Chapter 1

Introduction

Knowledge of what is possible is the beginning of happiness.
—George Santayana

In This Chapter:

- Performance failures and their consequences
- Managing performance
- Performance successes
- What is software performance engineering?
- SPE models and modeling strategies

1.1 Software and Performance

This book is about developing software systems that meet performance objectives. Performance is an indicator of how well a software system or component meets its requirements for timeliness. Timeliness is measured in terms of response time or throughput. The *response time* is the time required to respond to a request. It may be the time required for a single transaction, or the end-to-end time for a user task. For example, we may require that an online system provide a result within one-half second after the user presses the "enter" key. For embedded systems, it is the time required to respond to events, or the number of events processed in a time interval. The *throughput* of a system is the number of requests that can be processed in some specified time

interval. For example, a telephony switch may be required to process 100,000 calls per hour.

> *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.

Note: Other definitions of performance include additional characteristics such as footprint or memory usage. In this book, however, we are concerned primarily with issues of timeliness.

There are two important dimensions to software performance timeliness: *responsiveness* and *scalability*.

1.1.1 Responsiveness

Responsiveness is the ability of a system to meet its objectives for response time or throughput. In end-user systems, responsiveness is typically defined from a user perspective. For example, responsiveness might refer to the amount of time it takes to complete a user task, or the number of transactions that can be processed in a given amount of time. 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.

In end-user applications, responsiveness has both an objective and a subjective component. For example, we may require that the end-to-end time for a withdrawal transaction at an ATM be one minute. However, that minute may feel very different to different users. For a user in Santa Fe in the summer, it may seem quite reasonable. To a user in Minneapolis in January, a minute may seem excessively long. Both objective and user-perceived (subjective) responsiveness must be addressed when performance objectives are specified. For example, you can improve the *perceived* responsiveness of a Web application by presenting user-writable fields first. Then, build the rest of the page (e.g., the fancy graphics) while the user is filling in those fields.

Responsiveness is the ability of a system to meet its objectives for response time or throughput.

1.1.2 Scalability

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. The graph in Figure 1-1 illustrates how increasing use of a system affects its response time.

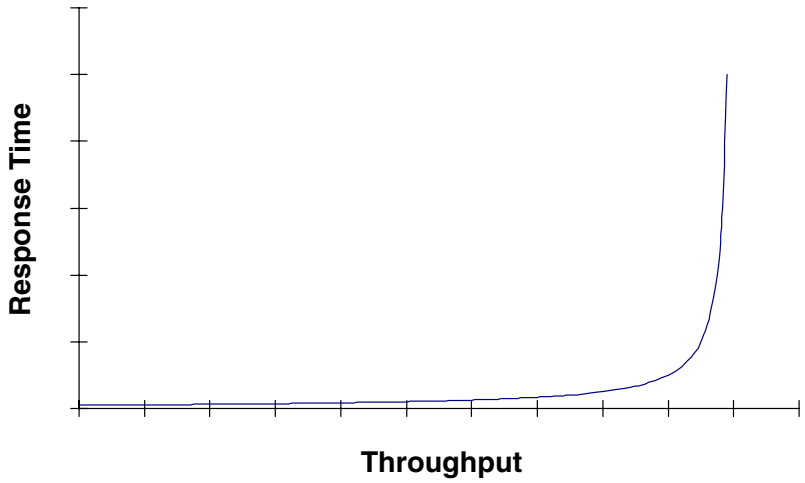

Figure 1-1: Scalability Curve

In Figure 1-1, we've plotted response time against the load on the system, as measured by the number of requests per unit time. As you can see from the curve, as long as you are below a certain threshold, increasing the load does not have a great effect on response time. In this region, the response time increases linearly with the load. At some point, however, a small increase in load begins to have a great effect on response time. In this region (at the right of the curve), the response time increases exponentially with the load. This change from a linear to an exponential increase in response time is usually due to some resource in the system (e.g., the CPU, a disk, the network, sockets, or threads) nearing one hundred percent utilization. This resource is known as the "bottleneck" resource. The region where the curve changes from linear to exponential is known as the "knee" because of its resemblance to a bent knee.

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.

Web applications are discussed in Chapters 5, 7, and 13.

