Building Successful Performance-Critical Software Systems

Dr. C. Douglass Locke
Locke Consulting, LLC
doug@doug-locke.com
www.doug-locke.com
412-977-6003

February 11, 2004

Performance-critical systems have been built for many years, but for every successful system, there are two to three unsuccessful systems. Even now, when processor speed, memory size, and communications bandwidth are at levels undreamed of only a few years ago, performance-critical systems provide major challenges as we attempt to master them.

Several questions must be answered.

- What do we mean by "performance-critical" systems?
- Why are performance-critical systems difficult to design and build?
- How can we predict the performance of performance-critical systems?
- What pitfalls to successful performance-critical design are most common, and how can we address them?
- What architectural implications can be drawn from the performance requirements for such systems?

What do we mean by "performance-critical" systems?

A performance-critical system is one in which one or more performance attributes must be achieved for the system to be successful. A performance-critical system might, for example, need to meet any or all of the following requirements:

- Respond to a critical message within 100 microseconds on average, with a worst-case response of 500 microseconds.
- Update the positions of a set up to 1000 tracks (a track might be a radar blip representing a vehicle) every second, without falling behind.
- Detect a hardware failure within 20 milliseconds and reconfigure the network within 20 milliseconds around the failure.
- Begin full operation from power-up within 300 millisconds.

Notice several things about this sample list of requirements. First, none of these performance requirements are *primary* requirements; they are all *secondary* or *derived* requirements. This means that they may not have even been clearly articulated when the system architecture and design began, but may have been exposed only when the system was partially completed.

Second, it is not obvious whether any of them are "hard" requirements. A hard performance requirement is one that must always be met in a non-failed system. A soft requirement can be missed occasionally, or by a small amount without considering the system to have failed.

Third, all of these requirements are incomplete. This is quite normal in systems requirements, and is one of the major reasons for system failure later during integration, system test, and deployment. For example, for the second requirement, is there a system limit on the number of tracks? If not, what is the expected behavior if the number exceeds 1000? Is it acceptable to crash at 1001 tracks? For another example, what are the arrival characteristics of the critical messages? Are they periodic? Are they exponentially distributed? Is there a conflict between and critical messages and tracks:

e.g., what is the implication on the critical message requirement if the number of tracks exceeds 1000? Which requirement is most important?

Why are performance-critical systems difficult to design and build?

The unanswered questions of the preceding section provide a hint as to why performance-critical systems are so difficult to design and build. The first problem is requirements definition. Most system requirements either ignore performance, or reflect only very high-level performance requirements. Referring to our previous example, the requirements probably have great detail about the computations needed when a critical message arrives, how track positions and properties are to be managed and computed, the need to recover from failures without stopping the system, and what the sequence of events must be at power-up. The performance characteristics are frequently left to the imagination of the developer, and remain incomplete until final test and deployment.

Even when the performance requirements are carefully defined, it is difficult to design a solution that will meet them. For example, the software must use concurrency to meet conflicting requirements, but few software professionals are comfortable designing software with concurrent threads of control. Priorities are most likely the only mechanism available to control system resource management and performance, but it's frequently not obvious how the priorities should be determined, or what the consequences are if they are not correct. Most systems and software professionals are aware of technologies such as rate-monotonic scheduling, but few fully understand their capabilities and limitations for performance-critical systems.

Another difficulty in performance-critical system design is adapting software methodologies not originally designed for performance-critical systems. For example, there are special problems using Object-Oriented (OO) methodologies in performance-critical systems. Windowing environments force a "callback" design methodology in which concurrency is frequently difficult to control. Use of a database introduces sequentiality in data accesses that prevents performance requirements from being met.

How can we predict the performance of performance-critical systems?

Predicting system performance is a very difficult problem whose solutions depend significantly on their design characteristics. Are the system input arrival patterns fairly static, or very dynamic? Are all computations known in advance, or will new computations arrive that were not anticipated at system design time?

Systems characterized primarily by static structures and input arrivals can frequently use rate-monotonic or deadline monotonic analysis to make their response times highly predictable. More dynamic systems will usually need to mix these kinds of analytical techniques with discrete-event simulation. Simulation is much more difficult to do because of the need to determine all the critical activities, accurately estimate their resource utilization, create a wide variety of input scenarios (i.e., use cases), and validate the results.

What pitfalls to successful performance-critical design are most common, and how can we address them?

A few of the most common pitfalls can be described simply. Probably the most common is to design the system assuming that the performance requirements will be met just because the processor and network speed, as well as memory and storage capacities, are much greater than preceding systems. This is wrong for two reasons – to overcome performance requirements without using specific performance-critical components, the overall performance would need to increase by 3-4 *orders of magnitude*, not by the usual 20-50% achieved with each roll of the technology.

The second most common pitfall is to use infrastructure elements that were designed only for throughput when response times are the critical issue. This fails because non-performance-critical components do not manage resources in a way that can produce predictable response time. It is not widely understood that the techniques used to optimize throughput are completely inappropriate for optimizing response time. For example, bounding response time when using a windowing environment can be done, but only by very careful planning and implementation, with close attention to concurrency.

What architectural implications can be drawn from the performance requirements for such systems?

The architecture of any system can be described as the highest level of design of the system. The architecture consists of a composition of a set of the major components, including both hardware and software, that will define the system. The choice of these components, or objects, and their interactions, is the most critical determinant of the performance of the resulting system.

To successfully construct a performance-critical system, many questions must be asked and answered. What performance requirements are met within each component? Which are met only by multiple components jointly? Where is information created, stored, modified, and communicated? What are the consequences of these choices? What synchronization is implied by the concurrency solutions proposed in an architecture? Should concurrency be visible, or hidden? Should synchronization be visible or encapsulated?

Conclusions

This brief paper cannot define solutions for all the critical issues for performance-critical systems, but it is intended to identify some of them so they can be addressed.

The actions that should be taken to enhance the probability of success for performance-critical systems depends on how far into the lifecycle the project has come. For systems in the requirements-generation phase, these answers to these questions can provide major help in ensuring that all the crucial requirements are considered and elucidated. For systems undergoing architecture creation or high-level design, these answers to these questions can help to evaluate the initial component and interaction choices. For systems in later development stages that are experiencing performance problems, these questions will probably point to issues that can be investigated as early as possible to solve or prevent the existing problems.

For systems in every development or deployment stage, a review of performance requirements and solutions can be performed. This can be done as a single review, or a continuing series of reviews throughout the system life-cycle. The latter has proven to be the most effective, especially for complex systems.