25

The Performance and Scalability Perspective

Desired Quality	The ability of the system to predictably execute within its mandated performance profile and to handle increased processing volumes
Applicability	Any system with complex, unclear, or ambitious performance requirements; systems whose architecture includes elements whose performance is unknown; and systems where future expansion is likely to be significant
Concerns	Response time, throughput, scalability, predictability, hardware resource requirements, and peak load behavior
Activities	Capture the performance requirements, create the performance models, analyze the performance models, conduct practical testing, assess against the requirements, and rework the architecture
Architectural Tactics	Optimize repeated processing, reduce contention via replication, prioritize processing, consolidate related workloads, distribute processing over time, minimize the use of shared resources, partition and parallelize, use asynchronous processing, and make design compromises
Problems and Pitfalls	Imprecise performance and scalability goals, unrealistic models, use of simple measures for complex cases, inappropriate partitioning, invalid environment and platform assumptions, too much indirection, concurrency-related contention, careless allocation of resources, and disregard for network and in-process invocation differences

This chapter discusses two related quality properties for large information systems: performance and scalability. These properties are important because, in large systems, they can cause more unexpected, complex, and expensive problems late in the system lifecycle than most of the other properties combined.

Intel chief Gordon Moore observed in 1965 that the processing power of computer chips doubled approximately every eighteen to twenty-four months (now known as Moore's Law). This remark seems to apply as much today as it did in 1965, so one would hope that by now performance and scalability would have receded as major concerns for most computer systems. Unfortunately, this isn't the case, for a couple of reasons.

The most fundamental reason for performance concerns is that the tasks we set our systems to perform have become much more complex over time, and the demands we make on the systems (in terms of the complexity, number of transactions, number of users, and so on) have also grown in ways that would have been unimaginable in the 1960s.

To make matters worse, the performance of a computer system depends on much more than the raw processing power of its hardware. The way that hardware is configured, the way resources are allocated and managed, and the way the software is written can have significant impacts (good or bad) on the system's ability to meet its performance goals. The simple fact is that we haven't become much better at managing the performance of our systems since the 1960s—and we've actually gotten worse in some ways, such as our lack of attention to runtime memory use.

The scalability property of a system is closely related to performance, but rather than considering how quickly the system performs its current workload, scalability focuses on the predictability of the system's performance as the workload increases. Even if your system meets its goals today, how confident are you that it still will in the future? Will it be able to cope with increased numbers of users, transactions, or messages? Will it be able to handle increased complexity of processing? How will it behave when unexpectedly presented with a huge increase in workload?

Applying the Performance and Scalability perspective to your architecture will help you answer all of these questions.

APPLICABILITY TO VIEWS

Table 25–1 shows how the Performance and Scalability perspective affects each of the views we discussed in Part III.

CONCERNS

Response Time

Response time is the length of time it takes for a specified interaction with the system to complete. For a human-oriented system, this could be the length of

TABLE 25–1 APPLICABILITY OF THE PERFORMANCE AND SCALABILITY PERSPECTIVE TO THE SIX VIEWS

View	Applicability
Functional	Applying this perspective may reveal the need for changes and compromises to your ideal functional structure to achieve the system's performance requirements (e.g., by consolidating system elements to avoid communication overhead). The models from this view also provide input to the creation of performance models.
Information	The Information view provides useful input to performance models, identifying shared resources and the transactional requirements of each. As you apply this perspective, you may identify aspects of the Information view as obstacles to performance or scalability. In addition, considering scalability may suggest elements of the Information view that could be replicated or distributed in support of this goal.
Concurrency	Applying this perspective may result in changes to the concurrency design due to identifying problems such as excessive contention on key resources. Alternatively, considering performance and scalability may result in concurrency becoming a more important design element to meet these requirements. Elements of concurrency views (such as interprocess communication mechanisms) can also provide calibration metrics for performance models.
Development	One of the possible outputs of applying this perspective is a set of guidelines related to performance and scalability that should be followed during software development. These guidelines will probably take the form of dos and don'ts (e.g., patterns and antipatterns) that must be followed as the software is developed in order to avoid performance and scalability problems later when it is deployed. You will capture this information in the Development view.
Deployment	The Deployment view is a crucial input to the process of considering performance and scalability. Many parts of the system's performance models are derived from the contents of this view, which also provides a number of critical calibration metrics. In turn, applying this perspective will often suggest changes and refinements to the deployment environment, to allow it to support the performance and scalability needs of the system.
Operational	The application of this perspective highlights the need for performance monitoring and management capabilities.

time between the user initiating the request and the response being available for her use (e.g., the time from clicking a user interface button to seeing the response screen populated with data). For an infrastructure-oriented system such as a database, this could be the time between invoking a service and the service returning a response (e.g., the time from calling a query application programming interface to obtaining the query results).

We define two broad classes of response times you may want to consider separately.

1. *Responsiveness* considers how quickly the system responds to routine workloads such as interactive user requests. The response time for such operations is typically in the order of a few seconds. The key consideration for such workloads is user productivity, ensuring that the system does not slow down its users.
2. *Turnaround time* is the time taken to complete (turn around) larger tasks. This is typically measured in minutes or hours, and the key considerations are whether the task can be completed in the time available to it and the impact the task has on the system responsiveness while it is running.

These two classes of response times can affect different types of stakeholders and often require quite different technical solutions to make sure that requirements of each type are met.

EXAMPLE The following examples show how requirements could be specified for the two classes of response times.

Responsiveness

1. Under a load of 350 update transactions per minute, 95% of transactions should return control to the user within 5 seconds of pressing the submit button.
2. Under the reference load (defined in a separate document), 90% of service requests should return a reply to the calling program within the following times:
   - Open account: 30 seconds
   - Update account details: 10 seconds
   - Retrieve account status: 5 seconds
   - Retrieve balances: 5 seconds, plus 1 second per account accessed

Turnaround Time

1. Assuming a total daily throughput of 850,000 transactions, the process of establishing a consolidated position against each of the firm's external counterparties should take no longer than 4 hours, including writing the results back to a database. It can be assumed that no other system activity will take place during this period.

2. It must be possible to resynchronize the system with all of the production line monitoring stations and reset the database to reflect the current production line state within 5 minutes. It can be assumed that no status updates will be processed during the resynchronization period.