Scalability is an increasingly important aspect of today's software systems. Web applications are a case in point. It is important to maintain the responsiveness of a Web application as more and more users converge on a site. In today's competitive environment, users will go elsewhere rather than endure slow response times.

In order to build scalability into your system, you must know where the "knee" of the scalability curve falls for your hardware/software environment. If the "knee" occurs before your target load requirements, you must either reduce the utilization of the bottleneck resource by streamlining the processing, or add additional hardware (e.g., a faster CPU or an extra disk) to remove the bottleneck.

This book presents an integrated set of solutions that you can use to build responsiveness and scalability into your software systems. These solutions include a combination of modeling, measurement, and other techniques, as well as a systematic process for applying them. They also include principles, patterns, and antipatterns that help you design responsiveness and scalability into your software. These techniques focus primarily on early life cycle phases to maximize your ability to economically build performance into your software. However, we also present solutions for systems that already exhibit performance problems.

1.2 The Importance of Performance

It's fair to ask at the outset: "Why is performance important?" The following anecdotes illustrate the answer:

> NASA was forced to delay the launch of a satellite for at least eight months. The satellite and the Flight Operations Segment (FOS) software running it are a key component of the multibillion-dollar Earth Science Enterprise, an international research effort to study the interdependence of the Earth's ecosystems. The delay was caused because the FOS software had unacceptable response times for developing satellite schedules, and poor performance in analyzing satellite status and telemetry data. There were also problems with the implementation of a control language used to automate operations. The cost of this rework and the resulting delay has not yet been

> determined. Nevertheless it is clearly significant, and the high visibility and bad press is potentially damaging to the overall mission. Members of Congress also questioned NASA's ability to manage the program. [Harreld 1998a], [Harreld 1998b]

And, on a lighter note:

> The lingerie retailer Victoria's Secret used its new Web site to broadcast its spring fashion show over the Internet. To make sure that there would be a large number of viewers, the company announced the show in a 30-second advertisement during the Super Bowl. A total of 1.5 million people logged on to the Web site to view the broadcast, which used concurrent video streams. Despite extensive pre-planning and the addition of more servers and load-balancing software, viewers experienced jerky video and interrupted audio. At least five percent of those trying to view the show were unable to access it. [Trott 1999]

Both of these anecdotes illustrate performance failures—the inability of a software product to meet its overall objectives due to inadequate performance. Additional instances of performance failures appear in Example 1-1.

Example 1-1: Performance Failures

Distributed Order Management System One Fortune 100 company attempted to implement a new distributed order management system that would integrate several legacy systems with new software to track the status of orders and trigger actions in the other systems at the proper time. Performance problems in the initial version prevented its timely deployment. After three significant schedule delays, a comprehensive study of the end-to-end performance of critical use cases identified significant architectural problems that could not be corrected with either tuning or additional hardware. An attempt was made to deploy the system to meet schedule requirements. Users were disgruntled, and did not use features intended to improve order management because of performance problems. The system was ineffective because of a failure to meet its performance requirements.

Accounting System Another large company attempted a reimplementation of its accounting system. The original schedule estimated a two-year completion. After seven years, the system had been reimplemented three times; none of these implementations met the performance objectives. The third attempt used 60 times the CPU time of the original attempt.

Dynamic Reporting with COTS A large bank attempted to avoid risks by using a commercial off-the-shelf (COTS) package that provided most of the functions it required. The interactive portion of the system performed acceptably, but the bank experienced serious problems with a desired dynamic reporting function. The internal database organization of the COTS package was not in the order desired in the reports, and the processing required to produce the "roll-ups" for desired totals was excessive. The bank was forced to create a new reporting function that would run at night, thus losing the benefit of producing the reports at the time they were desired.

Call Processing A re-implementation of a call processing system for a telecommunication switch also did not consider performance early in development. The initial object-oriented design required several hundred times the allotted time to complete a call.

Automated Teller Machine An object-oriented design for teller machines focused on reuse to streamline the customization required for each bank that purchased the machines. Developers were not worried about performance because the hardware speeds were far greater than typically required for user interactions, and they had never had performance problems with previous implementations. The first implementation was unusable due to performance problems, and required substantial re-work to correct problems.

