Chapter 16

SOFTWARE PERFORMANCE ENGINEERING

Connie U. Smith[1] and Lloyd G. Williams[2]
[1]Performance Engineering Services, PO Box 2640, Santa Fe, NM 87504, www.perfeng.com
[2]Software Engineering Research, 264 Ridgeview Lane, Boulder, CO 80302, (303) 938-9847

Abstract: Performance is critical to the success of today's software systems. However, many software products fail to meet their performance objectives when they are initially constructed. Fixing these problems is costly and causes schedule delays, cost overruns, lost productivity, damaged customer relations, missed market windows, lost revenues, and a host of other difficulties. This chapter presents *software performance engineering* (SPE), a systematic, quantitative approach to constructing software systems that meet performanced objectives. SPE begins early in the software development process to model the performance of the proposed architecture and high-level design. The models help to identify potential performance problems when they can be fixed quickly and economically.

Key words: performance, software performance engineering, architecture

1. INTRODUCTION

While the functionality delivered by a software application is obviously important, it is not the only concern. Over its lifetime, the cost of a software product is determined more by how well it achieves its objectives for quality attributes such as performance, reliability/availability or maintainability than by its functionality.

This chapter focuses on developing software systems that meet performance objectives. Performance is the degree to which a software system or component meets its objectives for timeliness. Thus, performance is any characteristic of a software product that you could, in principle, measure by sitting at the computer with a stopwatch in your hand.

There are two important dimensions to software performance: *responsiveness* and *scalability*. *Responsiveness* is the ability of a system to meet its objectives for response time or throughput. In real-time systems, responsiveness is a measure of how fast the system responds to an event, or the number of events that can be processed in a given time. *Scalability* is the ability of a system to continue to meet its response time or throughput objectives as the demand for the software functions increases. Scalability is an increasingly important aspect of today's software systems.

Performance is critical to the success of today's software systems. However, many software products fail to meet their performance objectives when they are initially constructed. Fixing these problems is costly and causes schedule delays, cost overruns, lost productivity, damaged customer relations, missed market windows, lost revenues, and a host of other difficulties. In extreme cases, it may not be possible to fix performance problems without extensive redesign and re-implementation. In those cases, the project either becomes an infinite sink for time and money, or it is, mercifully, canceled.

Performance cannot be retrofitted; it must be designed into software from the beginning. The "make it run, make it run right, make it run fast" approach is dangerous. Recent interest in software architectures has underscored the importance of architecture in achieving software quality objectives, including performance. While decisions made at every phase of the development process are important, architectural decisions have the greatest impact on quality attributes such as modifiability, reusability, reliability, and performance. As Clements and Northrup note [Clements and Northrup 1996]:

> "Whether or not a system will be able to exhibit its desired (or required) quality attributes is largely determined by the time the architecture is chosen."

Our experience is that performance problems are most often due to inappropriate architectural choices rather than inefficient coding. By the time the architecture is fixed, it may be too late to achieve adequate performance by tuning. While a good architecture cannot guarantee attainment of performance objectives, a poor architecture can prevent their achievement.

This chapter presents an overview of *software performance engineering* (SPE), a systematic, quantitative approach to constructing software systems that meet performanced objectives. SPE prescribes principles for creating responsive software, the data required for evaluation, procedures for obtaining performance specifications, and guidelines for the types of evaluation to be conducted at each development stage. It incorporates models for representing and predicting performance as well as a set of analysis methods [Smith 1990]. Use of the UML for deriving SPE models is discussed in [Smith and Williams 2002].

Because of the importance of architecture in determining performance, SPE takes an architectural perspective. The principles and techniques of SPE form the basis for PASA^{SM}, a method for performance assessment of software architectures [Williams and Smith 2002]. PASA was developed from experience in conducting performance assessments of multiple software architectures in several application domains including web-based systems, financial applications, and real-time systems. It uses the principles and techniques described in this chapter to determine whether an architecture is capable of supporting its performance objectives. When a problem is found, PASA also identifies strategies for reducing or eliminating those risks.

The PASA process consists of ten steps [Williams and Smith 2002]. They are based on the SPE modeling process described below. The method may be applied to new development to uncover potential problems when they are easier and less expensive to fix. It may also be used when upgrading legacy systems to decide whether to continue to commit resources to the current architecture or migrate to a new one. And it may be used on existing systems with poor performance that require speedy correction.

The next section describes the SPE model-based approach. The SPE modeling process is then illustrated with a case study.

2. OVERVIEW OF SOFTWARE PERFORMANCE ENGINEERING

SPE is a model-based approach that uses deliberately simple models of software processing with the goal of using the simplest possible model that identifies problems with the system architecture, design, or implementation plans. These models are easily constructed and solved to provide feedback on whether the proposed software is likely to meet performance goals. As the software process proceeds, the models are refined to more closely represent the performance of the emerging software.

The precision of the model results depends on the quality of the estimates of resource requirements. Because these are difficult to estimate for software architectures, SPE uses adaptive strategies, such as upper- and lower-bounds estimates and best- and worst-case analysis to manage uncertainty. For example, when there is high uncertainty about resource requirements, analysts use estimates of the upper and lower bounds of these quantities. Using these estimates, analysts produce predictions of the best-case and

^{SM} PASA and Performance Assessment of Software Architectures Method are service marks of Software Engineering Research and Performance Engineering Services.

worst-case performance. If the predicted best-case performance is unsatisfactory, they seek feasible alternatives. If the worst-case prediction is satisfactory, they proceed to the next step of the development process. If the results are somewhere in between, analyses identify critical components whose resource estimates have the greatest effect and focus on obtaining more precise data for them. A variety of techniques can provide more precision, including: further refining the architecture and constructing more detailed models or constructing performance prototypes and measuring resource requirements for key components.