Throughput

Throughput is defined as the amount of workload the system is capable of handling in a unit time period. Throughput and response time have a complex interrelationship in most systems. In general, the shorter your transaction processing time, the higher the throughput your system can achieve. However, as the load on the system increases (and throughput rises), the response time for individual transactions tends to increase. Therefore, it is quite possible to end up with a situation where throughput goals can be met only at the expense of response time goals, or vice versa. We can illustrate this with a simple example.

EXAMPLE A database server can support up to 500 concurrent users performing sales transactions; however, as the number of concurrent users increases, the response time the users see increases as well.

- With 10 concurrent users, a typical transaction is processed in 2 seconds.
- With 100 concurrent users, a typical transaction is processed in 4 seconds.
- With 500 concurrent users, a typical transaction is processed in 14 seconds.

It takes 1 second of "thinking time" for a user to enter a transaction.

If we have only 10 users, each user can theoretically perform 20 transactions per minute, and our total possible throughput is a modest 200 transactions per minute.

If the load on the system rises to 100 users, each user can process up to 12 transactions per minute. Our total possible throughput rises to 1,200 transactions per minute, but at the cost of doubling the response time.

If we operate at our peak load of 500 concurrent users, each user can process up to 4 transactions per minute. Our total possible system throughput is 2,000 transactions per minute, but the response time cost has risen significantly for the users.

As the architect, you need to make sure that you and your stakeholders understand these interrelationships and that you have balanced your stakeholders' different performance goals.

Scalability

Most systems are subject to workload growth in some form. Scalability is the ability of a system to handle this increased workload, which may be due to an increase in the number of requests, transactions, messages, or jobs the system is required to process per unit of time or an increase in the complexity of these tasks.

Long-term scalability always has an associated time element that considers how soon the increase in workload is anticipated to arrive. You may also need to consider transient scalability—that is, the ability to handle short bursts of increased workload (such as increased traffic to an Internet news site during an international crisis).

Predictability

In addition to providing acceptable response time and throughput, another desirable property of a computer system is its ability to perform predictably. By this we mean that similar transactions complete in very similar amounts of time regardless of when they are executed. Similarly, the maximum transaction throughput the system can cope with should not vary significantly over time (in particular, it shouldn't decrease).

Predictability is often a more desirable quality than absolute performance.

EXAMPLE Call center agents use a customer service system to answer customer queries over the telephone. Having identified the customer, the agent executes a transaction to retrieve the customer's details. Whether the response time for this transaction is 1 second or 6 seconds probably doesn't matter that much—5 seconds isn't a long pause during such a telephone conversation, and agents can incorporate this delay into their

conversations with the customers. Therefore, a predictable transaction time of 6 seconds is acceptable.

However, if the system is unpredictable and produces the result in any time from 1 second to 15 seconds, then even if the average transaction response time is significantly less than 6 seconds, this is still less acceptable to the agents because it will result in awkward conversation pauses for many of the longer retrievals.

Hardware Resource Requirements

A major part of the performance and scalability puzzle is working out how much (and what type of) hardware your system will need, and this is usually an early concern in any project, being captured as part of the Deployment view. Hardware must be considered early because it costs money, takes time to acquire, often needs people to operate it, and often needs to be housed in purpose-built environments. The amount of hardware needed for a system usually has a significant and very visible impact on its overall cost.

In general, more hardware means higher throughput and better response times, albeit at higher cost. Given this fundamental tension between cost and performance, your role is often to establish the minimum amount of hardware that will allow the system to meet its performance goals.

Peak Load Behavior

All computer systems eventually exhibit poor performance as the load on them increases. If you plot a graph of the average transaction response time against the load on the system, it will usually have the shape shown in Figure 25–1.

The system behaves well for a while: As workload increases, response time increases in a predictable, linear fashion. However, at a certain point things start to go very wrong, and the response times increase sharply. This point is known as the "knee" in the curve. Soon, the graph ends up as a nearly vertical line, indicating that response times have become so long that the system is effectively unusable. This behavior is usually caused by one or more critical resources in the system becoming so overloaded that it can no longer work effectively (e.g., a network card is so swamped by incoming connection requests that it cannot service any of them effectively).

This sort of behavior is exhibited by virtually every system we have come across. However, it obviously isn't acceptable to experience this effect during normal system operation. This means that your challenge is to identify where the "knee" in the performance graph for your system is and to make sure that the corresponding workload level will be irrelevant during normal system operation.

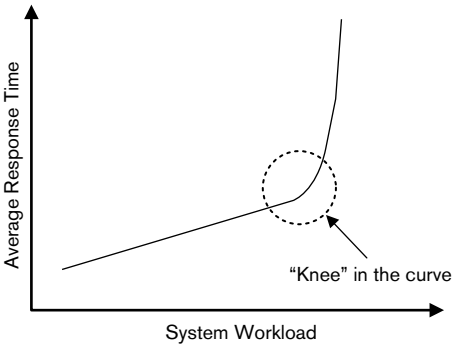

FIGURE 25–1 THE PERFORMANCE CURVE

ACTIVITIES: APPLYING THE PERFORMANCE AND SCALABILITY PERSPECTIVE

The activity diagram in Figure 25–2 illustrates a simple process for applying the Performance and Scalability perspective. In this section, we describe the activities in this process.

Capture the Performance Requirements

Ideally, you will already have a complete, consistent, and credible set of performance and scalability requirements as a result of the initial system requirements work performed. In reality, this usually isn't the case, and you need to collect these requirements, at a high level, as early as possible in the development lifecycle. Even if some requirements do exist, you need to verify their correctness and your understanding of them.

The performance requirements are often defined in business terms so that they are meaningful to the users of the system rather than the builders. Requirements such as "Be fast enough to support a 20,000-transaction-per-day back-office workload" are common and are useful because they describe the real stakeholder requirement rather than some abstract performance metric. However, at this stage, you need to translate such performance requirements into a set of key quantitative performance goals you are going to attempt to meet.

