8

Performance

An ounce of performance is worth pounds of promises.
—Mae West

It's about time.

Performance, that is: It's about time and the software system's ability to meet timing requirements. When events occur—interrupts, messages, requests from users or other systems, or clock events marking the passage of time—the system, or some element of the system, must respond to them in time. Characterizing the events that can occur (and when they can occur) and the system or element's time-based response to those events is the essence is discussing performance.

Web-based system events come in the form of requests from users (numbering in the tens or tens of millions) via their clients such as web browsers. In a control system for an internal combustion engine, events come from the operator's controls and the passage of time; the system must control both the firing of the ignition when a cylinder is in the correct position and the mixture of the fuel to maximize power and efficiency and minimize pollution.

For a web-based system, the desired response might be expressed as number of transactions that can be processed in a minute. For the engine control system, the response might be the allowable variation in the firing time. In each case, the pattern of events arriving and the pattern of responses can be characterized, and this characterization forms the language with which to construct performance scenarios.

For much of the history of software engineering, performance has been the driving factor in system architecture. As such, it has frequently compromised the achievement of all other qualities. As the price/performance ratio of hardware continues to plummet and the cost of developing software continues to rise, other qualities have emerged as important competitors to performance.

Nevertheless, all systems have performance requirements, even if they are not expressed. For example, a word processing tool may not have any explicit performance requirement, but no doubt everyone would agree that waiting an

hour (or a minute, or a second) before seeing a typed character appear on the screen is unacceptable. Performance continues to be a fundamentally important quality attribute for all software.

Performance is often linked to scalability—that is, increasing your system's capacity for work, while still performing well. Technically, scalability is making your system easy to change in a particular way, and so is a kind of modifiability. In addition, we address scalability explicitly in Chapter 12.

8.1 Performance General Scenario

A performance scenario begins with an event arriving at the system. Responding correctly to the event requires resources (including time) to be consumed. While this is happening, the system may be simultaneously servicing other events.

Concurrency

Concurrency is one of the more important concepts that an architect must understand and one of the least-taught in computer science courses. Concurrency refers to operations occurring in parallel. For example, suppose there is a thread that executes the statements

```
x := 1;
x++;
```

and another thread that executes the same statements. What is the value of *x* after both threads have executed those statements? It could be either 2 or 3. I leave it to you to figure out how the value 3 could occur—or should I say I interleave it to you?

Concurrency occurs any time your system creates a new thread, because threads, by definition, are independent sequences of control. Multitasking on your system is supported by independent threads. Multiple users are simultaneously supported on your system through the use of threads. Concurrency also occurs any time your system is executing on more than one processor, whether the processors are packaged separately or as multi-core processors. In addition, you must consider concurrency when parallel algorithms, parallelizing infrastructures such as map-reduce, or NoSQL databases are used by your system, or you utilize one of a variety of concurrent scheduling algorithms. In other words, concurrency is a tool available to you in many ways.

Concurrency, when you have multiple CPUs or wait states that can exploit it, is a good thing. Allowing operations to occur in parallel improves performance, because delays introduced in one thread allow the processor

to progress on another thread. But because of the interleaving phenomenon just described (referred to as a *race condition*), concurrency must also be carefully managed by the architect.

As the example shows, race conditions can occur when there are two threads of control and there is shared state. The management of concurrency frequently comes down to managing how state is shared. One technique for preventing race conditions is to use locks to enforce sequential access to state. Another technique is to partition the state based on the thread executing a portion of code. That is, if there are two instances of *x* in our example, *x* is not shared by the two threads and there will not be a race condition.

Race conditions are one of the hardest types of bugs to discover; the occurrence of the bug is sporadic and depends on (possibly minute) differences in timing. I once had a race condition in an operating system that I could not track down. I put a test in the code so that the next time the race condition occurred, a debugging process was triggered. It took over a year for the bug to recur so that the cause could be determined.

Do not let the difficulties associated with concurrency dissuade you from utilizing this very important technique. Just use it with the knowledge that you must carefully identify critical sections in your code and ensure that race conditions will not occur in those sections.

