Incorporating Software Performance Engineering Methods and Practices into the Software Development Life Cycle

André B. Bondi
Red Bank, New Jersey
bondia@acm.org

ABSTRACT

In many software development projects, attention is only paid to performance concerns after functional testing, when it usually too late to remedy disabling performance problems. Early attention to performance concerns and early planning of performance requirements and performance testing can prevent debacles like the early rollout of healthcare.gov while addressing cross-cutting concerns such as scalability, reliability and, security. Performance engineering methods may be integrated into all phases of the software lifecycle, from the conception of a system to requirements specification, architecture, testing, and finally to production. Performance expectations can be managed by carefully specifying performance requirements. Reviewing the architecture of a system before design and implementation take place reduces the risk of designing a system that contains inherent performance vice. Performance modeling can be used to justify architectural and design decisions and to plan performance tests. The outputs of such performance tests enable us able to identify concurrent programming and other issues that would not be apparent in unit testing. Finally, risk is mitigated by avoiding design antipatterns that undermine scalability and performance.

General Terms

Performance; Measurement; Software life cycle; Architecture

Keywords

Software performance engineering; Performance measurement and testing; Software life cycle; Modeling; Architecture

1. INTRODUCTION

Historically, performance concerns about a software system have often only been addressed when system is close to being delivered for production. In a keynote speech at SIGMETRICS 1981, J. C. Browne commented that performance evaluation was usually carried out in repairman mode, i.e., when the system is in production, rather than being part of the software design process [5]. More recently, Bass *et al* presented the results of a survey showing that performance was the single biggest risk factor affecting performance [WICSA2007]. Even though Smith and Williams have emphasized that "Build it, then tune it" is a mindset that makes performance failure almost inevitable [11], there are well known cases of systems exhibiting poor performance to the point of being all but unusable. The 2013 rollout of healthcare.gov, the US government's web site for applying for health insurance, is an example. According to press reports, the demand and performance requirements of healthcare.gov were not understood, and little time was allowed for performance testing [7].

We advocate that performance concerns be addressed from project inception to delivery, and give an overview of how performance engineering methods can be incorporated into various stages of the software development life cycle. This may be opposed by stakeholders on the ground that it takes time away from feature delivery, or that there is not enough staff time available for the purpose, among other reasons. It may be resisted by product managers who are reluctant to commit to a "performance number" because different market segments may have different performance needs.

Early involvement of a performance engineer and early use of performance engineering methods are essential to the mitigation of the business and engineering risks inherent in any large performance engineering project. The goal is to identify and address performance concerns early so as to reduce the risk of having to redo work. Early application and adoption of performance engineering practices provides insurance against the penalties, costs, and lost revenue associated with rework and late delivery.

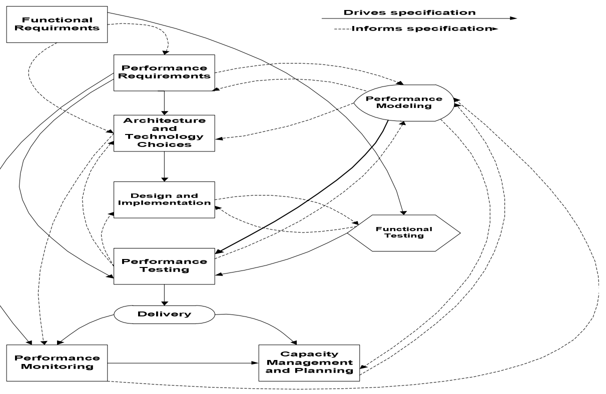

Figure 1. A performance engineering process and its relationship to a development process.

Figure 1 depicts the relationship between the steps of a performance engineering process and corresponding steps in a software development process. The functional requirements drive the form that functional testing will take. They also inform the nature of the performance requirements. The performance requirements describe how often various functions are executed, how long they will take, and the corresponding memory and secondary storage requirements. Performance requirements and performance models inform decisions about the system architecture and the technology platforms to be used as well as the planning of performance tests. When the system architecture is being mapped out, care must be taken to ensure that it will be able to meet disparate performance requirements, including those intended to meet regulatory and safety needs. This is a point at which the tradeoffs between performance and other cross-cutting concerns such as security should be identified. A performance model can help determine whether performance and security requirements and the costs of meeting them are compatible.

Performance testing usually occurs after functional testing, since there is little point in running full performance tests on a system or component that is known not to function properly. Just as functional testing is driven by functional requirements, performance test planning should be driven by performance requirements, knowledge of anticipated workloads, and the desired characteristics of performance curves. Among these characteristics is linearity of the average hardware utilizations with respect to the offered load [6]. The use of performance models to predict the capacity of a system in production is described in [6] and will not be discussed here.

2. PERFORMANCE ISSUES AT EACH LIFE CYCLE PHASE

The absence of or lack of clarity in performance requirements is a cause of unmet or unrealistic performance expectations. It is difficult to architect a system without understanding the performance requirements of various functions and the ability of various technologies to meet them. Even if performance requirements are soundly specified, poor architectural choices can undermine performance. Choices of deployment scenarios can induce foci of overloads or antipatterns such as god classes [11]. The scalability of a system might be impaired by the use of single threading when multithreaded operations could exploit multiple cores or processors. These architectural choices would be propagated into the development phase, where there are further possibilities to use performance antipatterns and poorly performing algorithms. Poorly designed performance tests might fail to reveal performance issues. Performance tests might be driven by workloads that are too small, resulting in an optimistic view of the capabilities of the system, or by workloads that are too large, with the opposite effect. Finally, lack of capacity or poor configuration in production could lead to unsatisfactory performance.