Electronic Trading Several online brokerage houses experienced unusually large numbers of hits on their Web sites following a stock market dip on October 27, 1997. The Web sites could not scale to meet the demand, so customers experienced long delays in using the sites, if they could get in at all. The result was that investors lost hundreds of thousands of dollars. At least one lawsuit has been filed alleging that online capacity was insufficient to meet users' needs.

Performance is an essential quality attribute of every software system. Many software systems, however, both object-oriented and non-object-oriented, cannot be used as they are initially implemented due to performance problems. For example, if the system is an end-user application, it may not respond rapidly enough to user actions, or handle the number of transactions that occur during peak load conditions. Or, if it is an embedded system, it may not respond rapidly enough to an external stimulus, or be able to process events that occur with a high frequency.

1.2.1 Consequences of Performance Failures

As the anecdotes above and in Example 1-1 illustrate, performance failures can have a variety of negative consequences. These include:

- *Damaged customer relations*: Your organization's image suffers because of poor performance. Even if the problem is fixed later, users will continue to associate poor performance with the product.
- *Business failures*: Poor performance means that your staff needs more time to complete key tasks, or that you need more staff to complete these tasks in the same amount of time. This may mean that you are unable to operate on a peak business day, to respond to customer inquiries, or to generate bills or payments in a timely fashion.
- *Lost income*: You lose revenue due to late delivery. In some cases, you may find yourself paying penalties for late delivery or failure to meet performance objectives.
- *Additional project resources*: Project costs rise as additional resources are allocated for "tuning" or redesign.
- *Reduced competitiveness*: "Tuning" or redesign results in late delivery that can mean missed market windows.
- *Project failure*: In some cases it will be impossible to meet performance objectives by tuning, and too expensive to redesign the system late in the process. These projects will be canceled.

1.2.2 Causes of Performance Failures

How and why does this happen? Our experience is that performance problems are most often due to fundamental architecture or design factors rather than inefficient coding. As Clements and Northrup point out:

> 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. [Clements and Northrup 1996]

This means that performance problems are introduced early in the development process. However, most organizations ignore performance until integration testing or later. With pressure to deliver finished software in shorter and shorter times, their attitude is: "Let's get it done. If there is a performance problem, we'll fix it later." Thus, performance

problems are not discovered until late in the development process, when they are more difficult (and more expensive) to fix.

This "fix-it-later" attitude is actually encouraged by many in the object-oriented community. The following quote from Auer and Beck illustrates this misinformation:

Fix-It-Later Attitude

> Ignore efficiency through most of the development cycle. Tune performance once the program is running correctly and the design reflects your best understanding of how the code should be structured. The needed changes will be limited in scope or will illuminate opportunities for better design. [Auer and Beck 1996]

Reliance on "fix-it-later" has its origins in two performance myths.

Performance Myth #1 This myth is based on the assumption that you need something to measure before you can begin to manage performance:

Performance Myth

> *It is not possible to do anything about performance until you have something executing to measure.*

The following quote from Jacobson et al. is typical of this misconception:

> Changes in the system architecture to improve performance should as a rule be postponed until the system is being (partly) built. Experience shows that one frequently makes the wrong guesses, at least in large and complex systems, when it comes to the location of the bottlenecks critical to performance. To make correct assessments regarding necessary performance optimization, in most cases, we need something to measure … [Jacobson et al. 1999]

Performance Reality

The reality is that you don't need to wait to address performance until you have some running code to measure. Performance models can predict performance during the architectural and early design phases of the project. Performance estimation and uncertainty-management techniques can compensate for the lack of precise measurements. The models are sufficient to allow evaluation of architectural or design alternatives. It is not necessary to "guess" the location of bottlenecks or to wait until measurements are available to begin the modeling.

In fact, waiting until there is sufficient code to make detailed measurements can be dangerous. As Clements notes:

> Performance is largely a function of the frequency and nature of intercomponent communication, in addition to the performance characteristics of the components themselves, and hence can be predicted by studying the architecture of a system. [Clements 1996]

As we noted earlier, the architectural decisions, those that have the greatest impact on performance, are made early in the project. Waiting until there is running code to evaluate the performance impact of these decisions means that you don't find problems until much later—when the architectural decisions are more difficult and expensive to change, if they can be changed at all.