—LB

Events can arrive in predictable patterns or mathematical distributions, or be unpredictable. An arrival pattern for events is characterized as *periodic*, *stochastic*, or *sporadic*:

- Periodic events arrive predictably at regular time intervals. For instance, an event may arrive every 10 milliseconds. Periodic event arrival is most often seen in real-time systems.
- Stochastic arrival means that events arrive according to some probabilistic distribution.
- Sporadic events arrive according to a pattern that is neither periodic nor stochastic. Even these can be characterized, however, in certain circumstances. For example, we might know that at most 600 events will occur in a minute, or that there will be at least 200 milliseconds between the arrival of any two events. (This might describe a system in which events correspond to keyboard strokes from a human user.) These are helpful characterizations, even though we don't know when any single event will arrive.

The response of the system to a stimulus can be measured by the following:

- *Latency.* The time between the arrival of the stimulus and the system's response to it.

- *Deadlines in processing.* In the engine controller, for example, the fuel should ignite when the cylinder is in a particular position, thus introducing a processing deadline.
- The *throughput* of the system, usually given as the number of transactions the system can process in a unit of time.
- The *jitter* of the response—the allowable variation in latency.
- The *number of events not processed* because the system was too busy to respond.

From these considerations we can now describe the individual portions of a general scenario for performance:

- *Source of stimulus.* The stimuli arrive either from external (possibly multiple) or internal sources.
- *Stimulus*. The stimuli are the event arrivals. The arrival pattern can be periodic, stochastic, or sporadic, characterized by numeric parameters.
- *Artifact*. The artifact is the system or one or more of its components.
- *Environment*. The system can be in various operational modes, such as normal, emergency, peak load, or overload.
- *Response*. The system must process the arriving events. This may cause a change in the system environment (e.g., from normal to overload mode).
- *Response measure.* The response measures are the time it takes to process the arriving events (latency or a deadline), the variation in this time (jitter), the number of events that can be processed within a particular time interval (throughput), or a characterization of the events that cannot be processed (miss rate).

The general scenario for performance is summarized in Table 8.1.

Figure 8.1 gives an example concrete performance scenario: Users initiate transactions under normal operations. The system processes the transactions with an average latency of two seconds.

TABLE 8.1 Performance General Scenario

Portion of Scenario	Possible Values
Source	Internal or external to the system
Stimulus	Arrival of a periodic, sporadic, or stochastic event
Artifact	System or one or more components in the system
Environment	Operational mode: normal, emergency, peak load, overload
Response	Process events, change level of service
Response Measure	Latency, deadline, throughput, jitter, miss rate

8.2 Tactics for Performance

The goal of performance tactics is to generate a response to an event arriving at the system within some time-based constraint. The event can be single or a stream and is the trigger to perform computation. Performance tactics control the time within which a response is generated, as illustrated in Figure 8.2.

At any instant during the period after an event arrives but before the system's response t`o it is complete, either the system is working to respond to that event or the processing is blocked for some reason. This leads to the two basic contributors to the response time: processing time (when the system is working to respond) and blocked time (when the system is unable to respond).