Two types of models provide information for architecture assessment: the *software execution model* and the *system execution model.* The software execution model is derived from UML models of the software It represents key aspects of the software execution behavior. It is constructed using execution graphs [Smith and Williams 2002] to represent workload scenarios. Nodes represent functional components of the software; arcs represent control flow. The graphs are hierarchical with the lowest level containing complete information on estimated resource requirements.

Solving the software execution model provides a static analysis of the mean, best- and worst-case response times. It characterizes the resource requirements of the proposed software alone, in the absence of other workloads, multiple users or delays due to contention for resources. If the predicted performance in the absence of these additional performance-determining factors is unsatisfactory, then there is no need to construct more sophisticated models. Software execution models are generally sufficient to identify performance problems due to poor architectural decisions.

If the software execution model indicates that there are no problems, analysts proceed to construct and solve the system execution model. This model is a dynamic model that characterizes the software performance in the presence of factors, such as other workloads or multiple users, which could cause contention for resources. The results obtained by solving the software execution model provide input parameters for the system execution model. Solving the system execution model provides the following additional information:

– refinement and clarification of the performance requirements
– more precise metrics that account for resource contention
– sensitivity of performance metrics to variations in workload composition
– identification of bottleneck resources
– comparative data on options for improving performance via: workload changes, software changes, hardware upgrades, and various combinations of each
– scalability of the architecture and design: the effect of future growth on performance

- identification of critical parts of the design
- assistance in designing performance tests
- effect of new software on service level objectives of other systems

The system execution model represents the key computer resources as a network of queues. Queues represent components of the environment that provide some processing service, such as processors or network elements. Environment specifications provide device parameters (such as CPU size and processing speed). Workload parameters and service requests for the proposed software come from the resource requirements computed by solving the software execution model. The results of solving the system execution model identify potential bottleneck devices and correlate system execution model results with software components.

If the model results indicate that the performance is likely to be satisfactory, developers proceed. If not, the model results provide a quantitative basis for reviewing the proposed architecture and evaluating alternatives. Feasible alternatives can be evaluated based on their cost-effectiveness. If no feasible, cost-effective alternative exists, performance goals may need to be revised to reflect this reality.

3. THE SPE MODELING PROCESS

The SPE modeling process focuses on the system's use cases and the scenarios that describe them. In a use-case-driven process such as the Unified Process ([Kruchten 1999], [Jacobson, et al. 1999]), use cases are defined as part of requirements definition (or earlier) and are refined throughout the design process. From a development perspective, use cases and their scenarios provide a means of understanding and documenting the system's requirements, architecture, and design. From a performance perspective, use cases allow you to identify the *workloads* that are significant from a performance point of view, that is, the collections of requests made by the users of the system. The scenarios allow you to derive the processing steps involved in each workload.

The SPE process includes the following steps. The activity diagram in Figure 16-1 captures the overall process.

1. *Assess performance risk*: Assessing the performance risk at the outset of the project tells you how much effort to put into SPE activities. If the project is similar to others that you have built before, is not critical to your mission or economic survival, and has minimal computer and network usage, then the SPE effort can be minimal. If not, then a more significant SPE effort is needed.

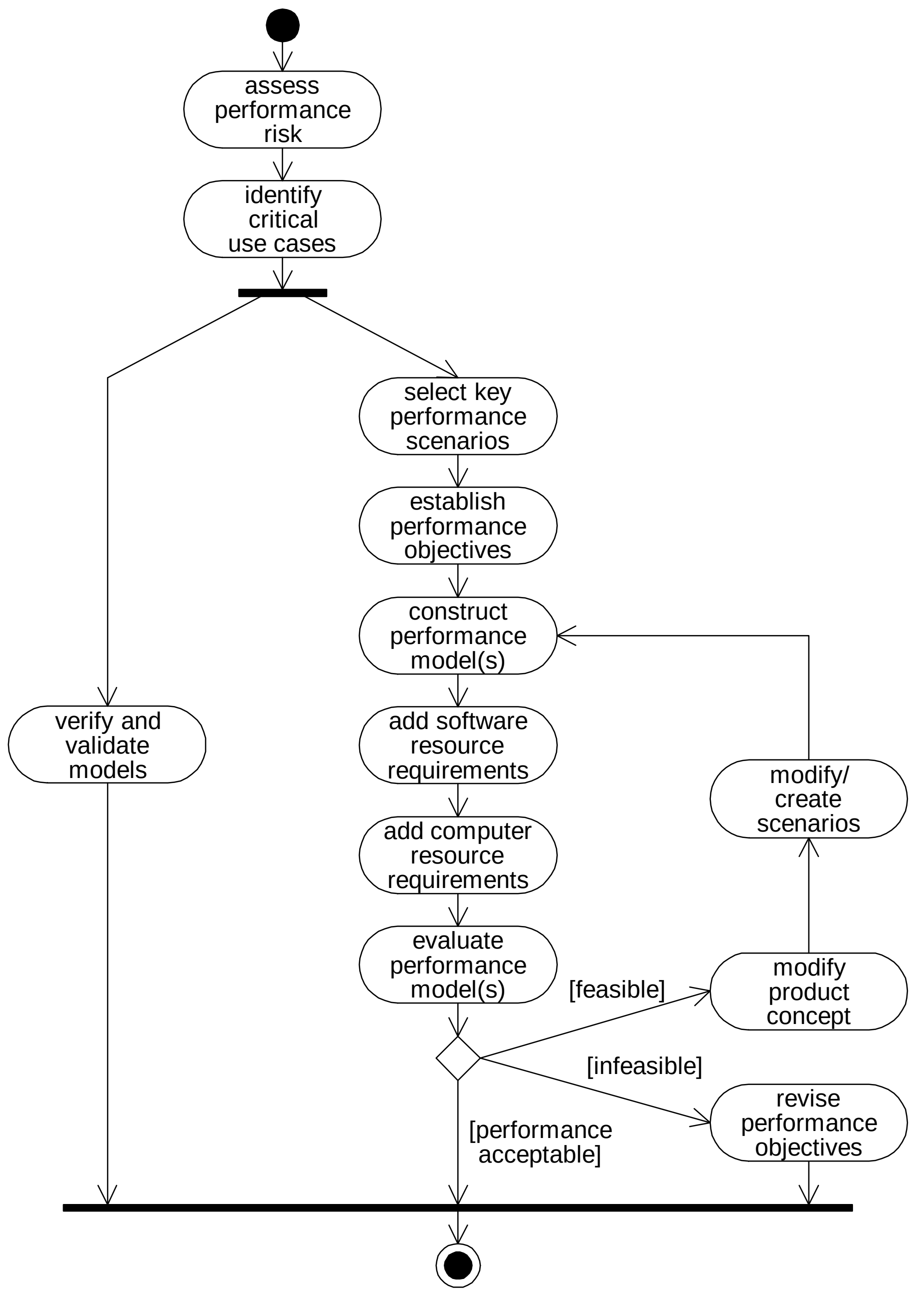
assess performance risk
identify critical use cases
select key performance scenarios
establish performance objectives
construct performance model(s)
verify and validate models
add software resource requirements
modify/ create scenarios
add computer resource requirements
evaluate performance model(s)
[feasible]
modify product concept
[infeasible]
revise performance objectives
[performance acceptable]