Performance Myth #2 This myth is based on the fear that adding performance management to the software process will delay project completion:

Performance Myth

Managing performance takes too much time.

Performance Reality

The reality is that performance management efforts do not automatically require significant amounts of time. The level of effort devoted to performance management depends on the level of risk. If there is little or no risk of a performance failure, then there is no need for an elaborate performance management program. If the risk of performance failure is high, then a higher level of effort is needed. In these cases, however, managing performance from the beginning of the software development process can actually reduce the overall project time by eliminating the need for time-consuming redesign and tuning.

Performance Myth #3 This myth is based on the fear that performance modeling will take too much time and consume too many project resources:

Performance Myth

Performance models are complex and expensive to construct.

Performance Reality

The reality is that simple models can provide the information required to identify performance problems and evaluate alternatives for correcting them. These models are inexpensive to construct and evaluate. It is no longer necessary to build a complex simulation model that is as difficult to write as the software itself. Numerous examples throughout this book illustrate these models and the results they provide. Using simple models in conjunction with the modeling principles discussed later in

this chapter ensures that you get the information that you need to make software architectural and design decisions when you need to, and in a cost-effective manner.

1.2.3 Getting It Right

Managing performance throughout the development process can reduce the risk of performance failure and lead to performance successes such as these:

> An airline reservation service bureau revised its airfare quote system to improve the accuracy of the "lowest fare" quotes. Performance engineers worked closely with developers throughout the project. The result was a system with 100 percent accurate quotes and *improved* performance.
>
> A major insurance company designed a system to provide Web access for its own agents as well as independent agents. The first version of the design called for a large amount of code (in the form of ActiveX agents) to be downloaded to client machines. Performance models of this approach showed that, if the downloaded code underwent a significant upgrade, it would take approximately three days at full bandwidth to download the changes to all of the client machines. The design was changed to rely less on downloaded code, and the system was deployed successfully.

These anecdotes illustrate that managing performance from the initial stages of the project can pay off in systems that meet performance objectives. Additional performance success stories appear in Example 1-2.

Example 1-2: Performance Successes

Event Update Performance engineers conducted a study of a new system early in the requirements analysis phase. Initial requirements called for events to be posted to an online relational database within three minutes of occurrence. The analysts estimated the size of the hardware required to support the requirement (assuming a streamlined software system) to be 20 mainframes!

Three minutes appeared to be a reasonable goal, but the tremendous data volume had dramatic consequences on hardware capacity. The performance engineering analysis allowed a quantitative assessment of the impact of this requirement, and made it possible to redefine the requirements to meet the underlying business goal with a more reasonable hardware configuration.

Distributed Data Access In another study, performance engineers studied the architecture for a new distributed system to provide customers with data about their telecommunication usage. The analysis showed that three of the use cases would meet their performance objectives, but one would require significant configuration upgrades to handle the stated workload intensity.

Developers evaluated trade-offs in the frequency of requests, the hardware and network configuration, and the software architecture and design. They selected a software architecture alternative that handled the required workload without hardware upgrades. It was easy to make the required changes before code was written.

This example shows the power of addressing performance early in development. Without the early performance analysis, the problems would have been discovered much later in the development process. Many of the software alternatives would no longer have been cost-effective, and customer pricing would have been fixed, so configuration upgrades would adversely affect the bottom line. It was easy to prevent these problems with early performance management.

Data Acquisition and Reporting In this case, performance analysis helped prevent a bad situation from becoming worse. An existing system failed to meet performance objectives because it did not scale up to support the number of users specified in the contract. As a result, the organization incurred substantial monetary penalties for failure to meet contract terms. After failing to correct problems through tuning, developers proposed replacing key portions of the system with a new object-oriented subsystem. In the process of constructing performance models of the original system (for model calibration, verification, and validation), two key problems were detected in the process synchronization strategy and in the system's technical architecture. Correcting these problems resolved the contract performance failure and provided time to address scalability in a more systematic manner. The proposed new subsystem would not have met performance requirements; in fact, the performance models predicted worse performance than with the original system!