Setting such quantitative goals involves identifying the underlying performance metrics that are implicitly defined by the business-oriented requirements. From the previous example, you might end up with a definition of 5,000 information lookup requests, 10,000 specific transaction entry requests, and 5,000 report requests distributed over a 9-hour period, with a peak load of 20% of the transactions occurring in 45 minutes. This is a set of

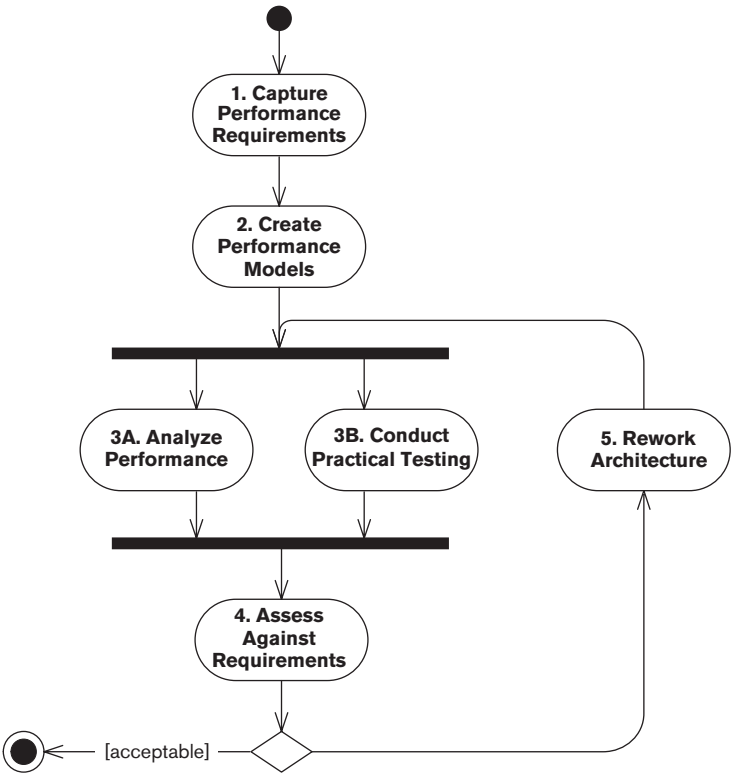

FIGURE 25–2 APPLYING THE PERFORMANCE AND SCALABILITY PERSPECTIVE

specific, quantitative goals, and you can establish whether or not you have met them by measurement and analysis.

A common problem with this process is understanding the business requirements enough to translate them into quantitative goals. Using experience, careful analysis, and the knowledge of domain experts who understand the workload are all possible solutions to this problem.

NOTATION Simplicity and clear communication are the goals when communicating performance requirements, and we have found that a simple approach based on text and tables is quite sufficient.

ACTIVITIES

Specify the Response Time Requirements. Response time targets are meaningful only in the context of a defined load. It may be easy for a computer

system to process a single transaction in 3 seconds but much harder to achieve this target when it is receiving 500 transactions a second. This means that response time requirements need to specify the context as well as a clearly defined response time goal (which also defines when a transaction starts and ends). In most systems, response time under constant load will vary according to some sort of distribution curve. Most transactions will complete at or near the average response time, but some will take longer, and a few will complete more quickly. In most cases, it isn't reasonable to expect every transaction to complete within the target response time. It is more realistic to require a certain proportion (such as 90% or 95%) of transactions to meet the target.

Specify the Throughput Requirements. Throughput is typically defined in terms of transactions per unit time (second, minute, or hour), where a transaction is a clearly definable unit of work, recognizable to a user of the system. The transactions used for throughput planning should normally be derived from the system's most important usage scenarios (rather than a specific technology-oriented transaction item, such as a database insert statement).

Specify the Scalability Requirements. Scalability requirements are usually defined in terms of the increase in workload that the system must be able to absorb over particular time periods while continuing to meet its existing response and throughput goals. Scalability requirements should also make clear any changes to the system that will be needed to meet these increased workload levels.

Create the Performance Models

Being able to collect performance data isn't useful in itself. You need to use the data in an effective way to allow you to understand and improve your system's performance. A key part of this process is creating performance models that allow you to gain this insight. Such a model allows you to assess the maximum "theoretical" workload for your system, supplies useful estimates for capacity planning, and provides a set of measures against which the actual system can be compared to assess its performance.

The types of models you can use for performance analysis vary widely from a few simple calculations on a scrap of paper to sophisticated statistical models to complete online simulations of systems. All these models have their place, but given our degree of expertise and the space available, we will limit our discussion to relatively simple pencil-and-paper performance models involving basic representations of system structure with the simple statistics used to analyze them. The performance engineering books mentioned in the Further Reading section at the end of this chapter provide more information on performance modeling.

NOTATION We have found the following notation methods most helpful when creating performance models.

- *Performance modeling notations*: Specialists in this area have developed several graphical and mathematical performance modeling approaches, including execution graphs, augmented Petri Nets, approaches based on queuing theory, and statistical approaches (see the Further Reading section for more information). Most of the notations used in these approaches are extensions of previously existing notations, with the advantage of being tailored specifically for performance modeling. As with most specialist notations, they often have the disadvantages of complexity and unfamiliarity; many are not widely understood outside the specialist computer measurement community.
- *Ad hoc diagrams*: A simple block diagram notation is probably sufficient for many performance models because they are not terribly complex. In fact, you can use a UML deployment diagram with some ad hoc extensions as the basis for a performance model. Such an approach has the virtue of simplicity but may have limitations for more sophisticated modeling applications.
- *Text and tables*: Some of the performance model will probably need to be captured using text and tables to describe model elements, capture key metrics, and illustrate relationships between elements that the graphical notation does not make clear.

ACTIVITIES

Identify the Performance-Critical Structure. Use the Deployment view of the system as the basis of the model by simplifying it to its essential performance-critical elements—such as processes, nodes, network links, and the main data storage (such as your main databases). Create a new, simple block diagram of the system illustrating the main runtime system elements and how they are connected.

Identify the Key Performance Metrics. Review the block diagram and identify the parts that need to be annotated with performance data to allow the creation of performance estimates. This normally includes the processing time for the main functional elements of the system, the request latency between the main system processes, the length of time taken by a typical database operation, the number of concurrent requests that each major element can handle, and so on.

Estimate the Performance Metrics. At this stage, the values for most of the key performance metrics are probably unclear. For each such metric, you need to derive a reliable estimate of its value. Some may be fairly obvious from previous experiences that you or others on your team have had. For the rest of

the metrics, try to create quick prototypes that allow you to derive estimates. If prototypes aren't practical, intelligent guesswork is probably your only remaining option. Whichever approach you use, make sure that the estimates are valid for a realistic workload and not just for a single transaction. When you've completed this process, the result will be a simple performance model you can use for prediction. To estimate the theoretical processing time for an element of system workload, you can trace its execution through the model and piece together the relevant performance estimates.

