularly, the documentation of any large software having significant performance requirements should include the views that describe its execution. Having such documentation and tools does not only help performance analysis; they are useful through the various stages of the life of a software.

Acknowledgements

We thank Alexander Ran of Nokia Research Center for his valuable comments and guidance during the preparation of this paper.

References

[1] T.L. Booth. Use of computation structure models to measure computation performance. In *Conference on Simulation, Measurement and Modeling of Computer Systems*, Boulder, CO, August 1979.

[2] R. Jain. *The Art of Computer Systems Performace Analysis: Technique for experimental design, measurement, simulation and modeling.* John Wiley and Sons, Inc, Littleton, Massachusetts, 1991.

[3] D.A. Menasce, Almeida A.F., and Dowdy L.W. *Capacity Planning and Performance Modeling.* Prentice Hall, New Jersey, 1994.

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

[5] D. Soni, R.L. Nord, and C. Hofmeister. Software architecture in industrial applications. In *ICSE'95*, pages 196–207, 1995.

[6] C.M. Woodside. A three-view model for performance engineering of concurrent software. *IEEE Transactions on Software Engineering*, 21(9):754–766, September 1996.