Figure 16-1. The SPE Modeling Process

2. Identify *critical use cases*: The critical use cases are those that are important to the operation of the system, or that are important to responsiveness as seen by the user. The selection of critical use cases is also risk driven. You look for use cases where there is a risk that, if performance goals are not met, the system will fail or be less than successful.

   Typically, the critical use cases are only a subset of the use cases that are identified during object-oriented analysis. In the UML, use cases are represented by use case diagrams

3. *Select key performance scenarios*: It is unlikely that all of the scenarios for each critical use case will be important from a performance perspective. For each critical use case, the key performance scenarios are those that are executed frequently, or those that are critical to the perceived performance of the system. Each performance scenario corresponds to a workload. We represent scenarios by using sequence diagrams augmented with some useful extensions.

4. *Establish performance objectives*: You should identify and define *performance objectives* and *workload intensities* for each scenario selected in step 2. Performance objectives specify the quantitative criteria for evaluating the performance characteristics of the system under development. These objectives may be expressed in three primary ways by response time, throughput, or constraints on resource usage. For information systems, response time is typically described from a user perspective, that is, the number of seconds required to respond to a user request. For real-time systems, response time is the amount of time required to respond to a given external event. Throughput requirements are specified as the number of transactions or events to be processed per unit of time.

   Workload intensities specify the level of usage for the scenario. They are specified as an arrival rate (e.g., number of sensor readings per second) or number of concurrent users.

Repeat steps 5 through 8 until there are no outstanding performance problems.

5. *Construct performance models*: We use execution graphs to represent software processing steps in the performance model. The sequence-diagram representations of the key performance scenarios are translated to execution graphs.

6. *Determine software resource requirements:* The processing steps in an execution graph are typically described in terms of the software resources that they use. Software resource requirements capture computational needs that are meaningful from a software perspective. For example, we might specify the number of messages sent or the number of database accesses required in a processing step.

You base estimates of the amount of processing required for each step in the execution graph on the operation specifications for each object involved. This information is part of the class definition in the class diagram. When done early in the development process, these may be simple best- and worst-case estimates. Later, as each class is elaborated, the estimates become more precise.

7. *Add computer resource requirements:* Computer resource requirements map the software resource requirements from step 6 onto the amount of service they require from key devices in the execution environment. Computer resource requirements depend on the environment in which the software executes. Information about the environment is obtained from the UML deployment diagram and other documentation. An example of a computer resource requirement would be the number of CPU instructions and disk I/Os required for a database access.

Steps 6 and 7 could be combined, and the amount of service required from key devices estimated directly from the operation specifications for the steps in the scenario. However, this is more difficult than estimating software resources in software-oriented terms and then mapping them onto the execution environment. In addition, this separation makes it easier to explore different execution environments in "what if" studies.

8. *Evaluate the models:* Solving the execution graph characterizes the resource requirements of the proposed software alone. If this solution indicates that there are no problems, you can proceed to solve the system execution model. This characterizes the software's performance in the presence of factors that could cause contention for resources, such as other workloads or multiple users.

If the model solution indicates that there are problems, there are two alternatives:

– *Modify the product concept*: Modifying the product concept involves looking for feasible, cost-effective alternatives for satisfying this use case instance. If one is found, we modify the scenario(s) or create new ones and solve the model again to evaluate the effect of the changes on performance.

– *Revise performance objectives*: If no feasible, cost-effective alternative exists, and then we modify the performance goals to reflect this new reality.

It may seem unfair to revise the performance objectives if you can't meet them (if you can't hit the target, redefine the target). It is not wrong if you do it at the outset of the project. Then all of the stakeholders in the system can decide if the new goals are acceptable. On the other hand, if you get to the end of the project, find that you didn't meet your goals, and *then* revise the objectives—*that's* wrong.