EXAMPLE Figure 25–3 shows an example of a simple performance model that might be built to investigate the performance of an order-entry process for an order-processing system.

The diagram shows that we have identified five performance-related parts of the system (plus the Browser Client), whose interactions we judge to be the crucial factors determining system performance. For each of these elements (or more accurately, those servicing others), we have estimated the response times that the element will provide to its clients under defined conditions. We have also estimated the communications latency between the elements, which varies widely (due to different technologies and deployment decisions).

This model provides two valuable insights. First, we can see how long we think a couple of crucial system transactions will take to execute. (We could extend this model or build other similar ones to investigate other aspects of the system's performance.) Second, the model helps us understand the set of performance-related assumptions we are making (such as the invocation latency being minimal between the Order Processor and the Price Calculator), which, if incorrect, may cause us problems later. The model also helps us focus on another step in the process, practical testing.

As we mentioned, the approach illustrated in Figure 25–3 is very simple. Although the model is helpful and provides useful insights, we made a number of important simplifications—for example, we're largely ignoring the modeling of queuing within the system. If you want to create more sophisticated performance models of your system, we recommend some books in the Further Reading section at the end of this chapter.

Analyze the Performance Models

In parallel with practical benchmark testing, you can calibrate the performance models with the previously estimated performance metrics and use the results to estimate likely system throughput under different scenarios. The advantage of using performance models is that analysis is cheaper, simpler,

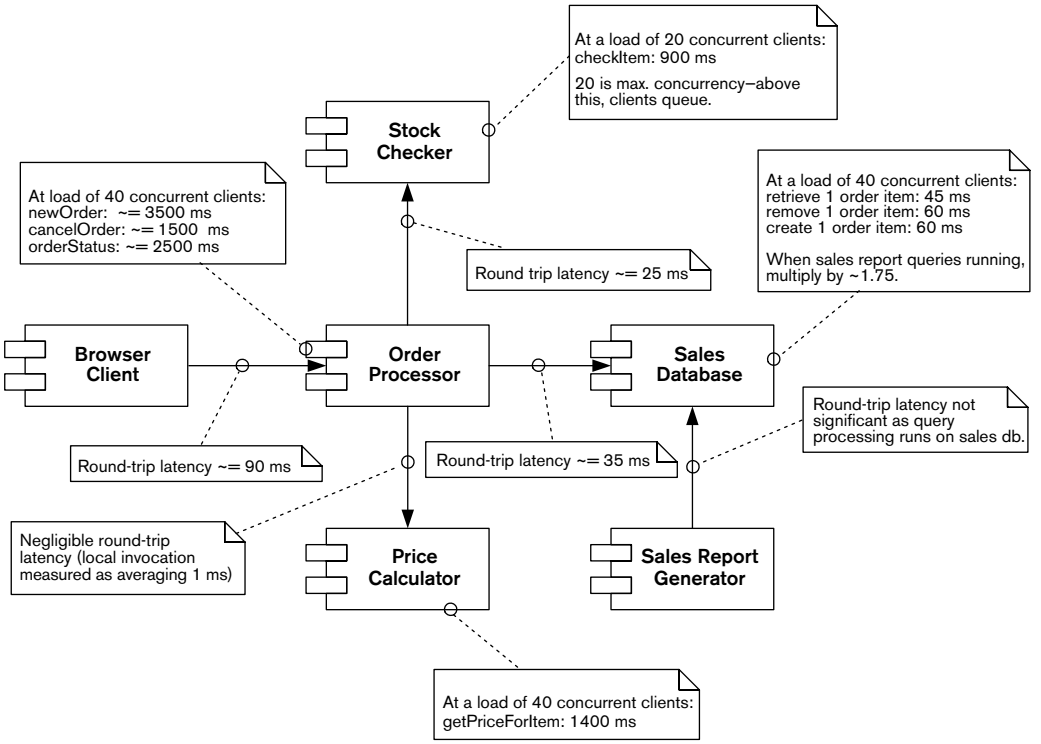

FIGURE 25–3 A SIMPLE PERFORMANCE MODEL

and quicker than benchmarking. The disadvantage is obviously that the results are only as good as the models, so they need to be carefully validated by some practical testing as well.

Consider using the performance models to explore a large number of scenarios quickly to find potential trouble spots, and then use this knowledge to drive more involved practical work.

NOTATION Analysis of performance models results in calculated performance data that you can usually represent in tabular form.

ACTIVITIES

Characterize the Workload. The first activity is to establish the "shape" of the workload your system needs to be able to process. This involves prioritizing and estimating the volume of each sort of request the system has been specified to handle, including an estimate of the routine processing needed.

Estimate the Performance. Having identified the kind of workload you expect, you can estimate the processing time of each piece of that workload. This involves using the performance models to identify the elements of the system involved in processing each piece of the workload and to determine how long the likely best- and worse-case durations will be. From this set of inputs, you can establish the estimated best- and worst-case processing times for each element of the expected workload. It is sensible to multiply these estimates by a factor that reflects the hidden inefficiencies likely to be present in the system. (A value of 20% is a good starting point for an information system.) At the same time, identify the least scalable element involved in processing each workload item, and use this information to estimate how many concurrent requests your system can handle.

Conduct Practical Testing

Although performance modeling is very valuable, it is of limited use in isolation; practical testing is also important. The testing you do can range from some simple isolated tests for assessing critical performance metrics to large-scale prototype benchmarking. The amount of practical testing you should perform depends on how confident you are that you can assess the likely performance of your system without it (and, of course, on how much time you have available to spend on this aspect of your architecture). For a system structure you have used before, with well-tested technology, in an application domain you have experience of, little practical testing may be needed. In a new domain, with new technology, or with a novel system architecture, you will probably want to perform a lot of tests.

NOTATION You can write up test results in a short report if appropriate.