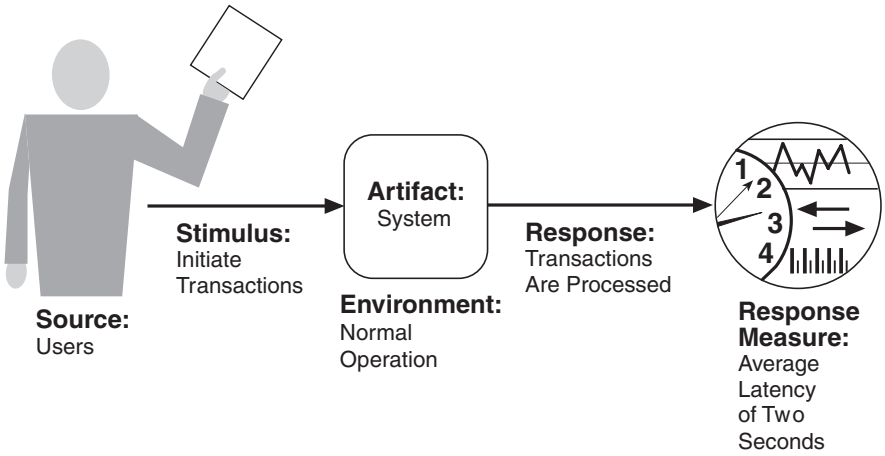

FIGURE 8.1 Sample concrete performance scenario

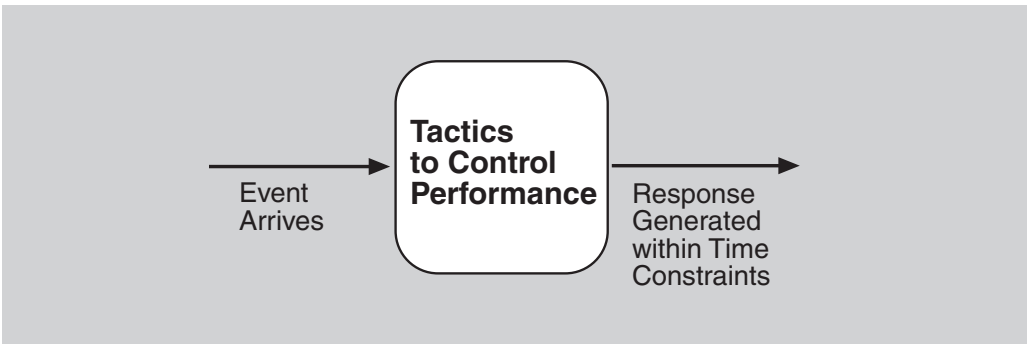

FIGURE 8.2 The goal of performance tactics

- *Processing time.* Processing consumes resources, which takes time. Events are handled by the execution of one or more components, whose time expended is a resource. Hardware resources include CPU, data stores, network communication bandwidth, and memory. Software resources include entities defined by the system under design. For example, buffers must be managed and access to critical sections[1] must be made sequential.

  For example, suppose a message is generated by one component. It might be placed on the network, after which it arrives at another component. It is then placed in a buffer; transformed in some fashion; processed according to some algorithm; transformed for output; placed in an output buffer; and sent onward to another component, another system, or some actor. Each of these steps consumes resources and time and contributes to the overall latency of the processing of that event.

  Different resources behave differently as their utilization approaches their capacity—that is, as they become saturated. For example, as a CPU becomes more heavily loaded, performance usually degrades fairly steadily. On the other hand, when you start to run out of memory, at some point the page swapping becomes overwhelming and performance crashes suddenly.

- *Blocked time.* A computation can be blocked because of contention for some needed resource, because the resource is unavailable, or because the computation depends on the result of other computations that are not yet available:

  - *Contention for resources.* Many resources can only be used by a single client at a time. This means that other clients must wait for access to those resources. Figure 8.2 shows events arriving at the system. These events may be in a single stream or in multiple streams. Multiple streams vying for the same resource or different events in the same stream vying for the same resource contribute to latency. The more contention for a resource, the more likelihood of latency being introduced.
  - *Availability of resources.* Even in the absence of contention, computation cannot proceed if a resource is unavailable. Unavailability may be caused by the resource being offline or by failure of the component or for some other reason. In any case, you must identify places where resource unavailability might cause a significant contribution to overall latency. Some of our tactics are intended to deal with this situation.
  - *Dependency on other computation.* A computation may have to wait because it must synchronize with the results of another computation or because it is waiting for the results of a computation that it initiated. If a component calls another component and must wait for that component to respond, the time can be significant if the called component is at the other end of a network (as opposed to co-located on the same processor).

1. A critical section is a section of code in a multi-threaded system in which at most one thread may be active at any time.

With this background, we turn to our tactic categories. We can either reduce demand for resources or make the resources we have handle the demand more effectively:

- *Control resource demand.* This tactic operates on the demand side to produce smaller demand on the resources that will have to service the events.
- *Manage resources.* This tactic operates on the response side to make the resources at hand work more effectively in handling the demands put to them.