Airline Reservations An airline reservation system included a component to recover and restore the state of the reservations data after a major outage. The project employed good performance management techniques throughout the development process. While the developers could do partial performance tests on the recovery component, the only way to determine actual success or failure was to experience an outage. It was not possible to run an end-to-end test. The performance models predicted that the system could handle the recovery in the required time, and, the first time that a recovery was needed, the performance goals were met.

Two significant morals emerge from these success stories.

If you intend to rely on hardware to solve performance problems, use performance models early (before other options are closed) to verify that this is a cost-effective solution.

There may be other, more cost-effective solutions than hardware upgrades. As the "distributed data access" anecdote in Example 1-2 shows, there are typically more alternatives early in the process. As more and more architecture and design decisions are made, some options may become prohibitively expensive or simply unavailable.

For its part, the "airline reservations" system story in Example 1-2 shows that

In some cases, end-to-end performance testing is not possible, so performance models are the only option.

1.3 How Should You Manage Performance?

Are you managing software performance reactively or proactively?

1.3.1 Reactive Performance Management

Do any of these sound familiar?

- "Let's just build it and see what it can do."
- "We'll tune it later; we don't have time to worry about performance now."
- "We can't do anything about performance until we have something running to measure."
- "Don't worry. Our software vendor is sure their product will meet our performance goals."
- "Performance? That's what version 2 is for."
- "We'll just buy a bigger processor."
- "Problems? We don't have performance problems."

If so, you're probably managing software performance reactively. Reactive performance management waits for performance problems to appear, and then deals with them in an ad hoc way.

Reactive performance management is just the "fix-it-later" approach in another guise. It has the same risks of cost overruns, missed deadlines, and project failure.

1.3.2 Proactive Performance Management

Proactive performance management anticipates potential performance problems and includes techniques for identifying and responding to those problems early in the process. There are several characteristics of proactive performance management:

- The project has a performance engineer responsible for tracking and communicating performance issues.
- Everyone on the project knows the name of the performance engineer.
- There is a process for identifying performance jeopardy (a danger of not meeting performance objectives) and responding to it.
- Team members are trained in performance processes.
- The project has an appropriate performance risk management plan based on shortfall costs and performance engineering activity costs.

With Software Performance Engineering (SPE), proactive performance management identifies and resolves performance problems early, avoiding the negative effects of the "fix-it-later" approach.

Historically, performance was managed either proactively or by relying on the hardware and support software to resolve problems. In the early days of computing, developers had no choice but to manage the space and time required by their software. Later, innovations in hardware and operating systems provided some relief, and developers began to worry less about performance. "Fix-it-later" was first advocated at this time. Software performance engineering techniques were originally proposed when performance failures due to the reactive "fix-it-later" approach first emerged.

As new technologies appeared, the learning curve for those technologies again required careful management to prevent performance problems. Once the performance aspects of the new technology were understood and the hardware and support software provided new features to accommodate the technology, developers could again rely more on the

hardware and support software. This process repeats as other new technologies are introduced, and the hardware and support software solutions catch up with them.

The use of new software technology requires careful attention to performance until the performance aspects of the new technology are understood.

Note that the "new" technology is not necessarily "bleeding edge" technology—it might just be new to you. If you don't have experience with the technology and the performance intuition that comes with it, the result is the same.

Distributed systems challenge performance intuition. Constructing them involves a complex combination of choices about processing and data location, platform sizes, network configuration, middleware implementation, and so on. Managing the performance of these distributed software systems will likely always call for quantitative assessment of these alternatives.

Intuition about performance problems is not sufficient; quantitative assessments are necessary to assess performance risks.

We have been seeing a phenomenon more and more frequently: as organizations adopt technologies with which they have little experience, such as Web applications, Common Object Request Broker Architecture (CORBA), or Enterprise JavaBeans, they have more performance failures. When using an unfamiliar technology, you are in a situation that requires proactive performance management.

If you are managing performance reactively on systems that use unfamiliar technology, your probability of experiencing a performance failure is much higher.

1.4 Software Performance Engineering

Software performance engineering is a systematic, quantitative approach to constructing software systems that meet performance objectives. SPE is an engineering approach to performance, avoiding the extremes of performance-driven development and "fix-it-later." SPE uses model

predictions to evaluate trade-offs in software functions, hardware size, quality of results, and resource requirements.