ACTIVITIES

Measure and Estimate the Performance Metrics. To support the process of creating a performance model, you need to measure or estimate values for a number of important performance metrics. A common performance testing activity is to perform a series of practical tests to estimate these metrics accurately. Such testing typically involves quickly prepared trial scenarios that allow a particular performance metric to be empirically tested (such as network response times from a Web server under load). The main pitfall to avoid with such testing is oversimplification of the test scenario, which results in much better performance than would be expected in a real deployment. Make sure that trial scenarios simulate a realistic context for the test metric (such as ensuring a realistic workload on the system element being tested).

Perform Benchmark Tests. Metric estimation is a highly specific activity that focuses on one microlevel metric. It can also be useful to perform more com-

plex practical testing to find out what performance can be expected from a particular configuration of candidate system elements. Such testing is often referred to as *benchmarking* and usually involves creating a trial end-to-end system to allow system-level performance to be estimated. As such, its relationship to metric estimation is similar to the relationship between integration and unit testing. Benchmarking is more involved than simple microlevel performance testing, but the major pitfall is the same: namely, ensuring that the test is valid. Make sure that the limits of the benchmark are understood and that it captures enough of the key characteristics of the planned system to provide a useful insight into its likely performance. (For example, make sure that system elements perform enough processing to be representative of the elements in the system under development.)

Assess against the Requirements

Having completed performance analysis and practical testing, you can compare the results of this process against the performance requirements and draw some initial conclusions. These conclusions may indicate that all should be well or that modifications need to be made to the architecture to resolve potential performance problems. You may also believe that further testing and analysis are required because you made assumptions that you no longer feel comfortable with.

The result of this process should be a decision to either complete this cycle of performance work or circle back through the process. If the latter, you will need to make appropriate modifications to your proposed architecture and repeat the analysis and testing cycle as needed, to increase or deepen your knowledge of how the system will behave from a performance perspective.

NOTATION This activity is a process rather than a model-building activity. However, you do need to clearly document its outputs so that decisions, assumptions, and assessments are captured for everyone's use as the system development process moves forward. Document this activity's outputs as concisely as possible (probably by using plain text), and over time you will create a useful library of performance information that can be applied in other situations.

ACTIVITIES

Identify the Risks. Reconsider the performance risks you believe you are facing, based on your testing and analysis. Clearly record and justify those risks you think are still a problem and those that have been addressed. For those that are still troublesome, identify why they are still present, and use these conclusions to drive the next iteration of performance work, should you decide you need one.

Review the Requirements. Work through each of your performance requirements and demonstrate to your own satisfaction (and the satisfaction of any interested stakeholders) whether or not the proposed architecture will meet each requirement.

Rework the Architecture

The output of this performance work is likely to be a number of cross-view changes you need to make to your candidate architecture. The most likely impact will be to the functional and deployment designs, but other aspects of the architecture (particularly the information- and concurrency-related parts) are also candidates for performance-related changes. Many of the tactics described in the next section imply cross-view changes (e.g., in extreme cases, partitioning and parallelization can require changes to the functional, concurrency, information, and deployment structures in order to support effective partitioning and parallel execution of the workload and consolidation of the results).

After changing the architecture, move into the next iteration of the process by modifying your performance model and rerunning or adding practical testing, to establish whether or not the proposed changes have the required effects.

NOTATION Use the same architectural notations used in your view models.

ACTIVITIES Any sort of architecture definition activity required to improve performance is relevant here, particularly the tactics discussed in the next section.

ARCHITECTURAL TACTICS

Optimize Repeated Processing

An old software engineering heuristic states that most systems spend 80% of their time running 20% of the system's code. This certainly matches our experience that most systems have a small number of common operations that the system spends most of its time performing.

The resulting performance implication is that you should focus your performance efforts on that core 20% of your system. To state this in slightly more sophisticated terms:

$$\text{operation total cost} = \text{operation invocation cost} \times \text{operation invocation frequency}$$

We can consider the total cost of a system operation to be the cost of a single invocation multiplied by the number of times we will invoke it during a unit time (e.g., per day). In turn, we can note that:

$$\text{system workload} = \sum_{(1..n)} \text{operation total cost}$$

The total workload for our system, for a unit time, is the sum of all of the total operation costs over that unit time (where we have *n* possible operations in our system).

In order to focus your performance engineering effort, rank your system's operations by the total cost metric, and make sure that you optimize the operations at the top of the list first.

Having this information also helps you make intelligent tradeoffs between operation optimizations. In many cases, optimizing for one operation can have a negative impact for another. In general, when you have to make these tradeoffs, the needs of the frequent operations should take precedence.

EXAMPLE A message bus is a piece of software infrastructure that allows applications to exchange messages easily and efficiently. The message bus receives messages from senders, performs any data transformation required, calculates how to route the message to its intended recipients, and delivers the message to them.

In order to process messages efficiently, the message bus could maintain information on its nodes, the routes between them, and the connectivity characteristics of each (such as communication latency). The bus would use this information to derive the most optimal route between message senders and receivers. This speeds up the process of route selection (which is a frequent activity), but the tradeoff is that whenever a node or link is added or removed (which happens rarely), the entire set of route tables has to be recalculated, which is a potentially expensive operation.

The goal of the performance engineering process is to minimize the overall system workload; this is one of the few sure ways to improve system performance.

Reduce Contention via Replication

As we discussed in Chapter 18, whenever you have concurrent operations in your system, you have the potential for contention. This contention is often a major source of performance problems, causing reduction in throughput and wasted resources.

Eliminating contention can be a difficult process, particularly when you must deal with a single point of contention (such as a shared data structure inside an operating system process). This can occur when the contention involved is actually within the system's underlying infrastructure (e.g., an application server), rather than in the software over which you have direct control.

A possible solution for some contention problems is to replicate system elements—hardware, software, or data.

EXAMPLE Many large Web sites support many millions of page views per day, far beyond the capacity of even the largest computers. They provide this service by deploying vast server farms, made up of hundreds or even thousands of computers each running a separate instance of the Web server. Special hardware is used to allocate incoming requests evenly across the Web servers to ensure that response times are consistent and that usage is maximized.

Of course, in this scenario, the network connection into the server farm can still be a bottleneck. By replicating the server nodes, we may just be moving the bottleneck to a different part of the system. This is a very common feature of performance work—solving a problem simply uncovers the next bottleneck in the system!

EXAMPLE A lottery system supports remote point-of-sale terminals in retail locations throughout the country. When a terminal is switched on in the morning, it is necessary to enter a user name and password before the terminal can be used. This authentication is done through a single central database, which experiences very heavy load between 7 and 8:30 A.M., resulting in login times of up to a minute.