Control Resource Demand

One way to increase performance is to carefully manage the demand for resources. This can be done by reducing the number of events processed by enforcing a sampling rate, or by limiting the rate at which the system responds to events. In addition, there are a number of techniques for ensuring that the resources that you do have are applied judiciously:

- *Manage sampling rate*. If it is possible to reduce the sampling frequency at which a stream of environmental data is captured, then demand can be reduced, typically with some attendant loss of fidelity. This is common in signal processing systems where, for example, different codecs can be chosen with different sampling rates and data formats. This design choice is made to maintain predictable levels of latency; you must decide whether having a lower fidelity but consistent stream of data is preferable to losing packets of data.
- *Limit event response*. When discrete events arrive at the system (or element) too rapidly to be processed, then the events must be queued until they can be processed. Because these events are discrete, it is typically not desirable to "downsample" them. In such a case, you may choose to process events only up to a set maximum rate, thereby ensuring more predictable processing when the events are actually processed. This tactic could be triggered by a queue size or processor utilization measure exceeding some warning level. If you adopt this tactic and it is unacceptable to lose any events, then you must ensure that your queues are large enough to handle the worst case. If, on the other hand, you choose to drop events, then you need to choose a policy for handling this situation: Do you log the dropped events, or simply ignore them? Do you notify other systems, users, or administrators?
- *Prioritize events*. If not all events are equally important, you can impose a priority scheme that ranks events according to how important it is to service them. If there are not enough resources available to service them when they arise, low-priority events might be ignored. Ignoring events consumes minimal resources (including time), and thus increases performance compared to a system that services all events all the time. For example, a building

management system may raise a variety of alarms. Life-threatening alarms such as a fire alarm should be given higher priority than informational alarms such as a room is too cold.

- *Reduce overhead*. The use of intermediaries (so important for modifiability, as we saw in Chapter 7) increases the resources consumed in processing an event stream, and so removing them improves latency. This is a classic modifiability/performance tradeoff. Separation of concerns, another linchpin of modifiability, can also increase the processing overhead necessary to service an event if it leads to an event being serviced by a chain of components rather than a single component. The context switching and intercomponent communication costs add up, especially when the components are on different nodes on a network. A strategy for reducing computational overhead is to co-locate resources. Co-location may mean hosting cooperating components on the same processor to avoid the time delay of network communication; it may mean putting the resources in the same runtime software component to avoid even the expense of a subroutine call. A special case of reducing computational overhead is to perform a periodic cleanup of resources that have become inefficient. For example, hash tables and virtual memory maps may require recalculation and reinitialization. Another common strategy is to execute single-threaded servers (for simplicity and avoiding contention) and split workload across them.
- *Bound execution times*. Place a limit on how much execution time is used to respond to an event. For iterative, data-dependent algorithms, limiting the number of iterations is a method for bounding execution times. The cost is usually a less accurate computation. If you adopt this tactic, you will need to assess its effect on accuracy and see if the result is "good enough." This resource management tactic is frequently paired with the manage sampling rate tactic.
- *Increase resource efficiency.* Improving the algorithms used in critical areas will decrease latency.

Manage Resources

Even if the demand for resources is not controllable, the management of these resources can be. Sometimes one resource can be traded for another. For example, intermediate data may be kept in a cache or it may be regenerated depending on time and space resource availability. This tactic is usually applied to the processor but is also effective when applied to other resources such as a disk. Here are some resource management tactics:

- *Increase resources.* Faster processors, additional processors, additional memory, and faster networks all have the potential for reducing latency.

Cost is usually a consideration in the choice of resources, but increasing the resources is definitely a tactic to reduce latency and in many cases is the cheapest way to get immediate improvement.