9. *Verify and validate the models*: Model verification and validation are ongoing activities that proceed in parallel with the construction and evaluation of the models. Model verification is aimed at determining whether the model predictions are an accurate reflection of the software's performance. It answers the question, "Are we building the model right?" For example, are the resource requirements that we have estimated reasonable?

   Model validation is concerned with determining whether the model accurately reflects the execution characteristics of the software. It answers the question [Boehm 1984], "Are we building the right model?" We want to ensure that the model faithfully represents the evolving system. Any model will only contain what we think to include. Therefore, it is particularly important to detect any model omissions as soon as possible.

   Both verification and validation require measurement. In cases where performance is critical, it may be necessary to identify critical components, implement or prototype them early in the development process, and measure their performance characteristics. The model solutions help identify which components are critical.

These steps describe the SPE process for one phase of the development cycle, and the steps repeat throughout the development process. At each phase, you refine the performance models based on your increased knowledge of details in the design. You may also revise analysis objectives to reflect the concerns that exist for that phase.

4. CASE STUDY

To illustrate the process of modeling and evaluating the performance of a real-time system, we will use an example based on a telephony switch. This is not a hard real-time system, but it does have some important performance objectives that are driven primarily by economic considerations: a telephony system should be able to handle as many calls per hour as possible.

The case study is based on information in [Schwartz 1988] and our own experience. It is not intended to be representative of any existing system. Some aspects of the call processing have been simplified so that we may focus on the basic performance issues and modeling techniques.

4.1 Overview

When a subscriber places a call, the local switch (the one that is connected to the caller's telephone) must perform a number of actions to set up and connect the call, and, when the call is completed, it must be cleared. For simplicity, we'll focus on the simplest type of call, often referred to as POTS (plain, ordinary telephone service). Figure 16-2 schematically illustrates the connection between the calling and called telephones. Note that switches A and B may be connected directly, or they may be connected by a route through the public switched telephone network (PSTN). It is also possible that the calling and called telephones are connected to the same local switch.

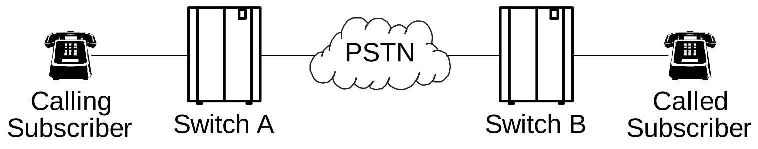

Figure 16-2. Telephony Network

The following sections illustrate the application of the SPE process to this case study.

4.1.1 Assess Performance Risk (Step 1)

Performance is very important, however, the construction of telephony software is well understood. This project will be the first time that this development organization has used object-oriented technology. Thus, because this technology is unfamiliar, we rate the performance risk as high. As a result, the SPE effort will be substantial, perhaps as much as 10% of the overall project budget.

4.1.2 Identify Critical Use Cases (Step 2)

Since we're limiting ourselves to POTS calls, the only use cases are PlaceACall and ReceiveACall. In placing or receiving a call, the local switch interacts directly with only one other switch in the PSTN. There may, however, be several intermediary switches involved between the caller and the receiver of the call.

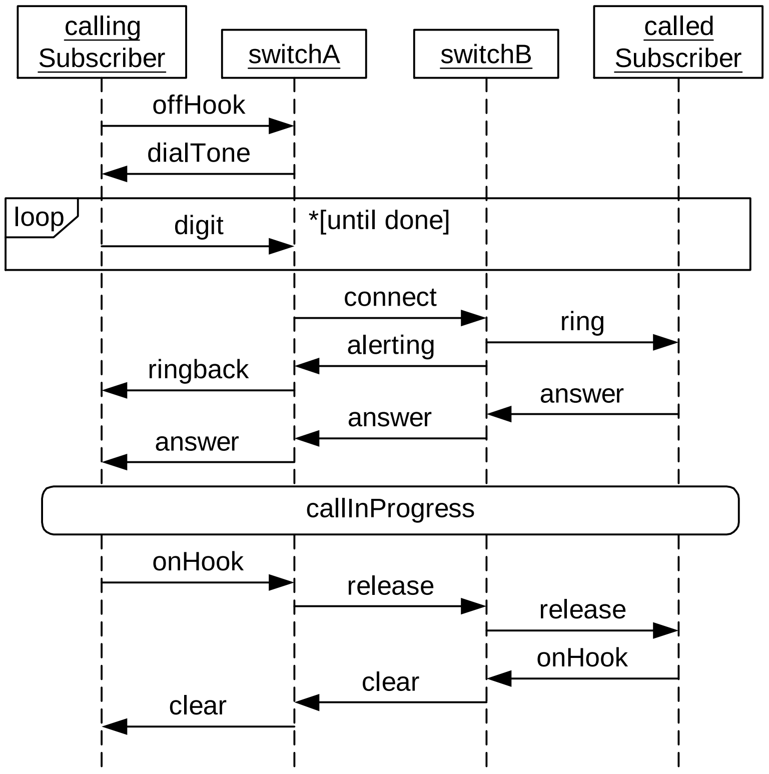

Figure 16-3. Call Sequence Diagram

4.1.3 Select Key Performance Scenarios (Step 3)

Figure 16-3 shows the sequence of events required to complete a call for a customer. For simplicity, we show only two switches in the PSTN. The rectangle labeled loop is an iterator and the rounded rectangle is a reference to another sequence diagram. These extensions to the UML sequence diagram notation are taken from the MSC standard [ITU 1996]. For more details on these extensions, see [Smith and Williams 2002].