To alleviate this problem, several regional authorization databases are set up, with each one supporting about 10% of the overall terminal population. Programs are written to distribute login information to each database overnight. As a result, login times are significantly improved.

This approach works only in certain situations, and a limiting factor is that the system often needs to be designed to take advantage of replication from the outset. However, where possible, it is worth considering because it can be a lot easier to solve a contention problem by avoiding it completely in this way rather than having to solve it directly.

Prioritize Processing

The workload in your system will vary in terms of its importance, ranging from extremely critical processing that needs to be performed as quickly as possible to routine work such as housekeeping that can be completed over an extended period.

A problem in many otherwise well-built systems can emerge when the overall performance of the system is within target, but some important tasks still take too long to execute. In these systems, the underlying hardware is typically busy, and the expected overall throughput is being processed. But critical workload is being processed at the same rate as less important operations, and this leads to the perception of a performance problem.

To avoid this situation, partition the system's workload into priority groups (or classes), and add the ability to prioritize the workload to process. This allows you to ensure that your system's resources will be applied to the right element of workload at any point in time, so the perception of performance problems is much less likely. A low-level example of this approach is the priority class-based thread and process scheduling built into most modern operating systems.

When dealing with large information systems, this prioritizing usually involves identifying the business criticality of each class of workload. Those types of workloads that have to complete in a timely manner for business to continue (such as order processing) are naturally prioritized over workloads that, while important, will not immediately impact business operation (such as management reporting).

EXAMPLE A Web-based system to support e-commerce may need to support order capture, customer account management, stock reporting, and sales reporting functions.

Probably all of the customer Web workload (such as order capture and customer account management) should be prioritized over other processing. The customers who use the Web site are extremely important stakeholders because a slow Web site will not only reduce immediate sales but may drive the customers to other vendors, thus damaging the business in the longer term.

In contrast, while the management team members are influential stakeholders, if the management reporting functions run slowly during times of peak load, it is unlikely to directly harm the business. Of course, making such a rational tradeoff can be challenging when dealing with live stakeholders.

The practical problem in most situations is finding the right balance between the different types of workloads. A number of factors need to be taken into account, including the relative importance of different stakeholders, the varying importance that different stakeholders place on different sorts of workloads, and the deadlines for processing various workloads. Finding the right balance for complex situations is often a case of applying

your judgment to find an acceptable option, rather than discovering any absolute "right" answer.

Consolidate Related Workload

The processing of most operations in an information system requires a certain amount of context to be available in order for the processing to take place. The management of this context information can itself be a significant overhead when the operation to be performed is small or the context is expensive to locate (e.g., when loaded from a database).

To address this, consolidate related workloads into batches and process groups of related requests together. This pattern of processing normally allows a single initialization step, a number of operation processing steps, and then a single tear-down step—so saving the initialization and tear-down steps that would be required for each operation if processed separately.

EXAMPLE A generic rating engine uses actual and predicted foreign exchange rates to calculate values in different currencies. To use the engine, you supply an amount, a "from" currency, a "to" currency, and a date—either spot (now) or forward (in the future)—and the engine performs the required conversion. To do this, it has to calculate a table of forward exchange rates based on complex formulas. Although this calculation is fairly quick, because it is repeated for each value supplied it has a significant impact on performance.

The engine is enhanced to take a list of conversions as input, rather than just a single one. This improves performance significantly because, for a batch of a hundred values, the calculations have to be performed only once rather than a hundred times.

Such an optimization is inherently related to the structure of the system and the way operations are processed within the system, so it is something best considered as part of architecture definition. Doing so can lead to large efficiency gains for some systems.

Distribute Processing in Time

Some systems need to process a similar workload continually at all times of day or night. However, in our experience, such systems are pretty rare (although some Internet systems do fall into this category). Because most internal systems support people at work, and because people work during relatively

fixed hours, systems often have different load requirements at different times of day. This means that the workload level for a system varies over time. The problem during the busy times is that (as we have already seen) concurrency tends to cause performance problems.

A useful system-level strategy for reducing system load, resource contention, and thus performance problems is to even out peaks and troughs of processing. Some peak workload is simply unavoidable—the result of the way people work. However, consider spending time during the architecture definition process to carefully analyze your system's workload during the peak times. In many cases you will find that some of the workload can be postponed to other times in your processing cycle. By moving these parts of the workload, you will improve performance during peak load times and use idle resources during quieter times.

Minimize Use of Shared Resources

At any particular time, each nonidle task running on a system is in one of two states:

1. Making use of a resource (e.g., a hardware resource such as processor, memory, disk, or network or a software resource such as a message service)
2. Waiting for a resource, either because it is busy (e.g., being used by another task) or not in a ready state (e.g., waiting for a head to move over a particular track on a disk or a software service to initialize)

As systems get busier and contention for shared resources increases, the waiting time takes proportionally more of the total elapsed time for tasks and contributes proportionally more to the overall response time.

Increasing the performance of resources that the task uses will help (because in general you have to wait less time for the tasks ahead of you in the queue to finish), although this may not be possible for a number of reasons, perhaps because you have reached the limits of the technology. The other way to alleviate this situation is to minimize the use of shared resources—that is, reduce the situations in which waiting has to occur. This is a complex topic, beyond the scope of this book, but you may want to consider the following strategies.

- Use techniques such as hardware multiplexing to eliminate hardware hot spots in your architecture.
- Favor short, simple transactions over long, complex ones where possible (because transactions tend to lock up resources for extended periods).

- Do not lock resources in human time (e.g., while waiting for a user to press a key).
- Try to access shared resources nonexclusively wherever possible.

Partition and Parallelize

If your system involves large, lengthy processes, a possible way to reduce their response time is to partition them into a number of smaller processes, execute these subprocesses in parallel, and, when the last subprocess is complete, consolidate all of the subprocess output into a single result. Whether this approach is likely to be effective in a particular situation depends on four primary factors:

1. Whether the overall process can be quickly and efficiently partitioned into subprocesses
2. Whether the resulting subprocesses can be executed independently to allow effective parallel processing
3. How much time it will take to consolidate the output of the subprocesses into a single result
4. Whether enough processing capacity is available to process the subprocesses in parallel faster than handling the same workload in a single process

If the overall process cannot be split easily into independent subprocesses, this technique isn't practical. Situations involving lengthy and expensive consolidation of subprocess output are suitable for this technique only if the consolidation cost is still small relative to that of the longer response time for the original process. Finally, if spare processing resources aren't available, parallelization is unlikely to be effective because the subprocesses will be executed one after the other and will be slower than the original design, due to the partitioning and consolidation overheads.