The "performance balance" in Figure 1-2 depicts a system that fails to meet performance objectives because its resource requirements exceed computer and network capacity. With SPE, you detect these problems early in development, and use quantitative methods to support cost-benefit analysis of hardware solutions versus software requirements or design solutions, versus a combination of the two. You implement software solutions before problems are manifested in code; organizations implement hardware solutions before testing begins. The quantitative assessment identifies trade-offs in software functions, hardware size, quality of results, and resource requirements.

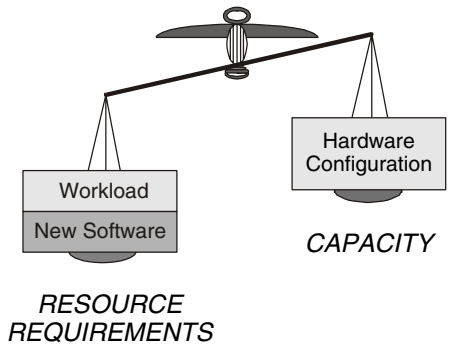

Figure 1-2: Performance Balance

SPE is a software-oriented approach: it focuses on architecture, design, and implementation choices. The models assist developers in controlling resource requirements by enabling them to select architecture and design alternatives with acceptable performance characteristics. The models aid in tracking performance throughout the development process and prevent problems from surfacing late in the life cycle (typically during final testing).

SPE also prescribes principles for creating responsive software, performance patterns and antipatterns for performance-oriented design, 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].

SPE is neither a new nor a revolutionary approach. It applies proven techniques to predict the performance of emerging software and respond to problems while they can be fixed with a minimum of time and expense.

We have found that, with SPE, it is possible to cost-effectively engineer new software systems that meet performance goals. By carefully applying the techniques of SPE throughout the development and integration process, it is possible to produce new systems that have adequate performance and exhibit the other qualities that have made object-oriented techniques so effective, such as reusability, relatively rapid deployment, and usability.

With SPE, you should be able to answer the following questions early in development:

- Will your users be able to complete tasks in the allotted time?
- Are your hardware and network capable of supporting the load?
- What response time is expected for key tasks?
- Will the system scale up to meet your future needs?

1.4.1 SPE Modeling Strategies

The use of these strategies is illustrated in Part II.

SPE uses several modeling strategies to obtain results quickly, to cope with uncertainty in estimates of software and hardware resource usage, and to control costs.

The Simple-Model Strategy The simple-model strategy leverages the SPE effort to provide rapid feedback on the performance of the proposed software.

Start with the simplest possible model that identifies problems with the system architecture, design, or implementation plans.

The early SPE models are easily constructed and solved to provide feedback on whether the proposed software is likely to meet performance goals. These simple models are sufficient to identify problems in the architecture or early design phases of the project. You can easily use

them to evaluate many alternatives because they are easy to construct and evaluate. Later, as more details of the software are known, you can construct and solve more realistic (and complex) models.

The Best- and Worst-Case Strategy The models rely upon estimates of resource requirements for the software execution. The precision of the model results depends on the quality of these estimates. Early in the software process, however, your knowledge of the details of the software is sketchy, and it is difficult to precisely estimate resource requirements. Because of this, SPE uses adaptive strategies, such as the best- and worst-case strategy

Use best- and worst-case estimates of resource requirements to establish bounds on expected performance and manage uncertainty in estimates.

For example, when there is high uncertainty about resource requirements, you use estimates of the upper and lower bounds of these quantities. Using these estimates, you produce predictions of the best-case and worst-case performance. If the predicted best-case performance is unsatisfactory, you look for feasible alternatives. If the worst-case prediction is satisfactory, you proceed to the next step of the development process with confidence. If the results are somewhere in between, the model analyses identify critical components whose resource estimates have the greatest effect, and you can focus on obtaining more precise data for them.

A variety of techniques can also provide more precise resource estimates, including: further refining the design and constructing more detailed models, or constructing a prototype or implementation of key components and measuring resource requirements.

The Adapt-to-Precision Strategy As the simple-model strategy states, simple models are appropriate for early life cycle studies. Later in the development process, SPE uses the adapt-to-precision strategy.