The sequence of events is as follows:

1. The caller picks up the telephone handset, generating an `offHook` event to the switch. The switch responds by applying a dial tone to the telephone.
2. The caller then dials a number of `digits`. The number of `digits` dialed may depend on the type of call (local versus long distance).

3. The switch analyzes the dialed digits, determines which outgoing trunk to use, and transmits a connect message to the next switch. This message “seizes” or reserves the connection until the call is completed.
4. The destination switch applies a ring signal to the called telephone, and sends an alerting message back to the originating switch which, in turn, applies a ringback tone to the calling telephone.
5. When the called subscriber answers, the destination switch sends an answer message to the calling switch, which completes the connection to the caller’s telephone. The call then proceeds until one of the parties hangs up.
6. When one of the parties hangs up, an onHook message is transmitted to his/her local switch. That switch then sends a release message to the other switch.
7. When the other party hangs up, a clear message is returned.

Our task is to provide the software that manages the processing of a call for an individual switch.[1]

4.1.4 Establish Performance Objectives (Step 4)

We’ll assume that a performance walkthrough has determined that a module should be able to set up three originating calls per second, and handle their setup within 0.5 second.

4.1.5 Construct Performance Models (Step 5)

To construct the performance models, we need to know the details of the processing that is performed during the execution of the scenario in Figure 16-3. We begin with an overview of the architecture and design of the switch.

A telephony switch serves a number of lines for subscriber telephones and trunks, over which calls can be made to, or arrive from, other switches. To make it possible to easily field switches of different capacities, or upgrade an existing switch to handle more lines, it has been decided that the switch will be composed of a number of module processors. Each module processor serves a fixed number of lines or trunks. To increase the capacity of a particular switch, we simply add more module processors, up to the maximum capacity of the switch.

[1] The sequence diagram in Figure 16-3 shows one scenario from the PlaceACall use case. There are several other scenarios belonging to this use case. For example, some calls receive a busy signal, some are unanswered, sometimes the caller hangs up before the call can be answered, and so on. For this example, we focus on the scenario described by Figure 16-3. Later, we’ll discuss how to include these other possibilities.

When a subscriber places a call, it is handled by the module processor that is connected to the user's telephone (the *calling* module processor). The calling module processor sets up the call and determines a path through the switch to a module processor (the *called* module processor) connected to the required outgoing line or trunk. The outgoing line may be attached to the called party's telephone, or it may be connected via an outgoing trunk to another switch in the PSTN.

Each module also has a line/trunk interface. This interface provides analog-to-digital and digital-to-analog conversion for analog telephone lines, as well as capabilities for communication with other switches via trunks. The line/trunk interface also provides a path for communication with other modules within the same switch.

With this architecture, each call is handled by two module processors within each switch: a calling module processor and a called module processor. Each module needs objects to manage its view of the call.

To accommodate the two views of a call, we use two active classes: an OriginatingCall and a TerminatingCall, as shown in Figure16-4.[2] For each call, there is an instance of OriginatingCall in the calling module processor, and an instance of TerminatingCall in the called module processor. Each instance of OriginatingCall and TerminatingCall has a DigitAnalyzer object to analyze the dialed digits, and a (shared) Path object to establish and maintain the connection. These are passive objects that execute on the thread of

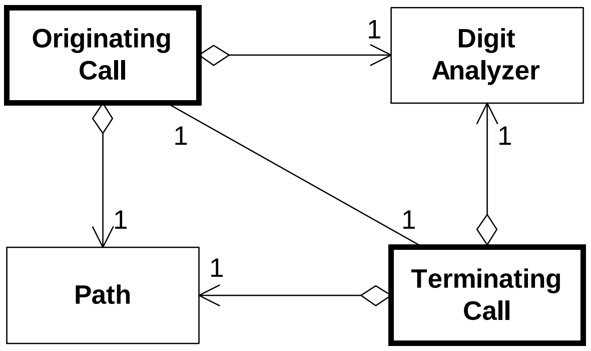

Figure 16-4. Class Diagram

control belonging to their respective call objects

When a subscriber goes "off hook," an OriginatingCall object is created to handle the call. The OriginatingCall object, in turn, creates instances of