It is also important to remember that this approach is less efficient than using a single linear process (due to the partitioning and consolidation overheads) and achieves a reduction in response time at the price of requiring more processing resources.

EXAMPLE A feature offered by a sales management system is to generate a number of derived measures of sales performance (volume, profitability, average price, and so on) across the company, broken down by region. This feature is likely to involve a lengthy process of calculation over the sales information held in the system but is also likely to be a

good candidate for partitioning and parallelization. If the request is partitioned into a process for each sales region (rather being run as a single process), each region's results can be calculated in parallel and the results consolidated to form the final report. If the underlying data can be efficiently accessed in *n* parallel streams, assuming similar data volumes per sales region, the parallel process is likely to complete in roughly 1/*n*th of the time taken if run as a single process (plus some time for result consolidation, which is likely to be small compared to calculation time). This response time can be greatly reduced, at the cost of some processing efficiency.

Use Asynchronous Processing

One way to improve perceived response times for users is to carry out some processing asynchronously—that is, in the background, after the system has returned a response to the user.

EXAMPLE Several desktop computers share access to a single high-quality color printer by means of a shared printer server. Desktop applications send print requests to the print server, which manages them through a set of queues. Print requests for even the largest jobs complete as soon as the request has been received and acknowledged by the print server, rather than having to wait for the print to physically complete.

Once a document has been printed, the server notifies the originating PC, and a small message box is displayed to the user. Of course, this may happen some time later—possibly not until the next day in the case of very large print jobs. If printing fails for any reason, notification is also sent to the desktop, and it is the user's responsibility to resubmit the job.

You need to use this technique with some care, and only where it is really necessary, for several reasons. First, it is usually significantly more complex to implement, requiring some sort of background service to carry out the asynchronous parts of processing for you. More important, you have to develop strategies for dealing with situations where the background processing fails: It may be minutes or even hours after the transaction was initiated, and the user may not even be available to take corrective action. Finally, asynchronous processing doesn't actually reduce the amount of workload that needs to be performed or make the system more efficient (and thus actually solve the performance issue); it just moves the workload around.

Make Design Compromises

If other performance tactics you've tried have not resulted in acceptable performance or cannot be applied for some reason, you may need a more extreme approach. Many of the techniques of good architecture definition that we have described in this book can themselves cause performance problems in extreme situations. For example, a loosely coupled, highly modular, and coherent system tends to spend more time communicating between its modules than a tightly coupled, monolithic one does.

Although a loosely coupled and highly modular design should always be the goal, you should be aware that the benefits it provides might come at some performance cost. Where performance is a critical concern and other tactics have failed, you might need to compromise the ideal structure of your design, for example, by:

- Moving to a more tightly coupled, monolithic design to minimize internal communication delays and contention
- Denormalizing parts of your data model to optimize accesses to data
- Using very coarse-grained operations between distributed system elements
- Duplicating data or processing locally to minimize traffic over slow communication channels

Making such a change may improve performance but is likely to have costs in terms of maintainability and possibly even ease of use. You should carefully assess the desirability of these tradeoffs.

PROBLEMS AND PITFALLS

Imprecise Performance and Scalability Goals

It is far too common for even large projects to have vague, incomplete, or ambiguous performance goals or to have failed to consider the scalability of the system altogether. This is simply storing up trouble for the future—you won't have a suitable framework for designing, tuning, and building the system, and it won't be possible to determine unambiguously whether or not the system is performing acceptably.

RISK REDUCTION

- Define and obtain approval from your stakeholders for clear, measurable performance and scalability goals.
- Satisfy yourself that your performance and scalability goals are realistic and achievable.

- Communicate the goals to the architecture, design, and build teams so that they can be factored into their work.
- Make sure that acceptance criteria are based only on the agreed goals and do not include other (possibly implied) performance or scalability goals.

Unrealistic Models

The process of building a performance model can be an absorbing one, and the result is often a sophisticated model of the likely performance characteristics of the system. A problem that can result from an impressive model is an overreliance on its abilities. It is easy to be lulled into a false sense of security when a model suggests that no performance problems exist. A lack of problems revealed by a model does not necessarily mean that no problems exist in the real system; the model is an abstraction of reality and only as good as its match with that reality.

RISK REDUCTION

- Balance and augment the performance modeling activity with enough practical testing to make sure that the assumptions underpinning the models are valid and that the conclusions resulting from them are credible.
- Continue parallel practical testing right through the modeling process.
- Always check your modeling results against practical test results.

Use of Simple Measures for Complex Cases

Achieving acceptable performance in a computer system is a complex process, with many variables to worry about simultaneously. A common pitfall is to oversimplify the performance testing and modeling process to make it easier, but then to assume that its results will apply to much larger and more complex cases by simple analogy. Practical testing should reflect the real runtime environment you expect for your system, and if your performance models are very simple, the conclusions drawn from them must be used with care.

As a simple example, consider the practical testing you perform to calibrate your model. If these practical tests do not simulate a realistic runtime environment (e.g., by being single-threaded rather than heavily concurrent), the results of these tests are unlikely to reflect the way the real system will behave.

RISK REDUCTION

- Continually question the validity of your analysis and testing conclusions.
- Consider the differences between the test environment and the real system runtime environment to spot critical divergences that are likely to invalidate the performance engineering process.

Inappropriate Partitioning

In Chapter 16, we discussed poor partitioning as a possible pitfall for the Functional view, but it is often also problematic when considering performance. Partitioning becomes a problem when one or more elements appear to be involved in nearly all of the transactions in the system. This often means that these elements become bottlenecks and prevent acceptable performance being achieved anywhere in the system because of their dominant role.

A related problem, which is often a symptom of poor partitioning, is to ignore the performance differences of local and remote processing. We discuss this problem near the end of this chapter when we talk about network and in-process invocation.

RISK REDUCTION

- Watch for functional elements that are highly connected to a large percentage of the system's other functional elements (see the discussion of "God elements" in Chapter 16). These system elements may be your potential bottlenecks.