- *Introduce concurrency*. If requests can be processed in parallel, the blocked time can be reduced. Concurrency can be introduced by processing different streams of events on different threads or by creating additional threads to process different sets of activities. Once concurrency has been introduced, scheduling policies can be used to achieve the goals you find desirable. Different scheduling policies may maximize fairness (all requests get equal time), throughput (shortest time to finish first), or other goals. (See the sidebar.)
- *Maintain multiple copies of computations*. Multiple servers in a client-server pattern are replicas of computation. The purpose of replicas is to reduce the contention that would occur if all computations took place on a single server. A *load balancer* is a piece of software that assigns new work to one of the available duplicate servers; criteria for assignment vary but can be as simple as round-robin or assigning the next request to the least busy server.
- *Maintain multiple copies of data.* *Caching* is a tactic that involves keeping copies of data (possibly one a subset of the other) on storage with different access speeds. The different access speeds may be inherent (memory versus secondary storage) or may be due to the necessity for network communication. *Data replication* involves keeping separate copies of the data to reduce the contention from multiple simultaneous accesses. Because the data being cached or replicated is usually a copy of existing data, keeping the copies consistent and synchronized becomes a responsibility that the system must assume. Another responsibility is to choose the data to be cached. Some caches operate by merely keeping copies of whatever was recently requested, but it is also possible to predict users' future requests based on patterns of behavior, and begin the calculations or prefetches necessary to comply with those requests before the user has made them.
- *Bound queue sizes*. This controls the maximum number of queued arrivals and consequently the resources used to process the arrivals. If you adopt this tactic, you need to adopt a policy for what happens when the queues overflow and decide if not responding to lost events is acceptable. This tactic is frequently paired with the limit event response tactic.
- *Schedule resources*. Whenever there is contention for a resource, the resource must be scheduled. Processors are scheduled, buffers are scheduled, and networks are scheduled. Your goal is to understand the characteristics of each resource's use and choose the scheduling strategy that is compatible with it. (See the sidebar.)

The tactics for performance are summarized in Figure 8.3.

Scheduling Policies

A *scheduling policy* conceptually has two parts: a priority assignment and dispatching. All scheduling policies assign priorities. In some cases the assignment is as simple as first-in/first-out (or FIFO). In other cases, it can be tied to the deadline of the request or its semantic importance. Competing criteria for scheduling include optimal resource usage, request importance, minimizing the number of resources used, minimizing latency, maximizing throughput, preventing starvation to ensure fairness, and so forth. You need to be aware of these possibly conflicting criteria and the effect that the chosen tactic has on meeting them.

A high-priority event stream can be dispatched only if the resource to which it is being assigned is available. Sometimes this depends on preempting the current user of the resource. Possible preemption options are as follows: can occur anytime, can occur only at specific preemption points, and executing processes cannot be preempted. Some common scheduling policies are these:

- *First-in/first-out.* FIFO queues treat all requests for resources as equals and satisfy them in turn. One possibility with a FIFO queue is that one request will be stuck behind another one that takes a long time to generate a response. As long as all of the requests are truly equal, this is not a problem, but if some requests are of higher priority than others, it is problematic.
- *Fixed-priority scheduling.* Fixed-priority scheduling assigns each source of resource requests a particular priority and assigns the resources in that priority order. This strategy ensures better service for higher priority requests. But it admits the possibility of a lower priority, but important, request taking an arbitrarily long time to be serviced, because it is stuck behind a series of higher priority requests. Three common prioritization strategies are these:
  - *Semantic importance.* Each stream is assigned a priority statically according to some domain characteristic of the task that generates it.
  - *Deadline monotonic.* Deadline monotonic. Deadline monotonic is a static priority assignment that assigns higher priority to streams with shorter deadlines. This scheduling policy is used when streams of different priorities with real-time deadlines are to be scheduled.
  - *Rate monotonic.* Rate monotonic is a static priority assignment for periodic streams that assigns higher priority to streams with shorter periods. This scheduling policy is a special case of deadline monotonic but is better known and more likely to be supported by the operating system.
- *Dynamic priority scheduling.* Strategies include these:
  - *Round-robin.* Round-robin is a scheduling strategy that orders the requests and then, at every assignment possibility, assigns the resource to the next request in that order. A special form of

round-robin is a cyclic executive, where assignment possibilities are at fixed time intervals.

- *Earliest-deadline-first.* Earliest-deadline-first. Earliest-deadline-first assigns priorities based on the pending requests with the earliest deadline.
- *Least-slack-first.* This strategy assigns the highest priority to the job having the least "slack time," which is the difference between the execution time remaining and the time to the job's deadline.

For a single processor and processes that are preemptible (that is, it is possible to suspend processing of one task in order to service a task whose deadline is drawing near), both the earliest-deadline and least-slack scheduling strategies are optimal. That is, if the set of processes can be scheduled so that all deadlines are met, then these strategies will be able to schedule that set successfully.