2 The active classes are indicated using the usual stereotype of a thick-bordered rectangle.

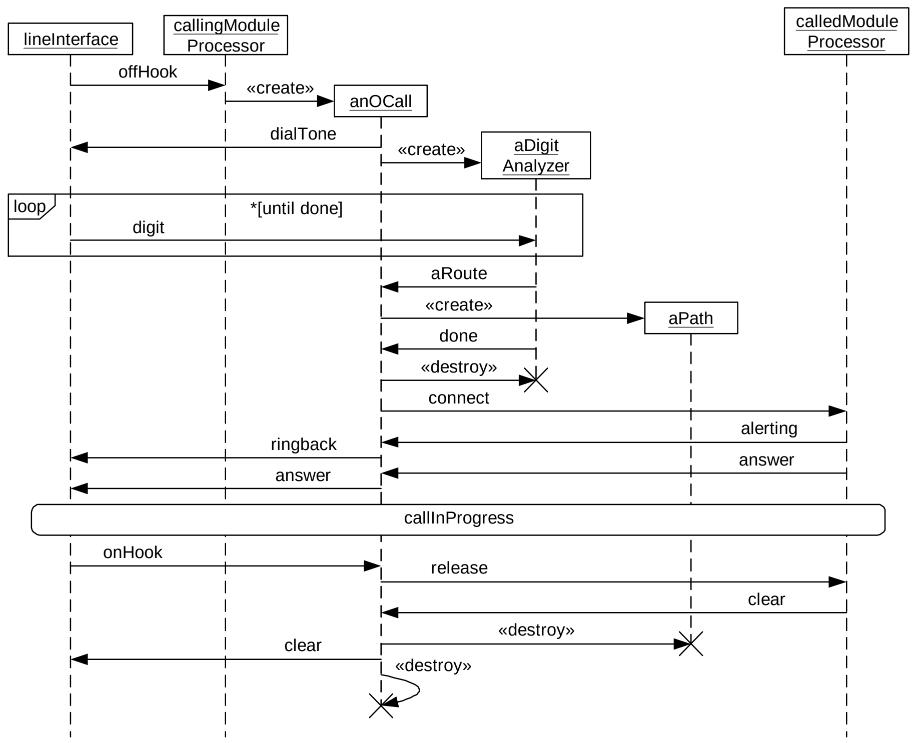

Figure 16-5. Call Origination

`DigitAnalyzer` and `Path` as needed. (The `Path` object maintains the physical connection through the switch—it does not need to interact with other objects.) The refined sequence diagram for call origination is shown in Figure 16-5. There is a similar refined sequence diagram for call termination which is not shown here.

The software model for call origination is straightforward; there is no looping or branching. The only issue in constructing this model is how to handle the time between when the call is connected and when one of the parties hangs up. We could estimate an average call duration, and include it as a delay corresponding to the `callInProgress` step in the sequence diagram. This is awkward, however, and the long delay will hide more interesting aspects of the model solution.

It is much simpler to divide this scenario into two parts: one for initiating the call, and one for ending the call. Thus, we create a separate performance scenario for originating a call and for a hang-up. The hang-up scenario would have the same intensity (arrival rate) as call origination, because every call that is begun must also be ended.

We actually have four performance scenarios for each module processor: call origination (for calls that originate in that module); call termination (for calls that terminate in that module), calling-party hang-up (for when the calling party hangs up first), and called-party hang-up (for when the called party hangs up first).

Figure 16-6 shows the execution graph for call origination. Most of the nodes in this execution graph are expanded nodes that aggregate several steps in the sequence diagram of Figure16-5. This graph shows the major steps in call origination.

Figure 16-7 shows the execution graph corresponding to the calling party ending the call.

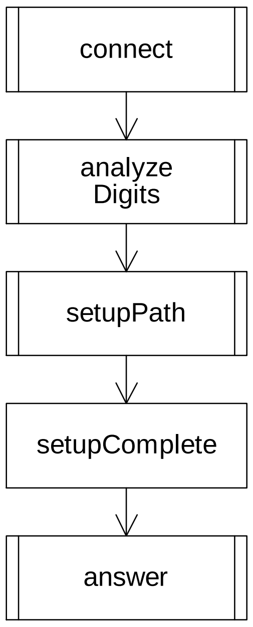

Figure 16-6. Call Origination Execution Graph

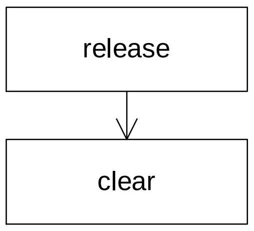

Figure 16-7. Calling-Party Hang-Up Execution Graph

4.1.6 Determine Software Resource Requirements (Step 6)

The types of software resources will differ depending on the type of application and the operating environment. The types of software resources that are important for the telephony switch are:

– CPU—the number CPU instructions (in thousands) executed for each step in the scenario

– Line I/F—the number of visits to the line interface for each step in the scenario

We specify requirements for each of these resources for each processing step in the execution graph. Figure 16-8 and Figure 16-9 show the expansion of the first two nodes in the call origination execution graph of Figure 16-6. The figures also show the resource requirements for each node. Again, these resource requirements are reasonable for this type of system, but are not representative of any particular telephony switch. Details for other nodes are omitted to save space.

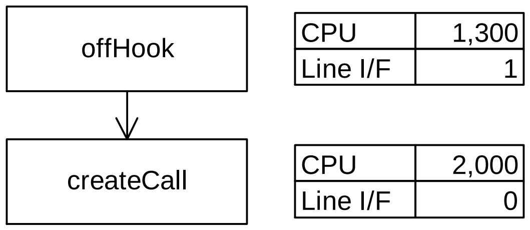

Figure 16-8. Expansion of connect node

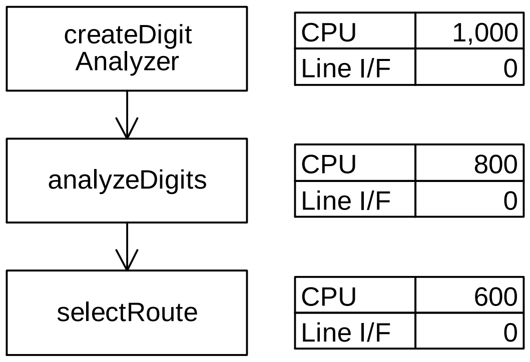

Figure 16-9. Expansion of analyzeDigits node

4.1.7 Add Computer Resource Requirements (Step 7)

We must also specify the *computer resource requirements* for each software resource request. The values specified for computer resource requirements connect the values for software resource requirements to device usage in the target environment. The computer resource requirements also specify characteristics of the operating environment, such as the types of processors/devices, how many of each, their speed, and so on.

Table 16-1 is known as an overhead matrix; it contains the computer resource requirements for the telephony switch. There are two devices of interest: the CPU and the line interface. CPU requirements are specified in units of K instructions.