Invalid Environment and Platform Assumptions

Whenever an innovative system is developed, a level of risk exists because of assumptions that have to be made about its environment or the underlying technology. This risk can result from new technology that has not been widely used before (or perhaps not on this scale), or it can result from assumptions about the system's environment (e.g., peak request volume).

Some of these risks are unavoidable and are simply the result of being first. What is surprising is the number of projects that either run into performance problems in well-understood areas or simply don't control their risks, even when this is perfectly possible.

RISK REDUCTION

- Identify and validate your assumptions, and assess them for risks as part of performance analysis.
- Identify your possible mitigation strategies if your assumptions are proved incorrect.

Too Much Indirection

It has been said that any problem in software engineering can be solved by adding another level of indirection.[1] It is certainly true that indirection is a

1. Attributed to Andrew Koenig.

powerful tool—used wisely, it can make systems more flexible, easier to change, more powerful, and even more elegant. However, another famous adage also applies: There is no such thing as a free lunch!

A problem with indirection is that it introduces hidden work into the system. Some forms of indirection (e.g., object references in an object-oriented language) do not normally introduce enough overhead to cause a problem in anything apart from the most performance-critical situations. However, other forms of indirection (e.g., key mapping in a database) can add a significant percentage of overhead for certain sorts of processing (e.g., some updates may need two disk writes rather than one).

RISK REDUCTION

- Be careful how much indirection you introduce into the performance-critical parts of your system.
- If you have a lot of indirection in the implementation of critical path operations, carefully assess the tradeoff between the functional advantages and the performance disadvantages.

Concurrency-Related Contention

Any system with concurrent processing and shared resources has the potential for contention between threads of execution. In its most extreme form, this contention can slow a concurrent system to a crawl as threads spend most of their time "thrashing" in and out of wait states trying to obtain access to shared resources. Even in less extreme cases, such contention can become a significant bottleneck and cause performance problems throughout the system. Careful analysis and design of your system to avoid such bottlenecks is an important part of the performance work for concurrent systems.

RISK REDUCTION

- Inspect your Functional and Concurrency views to identify the elements of your system that must deal with a significant amount of concurrent processing.
- Investigate the proposed implementation of critical elements as part of your modeling, testing, and analysis process to convince yourself that they will not become bottlenecks that grind your system to a halt.
- During software construction, test the concurrent behavior of critical elements as early as possible.

Careless Allocation of Resources

Increases in computing power in recent years mean that, in general, we need to be much less conscious of using hardware resources frugally than used to

be the case. However, an often-overlooked problem related to this new freedom is the fact that the process of allocating and freeing runtime resources (such as memory or locks) requires resources itself. This is easy to forget because the process is often implicit (such as the Java language's automatic garbage collection or a relational database's automatic lock allocation). However, this doesn't mean it is free!

A problem that can creep into many information systems is excessive allocation and freeing of runtime resources. This is often revealed by elements that appear to run much slower than expected without any obvious way for the time to be lost.

RISK REDUCTION

- Avoid large amounts of dynamic resource allocation and deallocation in critical path elements.
- Consider preallocating resources at less critical times (such as startup or during quiet periods).
- Consider whether allocated resources can be reused more cheaply than freeing and reallocating them.
- Work with software developers to understand the problem and document guidelines and patterns to explain good practice (in the Development view).

Disregard for Network and In-Process Invocation Differences

Modern computing technology provides the option of distributing a system across a number of machines and accessing resources located anywhere on the network. Such distribution is a major feature of many modern information systems. In fact, much of today's information systems technology aims to allow the location of deployed elements to be changed after they have been developed.

It is easy to accidentally ignore the performance differences between invoking operations locally within an address space (process), between two processes on one machine, and between processes on machines located an immense distance apart. In reality, the response times of these different situations can differ by an order of magnitude or more.

If you ignore this critical distinction, you run the risk of extremely unpleasant performance surprises when you deploy the system and find that one or more of your critical interelement interactions is ten times slower than you had assumed due to the locations of key system elements.

RISK REDUCTION

- Consider interelement distribution and possible remote invocation as part of your fundamental architecture definition process.
- Make sure that the locations of elements and their interelement invocation costs are accurately reflected in your performance model to allow you to take possible invocation latencies into account.

CHECKLISTS

Checklist for Requirements Capture

- Have you identified approved performance targets, at a high level at least, with key stakeholders?
- Have you considered targets for both response time and throughput?
- Do your targets distinguish between observed performance (i.e., synchronous tasks) and actual performance (i.e., taking asynchronous activity into account)?
- Have you assessed your performance targets for reasonableness?
- Have you appropriately set expectations among your stakeholders of what is feasible in your architecture?
- Have you defined all performance targets within the context of a particular load on the system?

Checklist for Architecture Definition

- Have you identified the major potential performance problems in your architecture?
- Have you performed enough testing and analysis to understand the likely performance characteristics of your system?
- Do you know what workload your system can process? Have you prioritized the different classes of work?
- Do you know how far your proposed architecture can be scaled without major changes?
- Have you identified the performance-related assumptions you have made (and validated them if needed)?
- Have you reviewed your architecture for common performance pitfalls?

FURTHER READING

You can find a very practical yet thorough tutorial on the process of performance engineering for real systems in a book written by two well-known and widely regarded specialists in the field [SMIT02]. We have found this book very useful, and it has influenced much of our thinking in this area. A number of our performance pitfalls are similar to performance antipatterns in this book (particularly their God Class, Circuitous Treasure Hunt, One-Lane Bridge, and Excessive Dynamic Allocation), and this book provides valuable advice on recognizing and avoiding these problems.

Another comprehensive, if rather more daunting, explanation of performance engineering focuses on its quantitative aspects [JAIN91]. This book contains more information on statistical and simulation-based performance engineering than most of us will ever be able to apply, but it presents comprehensive explanations of a number of very usable techniques.

Two examples of practical, technology-focused performance books are Wise [WISE97] and Killelea [KILL98]. The first provides a lot of hands-on information on how to solve (and avoid) performance problems in distributed systems. The second focuses on Web-based systems and provides good examples of how to consider performance work for such systems.

Finally, if you'd like to check what Moore's Law really says, Intel makes it available via its Web site at www.intel.com/research/silicon/mooreslaw.htm.