- *Static scheduling.* A cyclic executive schedule is a scheduling strategy where the preemption points and the sequence of assignment to the resource are determined offline. The runtime overhead of a scheduler is thereby obviated.

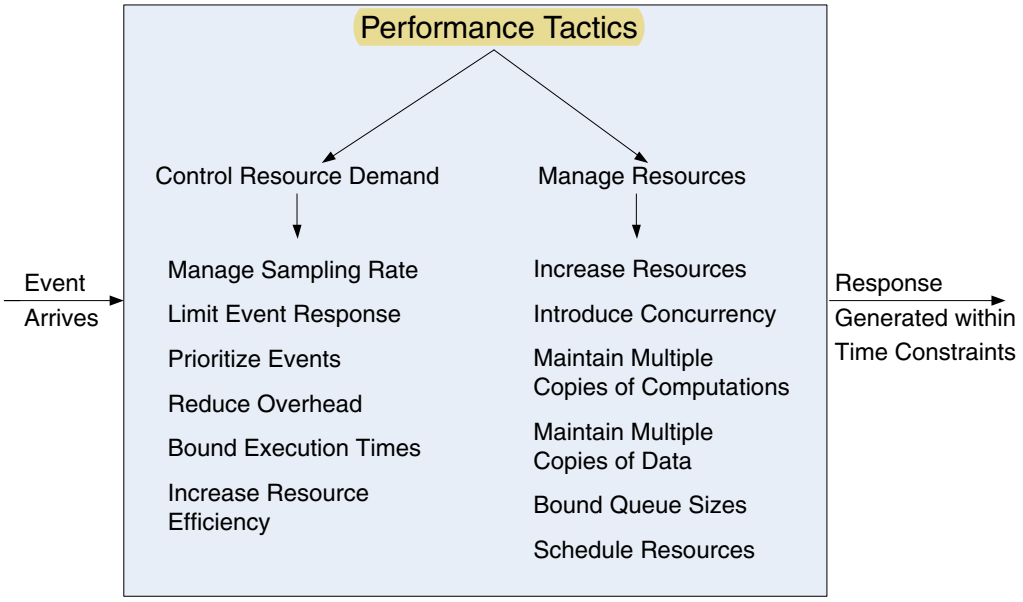

FIGURE 8.3 Performance tactics

Performance Tactics on the Road

Tactics are generic design principles. To exercise this point, think about the design of the systems of roads and highways where you live. Traffic engineers employ a bunch of design "tricks" to optimize the performance of these complex systems, where performance has a number of measures, such as throughput (how many cars per hour get from the suburbs to the football stadium), average-case latency (how long it takes, on average, to get from your house to downtown), and worst-case latency (how long does it take an emergency vehicle to get you to the hospital). What are these tricks? None other than our good old buddies, tactics.

Let's consider some examples:

- *Manage event rate.* Lights on highway entrance ramps let cars onto the highway only at set intervals, and cars must wait (queue) on the ramp for their turn.
- *Prioritize events.* Ambulances and police, with their lights and sirens going, have higher priority than ordinary citizens; some highways have high-occupancy vehicle (HOV) lanes, giving priority to vehicles with two or more occupants.
- *Maintain multiple copies.* Add traffic lanes to existing roads, or build parallel routes.

In addition, there are some tricks that users of the system can employ:

- *Increase resources.* Buy a Ferrari, for example. All other things being equal, the fastest car with a competent driver on an open road will get you to your destination more quickly.
- *Increase efficiency.* Find a new route that is quicker and/or shorter than your current route.
- *Reduce computational overhead.* You can drive closer to the car in front of you, or you can load more people into the same vehicle (that is, carpooling).

What is the point of this discussion? To paraphrase Gertrude Stein: performance is performance is performance. Engineers have been analyzing and optimizing systems for centuries, trying to improve their performance, and they have been employing the same design strategies to do so. So you should feel some comfort in knowing that when you try to improve the performance of your computer-based system, you are applying tactics that have been thoroughly "road tested."

—RK

8.3 A Design Checklist for Performance

Table 8.2 is a checklist to support the design and analysis process for performance.