Table 16-1. Overhead Matrix

Devices	CPU	Line I/F
Quantity	1	1
Service Units	K Instr.	Visits
CPU	1	
Line I/F	100	1
Service Time	.000015	.005

Several processing steps require sending or receiving one or more messages via the line interface. We could explicitly model the sending or receiving of each message. However, that level of detail complicates the model and adds nothing to our understanding of the software's performance. Instead, we include the line interface as overhead, and specify the number of visits to the line interface to send or receive a message for each processing step. Each visit to the line interface requires 100K CPU instructions and a 5 ms. delay to enqueue or dequeue a message and perform the associated processing.

4.1.8 Evaluate the Models (Step 8)

The arrival rate for the call origination scenario is 3 calls per second. For each originating call in one module, there must be a corresponding terminating call in some other module. Thus, on average, each module must also handle three termination calls per second.

To derive intensities for the hang-up scenarios, we'll assume that the probability of the calling party hanging up first is the same as the probability of the called party hanging up first. Then, the arrival rates for these scenarios are the same and, to keep a steady-state rate of calls, they must each also be 3 calls per second. Table 16-2 summarizes the workload intensities for the four scenarios.

Table 16-2. Workload Intensities

Scenario	**Intensity**
Call Origination	3 calls/sec
Call Termination	3 calls/sec
Calling Party Hang-Up	3 calls/sec
Called Party Hang-Up	3 calls/sec

For this example, we focus on the call origination scenario. We'll follow the simple-model strategy and begin with the software execution model. This will tell us whether we can meet the goal of 3 calls per second in the best case—with no contention between scenarios. Figure 16-10 shows the solution for the software execution model (no contention) for this scenario. The overall response time is 0.2205 second. The time required to set up the call is 0.1955 second (0.2205 - 0.0250).[3]

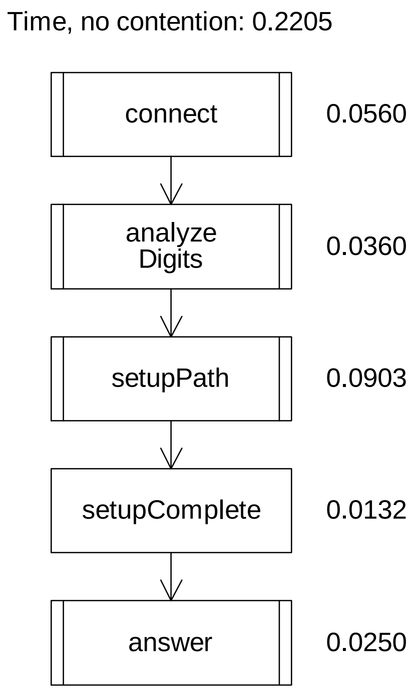

Figure 16-10. Software Execution Model Solution

The response time indicated by the software model is well within our goal of 0.5 second, so we proceed to solve the system model to determine the effects of contention for this one scenario. The system model solution indicates a residence time of 0.3739 second for call setup. This is still within our required time limit. Table 16-3 shows the residence time, time for call setup, and CPU utilization for each of the four scenarios.

[3] The time to set up a call does not correspond directly to the end-to-end time for the execution graph of Figure16-6. Call setup does not include the processing that occurs after the called party has answered (the last node in the graph). Thus, the time to set up the call is taken to be the time from when the user goes "off hook" until the `setupComplete` step is done (the first four nodes in Figure 16-6).

Table 16-3. Contention Results for Individual Scenarios with Object Creation

Scenario	Residence Time	Setup Time	CPU Utilization
Call Origination	0.4156 sec	0.3739 sec	0.53
Call Termination	0.2326 sec	0.2143 sec	0.38
Hang-Up (Called Party)	0.0492 sec		0.12
Hang-Up (Caller)	0.0661 sec		0.14

We now proceed to the next level of complexity in our modeling, constructing and solving the system execution model for all four scenarios. The solution to this model will show the effects of contention among the four scenarios for system resources. The solution indicates a residence time of 16.31 seconds for call setup. This is clearly well over our design goal of 0.5 second.

The reason for this high number can be found by examining the CPU utilization. With all four scenarios executing on the same processor, the CPU utilization is 1.00—the CPU is saturated. In fact, if you add the utilizations for the individual scenarios in Table 16-3, you find that they total more than 1.00!

The formula for residence time at a device is:

$$RT = S/(1 - U)$$

Where S is the service time and U is the utilization.

As the CPU utilization gets closer to 1, the residence time goes to infinity. Our result of more than 16 seconds is an indication that the denominator in this formula is approaching zero.

To meet our design goal, we must reduce the CPU utilization. While no single scenario exceeds the limits, the combined demand on the CPU is enough to put us over the limit.

If we pre-allocate a block of call objects instead of creating them dynamically, we can save this unnecessary overhead. This is an example of “recycling” objects—one of the recommended refactorings of the Excessive Dynamic Allocation antipattern [Smith and Williams 2002]. Each call object is used over and over again, rather than creating a new one for each `offHook` event.

When this change is made, the software model result for call origination becomes 0.1280 second (call setup only), and the contention solution for this scenario is 0.1726 second. Table 16-4 shows the residence time, time for call setup, and CPU utilization for each of the four scenarios without dynamic object creation.

Table 16-4. Contention Results for Individual Scenarios without Object Creation

Scenario	Residence Time	Setup Time	CPU Utilization
Call Origination	0.2048 sec	0.1726 sec	0.32
Call Termination	0.1243 sec	0.1087 sec	0.22
Hangup (Called Party)	0.0224 sec		0.05
Hangup (Caller)	0.0376 sec		0.08

Solving the system execution model with all four revised scenarios shows a residence time for call origination of 0.3143 second, with an overall CPU utilization of 0.68, which is within our performance objective.