Match the details represented in the models to your knowledge of the software processing details.

As the design and implementation proceed and more details are known, you expand the SPE models to include additional information in areas that are critical to performance.

1.4.2 SPE Models

The SPE models are similar to those used for conventional performance evaluation studies. In conventional studies (of existing systems), capacity planners model systems to predict the effect of workload or configuration changes. The conventional modeling procedure is as follows:

- Study the computer system.
- Construct a system execution model (usually a queueing network model).
- Measure current execution patterns.
- Characterize workloads.
- Develop model input parameters.
- Validate the model by solving it and comparing the model results to observed and measured data for the computer system.
- Calibrate the model until its results match the measurement data.

Planners then use the model to evaluate changes to the computer system by modifying the corresponding workload parameters, the computer system configuration parameters, or both. Capacity planners rely on these models for planning future acquisitions. The model precision is sufficient to predict future configuration requirements; models are widely used, and they work, so we use them as the basis for SPE.

SPE performance modeling is similar. However, because the software does not yet exist, and we do not want to wait for measurements of the software, we first model the software explicitly. Figure 1-3 illustrates the difference between conventional models and SPE models. With conventional models, we are concerned with modeling workloads that already exist. The system execution model quantifies the effects of contention for computer resources by different types of existing work. With SPE, we want to include software that does not yet exist. To do this, we introduce a new type of model: the software execution model. The software execution model represents key facets of the envisioned software execution behavior. Its solution yields workload parameters for the system execution model that closely resemble the conventional models.

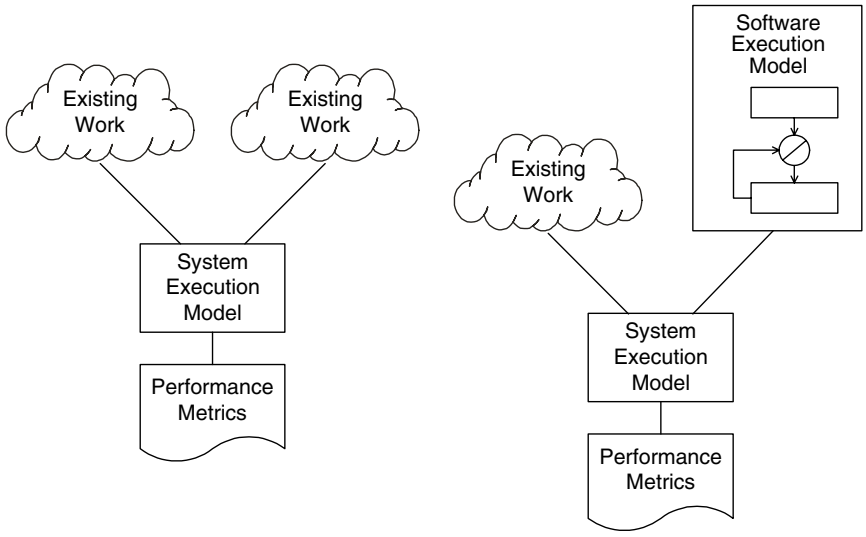

Figure 1-3: Conventional versus SPE Models

1.5 SPE for Object-Oriented Systems

The bad news is that object-oriented (OO) systems present special problems for SPE. The functionality of object-oriented systems is decentralized. Performing a given function is likely to require collaboration among many different objects from several classes. These interactions can be numerous and complex and are often obscured by polymorphism, making them difficult to trace. Distributing objects over a network compounds the problem.

The good news is that object-oriented modeling notations (such as the Unified Modeling Language [UML]) and methods actually help to reduce the impact of these problems. Much of the information needed to perform SPE can be captured as part of the object-oriented analysis and design process with a minimum of disruption. Use cases, which are identified as part of the requirements definition, are a natural link between software development activities and SPE. The scenarios that describe the use cases provide a starting point for constructing the performance models.

Our approach to SPE is tightly integrated with object-oriented notations, such as the UML. It does not require that you become an expert in performance modeling. To do SPE, you construct analysis and design models as you have always done. When you want to evaluate the emerging software's performance, you use your object-oriented analysis or design models to derive a performance model. Solving the model gives you feedback on the performance and suggests ways of revising the design.