We have been following the simple-model strategy [Smith and Williams 2002]; at each step building the simplest possible model that will uncover any problems. We have modeled call processing assuming that all calls are actually completed. As we noted earlier, this is not the actual case. In fact, some calls receive a busy signal, some are unanswered, sometimes the caller hangs-up before the call can be answered, and so on. At this point, we might go back and include these other possibilities. We could then construct scenarios for these additional possibilities, and use either probabilities or arrival rates to reflect the percent of time that they occur.

We'll leave the construction and solution of these additional models as an exercise for the reader.[4]

4.1.9 Verify and Validate the Models (Step 9)

We need to confirm that the performance scenarios that we selected to model are critical to performance, and confirm the correctness of the workload intensity specifications, the software resource specifications, the computer resource specifications, and all other values that are input into the model. We also need to make sure that there are no large processing requirements that are omitted from the model. To do this, we will conduct measurement experiments on the operating environment, prototypes, and analogous or legacy systems early in the modeling process. We will measure evolving code as soon as viable. SPE suggests using early models to identify components critical to performance, and implementing them first. Measuring them and updating the model estimates with measured values increases precision in key areas early.

[4] Motivated readers will find this example in [Smith and Williams 2002]
C. U. Smith and L. G. Williams, *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*, Boston, MA, Addison-Wesley, 2002.. It is possible to obtain the models from www.perfeng.com for further study using the methods explained in the book.

5. SUMMARY

Architectural decisions are among the earliest made in a software development project. They are also among the most costly to fix if, when the software is completed, the architecture is found to be inappropriate for meeting performance objectives. Thus, it is important to be able to assess the impact of architectural decisions on performance at the time that they are made. The SPE process focuses on the system's use cases and the scenarios that describe them. This focus allows you to identify the workloads that are most significant to the software's performance, and to focus your efforts where they will do the most good.

SPE begins early in the software development process to model the performance of the proposed architecture and high-level design. The models help to identify potential performance problems when they can be fixed quickly and economically.

Performance modeling begins with the software execution model. You identify the use cases that are critical from a performance perspective, select the key scenarios for these use cases, and establish performance objectives for each scenario. To construct the software execution model, you translate the sequence diagram representing a key scenario to an execution graph. This establishes the processing flow for the model. Then, you add software and computer resource requirements and solve the model.

If the software execution model solution indicates that there are no performance problems, you can proceed to construct and solve the system model to see if adding the effects of contention reveals any problems. If the software execution model indicates that there are problems, you should deal with these before going any further. If there are feasible, cost-effective alternatives, you can model these to see if they meet the performance goals. If there are no feasible, cost-effective alternatives, you will need to modify your performance objectives, or perhaps reconsider the viability of the project.

To be effective, the SPE steps described in this chapter should be an integral part of the way in which you approach software development. SPE can easily be incorporated into your software process by defining the milestones and deliverables that are appropriate to your organization, the project, and the level of SPE effort required.

The quantitative techniques described in this chapter form the core of the SPE process. SPE is more than models and measurements, however. Other aspects of SPE focus on creating software that has good performance characteristics, as well as on identifying and correcting problems when they arise. They include [Smith and Williams 2002]:

- Applying *performance principles* to create architectures and designs with the appropriate performance characteristics for your application
- Applying *performance patterns* to solve common problems
- Identifying *performance antipatterns* (common performance problems) and refactoring them to improve performance
- Using *late life cycle techniques* to ensure that the implementation meets performance objectives

By applying these techniques, you will be able to cost-effectively build performance into your software and avoid performance failures.

The SPE principles and techniques also form the basis for PASA^{SM}, a method for the performance assessment of software architectures. This method may be applied to new development to uncover potential problems when they are easier and less expensive to fix. It may also be used when upgrading legacy systems to decide whether to continue to commit resources to the current architecture or migrate to a new one. And it may be used on existing systems with poor performance that requires speedy correction. The topics not included here are explained in [Smith and Williams 2002]

6. REFERENCES

[Boehm 1984] B. W. Boehm, "Verifying and Validating Software Requirements and Design Specifications," *IEEE Software*, vol. 1, no. 1, pp. 75-88, 1984.

[Clements and Northrup 1996] P. C. Clements and L. M. Northrup, "Software Architecture: An Executive Overview," Technical Report No. CMU/SEI-96-TR-003, Carnegie Mellon University, Pittsburgh, PA, February, 1996.

[ITU 1996] ITU, "Criteria for the Use and Applicability of Formal Description Techniques, Message Sequence Chart (MSC)," International Telecommunication Union, 1996.

[Jacobson, et al. 1999] I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process*, Reading, MA, Addison-Wesley, 1999.

[Kruchten 1999] P. Kruchten, *The Rational Unified Process: An Introduction*, Reading, MA, Addison-Wesley, 1999.

[Schwartz 1988] M. Schwartz, *Telecommunications Networks: Protocols, Modeling and Analysis*, Reading, MA, Addison-Wesley, 1988.

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

[Smith and Williams 2002] C. U. Smith and L. G. Williams, *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*, Boston, MA, Addison-Wesley, 2002.

[Williams and Smith 2002] L. G. Williams and C. U. Smith, "PASA^{SM}: A Method for the Performance Assessment of Software Architectures," *Proceedings of the Third International Workshop on Software and Performance (WOSP2002)*, Rome, Italy, July, 2002, pp. 179-189.