SPE is also language independent. The models are constructed from architectural and design-level information. Thus, SPE works with C++ and Java as well as with other object-oriented and non-object-oriented languages. The execution behavior of the software will be different with different languages. Nevertheless, this is reflected in the resource requirement specifications, not the model structure.

This integration is addressed in Chapter 15.

SPE can be easily integrated into the software development process. It has been used with traditional process models, such as the waterfall model. It works especially well with iterative, incremental processes such as the Spiral Model [Boehm 1988] or the Unified Process [Kruchten 1999], [Jacobson et al. 1999]. With an iterative, incremental process, you can use SPE techniques to assess and reduce the risk of performance failure at the project outset, and at each subsequent iteration.

1.5.1 What Does It Cost?

The cost of SPE is usually a minor component of the overall project cost. Lucent Technologies has reported that the cost of SPE for performance-critical projects is about two to three percent of the total project budget. For other, less critical, projects, SPE typically costs less than one percent of the total project budget. For projects where the performance risk is very high, the SPE expenditure may be as much as ten percent of the project budget.

SPE efforts can save far more than they cost by detecting and preventing performance problems. Bank One reported that, on one project, SPE costs over a five-month period were $147,000. During this time, the team analyzed three applications and identified modifications that resulted in a projected annual savings of $1,300,000 [Manhardt 1998]. Similarly, the performance engineering group at MCI reported a $20,000,000 savings in one year with SPE due to reduced resource

requirements that resulted in deferred configuration upgrades [CMG 1991].

1.5.2 What Do You Need?

To perform SPE studies, you need two essential skills:

- Estimation techniques for specifying best- and worst-case resource requirements
- Enough modeling background to understand the translation of OO models into performance models, and to understand the results

This book provides the information you need to develop these skills.

Other factors will make your SPE efforts easier. The following list starts with factors that are easy to find and progresses to those that require more effort but provide substantial benefits.

1. Performance specialists in your computer systems department (capacity planners, system programmers, and so on.) can assist with measurement studies and modeling techniques.

2. SPE tools make it easier for you to create and evaluate the models.

3. Management commitment provides sufficient time and resources to evaluate the risk of performance failures early in the project, and take appropriate steps based on the results. Management commitment provides the incentive and the ability to conduct SPE tasks.

4. Formal integration of SPE into the software development process with specific SPE milestones and deliverables makes SPE less people-dependent and ensures that it will be uniformly applied across projects.

5. An official SPE organization can provide a team of specialists to assist many projects with the SPE modeling and analysis steps. This organization is not the same as the computer system performance specialists in item 1. This team is more closely allied with the development organization, and provides software-specific performance engineering support.

These topics and others are also addressed in this book.

SPE is not a silver bullet—you cannot buy one book or one tool and expect that it alone will resolve performance problems.

SPE takes thought, effort, and analysis to produce the desired results. However, the return on investment for SPE more than justifies its use.

1.6 Summary

Performance is an essential quality attribute of every software system. Many object-oriented and non-object-oriented software systems, however, cannot be used as they are initially implemented due to performance problems. Systems delivered with poor performance result in damaged customer relations, lost productivity for users, lost revenue, cost overruns due to tuning or redesign, and missed market windows.

It is possible to cost-effectively design performance into new software systems. Doing this requires careful attention to performance goals throughout the life cycle. Software performance engineering (SPE) provides a systematic, quantitative approach to managing performance throughout the development process.

SPE 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. It is relatively easy to construct and solve these models to determine whether the proposed software is likely to meet performance goals. As the software development process proceeds, we refine the models to more closely represent the performance of the emerging software and re-evaluate performance.

This book discusses SPE for object-oriented systems. Object-oriented systems present special problems for SPE because their functionality is decentralized, and performing a given function is likely to require collaboration among many different objects from several classes. This makes the processing flow for object-oriented systems difficult to know a priori. Distributing objects over a network compounds the problem. In the following chapters, we illustrate how to use SPE techniques and the UML to overcome these problems.

SPE is not a silver bullet or a cure-all for performance problems. SPE takes thought, effort, and analysis to produce the desired results.

However, SPE efforts can save far more than they cost by detecting and preventing performance problems. The return on investment for SPE more than justifies its use.