2.1 Performance Requirements

By linking performance requirements to functional requirements and to business engineering needs, one ensures that they are traceable and not superfluous. By linking them to other cross-cutting concerns such as security [10], one is driven to explore the tradeoffs between the concerns and to modify the system architecture accordingly. By insisting that performance requirements be written in measurable testable terms, we are forced to verify that instrumentation is there to provide the verification and that the performance requirements are meaningful within the context of the domain. The guidelines we suggest for writing sound performance requirements are quite similar to those for functional requirements in IEEE Standard 830 [8]. Like functional requirements, performance requirements should be traceable, so that we know why each one is there. They should be based on a well-defined and explicitly stated set of assumptions. They should be verifiable. They should also be numerically consistent. If two or more performance requirements lead to numerical inconsistency or contradict each other, the cause should be investigated before the performance requirements document is released following review. Performance modeling methods can be used to demonstrate numerical consistency or inconsistency between performance requirements. The demonstrations should be mentioned in the performance requirements documents in support of verifiability and traceability, and, in the case of inconsistency, as triggers for correction.

2.2 Performance Engineering and Architecture

An architect should have an overview of cross-cutting concerns such as performance, reliability, scalability, availability, security, and the choices of software and hardware platforms. Architects and performance engineers can jointly assess the performance requirements to determine scalability needs and identify platform choices while bearing cross cutting concerns in mind. A performance engineer can support an architect's efforts by drawing attention to performance pitfalls during architectural reviews, clarifying performance requirements, and suggesting scheduling rules and other design choices to help meet them.

Even if not much is known with certainty about a system that does not yet exist (e.g., because performance data is lacking), a performance engineer's experience about software performance pitfalls and the properties of queueing systems and scheduling rules can be used to great effect to mitigate performance risks at this stage of the development cycle. When technological choices are being discussed, a performance engineer can have a significant impact by asking about their known performance characteristics and about how often they will be invoked. In cases of doubt, it may be prudent to defend against the possibility of a platform being too slow by recommending early performance testing and benchmarking [2], [9].

The following are among the questions that should be asked during architecture reviews to support performance needs:

- What is the end-to-end flow of information?
- What technologies and parts of the system pose the biggest risk to performance?
- Are there any object pools or other passive software objects such as lock that could become software bottlenecks?
- Does the design contain any performance antipatterns that could cause performance to degrade or that could impede scalability?
- What are the potential foci of overload?
- Can the chosen platforms handle the required actions at the desired rates and with turnaround times that are low enough to meet performance requirements?
- Is any part of the system single-threaded when multithreading could be used to exploit multiple CPUs?
- Can priority scheduling be supported if needed? Answering this question may require an understanding of the data structures and protocols that are needed to implement the system.

Rules of thumb can be used to identify performance pitfalls and antipatterns. God classes and foci of overload can be spotted by reviewing UML message sequence charts, activity and collaboration diagrams, and deployment scenarios. If a message sequence diagram shows that at least one swim lane has large numbers of arrows pointing into it or coming out of it, the corresponding object is likely to be focus of overload, a god class, or both. One-lane bridges and scheduling rules such as the museum checkroom pattern leading to deadlock [3] may be harder to spot, but the benefit of identifying them is the reduced risk of concurrent programming problems that are hard to diagnose.

2.3 Performance Engineering in Design and Implementation

The approach to taking performance considerations into account at the design and implementation stage is similar to that taken in the architectural phase, except that the focus will be on finer details than would be considered when discussing the architecture. As with the architecture, we recommend that a design review be done while keeping the risks of various performance antipatterns in mind. The implementation should avoid the use of busy waiting to implement mutual exclusion and synchronization. Scheduling rules must be free of dependencies that cause deadlocks and livelocks. A review of the implementation should flag and other activities such as insertion sorts whose processing costs are at least polynomial in the number of items involved.

2.4 Performance Testing

Functional testing can be leveraged to support performance testing, because performance tests involve the repeated invocation of the use cases exercised in the functional tests in a controlled and predictable manner. Finally, performance and resource usage monitoring in production can be combined with performance models to identify areas for performance improvement. Of course, our emphasis should be on early performance intervention rather than on measurement in production, but the latter is necessary to verify the effectiveness of and correct defects in the former.

Performance testing usually occurs towards the end of the development cycle, after functional testing. This is understandable, because performance tests of a system that does not meet functional requirements may not tell us accurately about the performance of a system that does meet them. Where an application is built on services and other building blocks, it is often useful to test those before they are built into applications. Thus, there are distinct phases of the development cycle during which performance testing is useful for containing engineering risk.

- To contain the performance risk inherent in a choice of hardware platform, software platform, programming environment, or operating system, one may subject it to synthetic loads that exercise basic functions that will be invoked frequently. This reduces the risk of building a system on the platform, only to find just before delivery that the platform is inherently unable to execute the basic functions fast enough to meet performance requirements [2], [8].
- Performance tests of the implementation of a system or of a part of the system should be done after functional testing, so that time is not wasted on testing the performance of a system that does not work. One can test the use cases of each service it is implemented. One should also test the system with multiple use cases being exercised concurrently as they would be in production.

Notice that functional tests will not usually expose concurrent programming errors, as these are usually unit tests done in single user mode. Performance tests may reveal concurrent programming errors. These can be manifested as deadlocks, livelocks, or thread safety and divisibility errors. Symptoms of thread safety and divisibility errors include the frequent transaction failures and the frequently reattempted actions, and corrupted data. Symptoms of deadlock include sudden drops in CPU utilizations and average response times that oscillate wildly over time. Symptoms of software bottlenecks include response times that grow as functions of the offered load and over time while CPU and device utilizations level off as functions of the offered load [4], [1].

Performance tests should be structured to reveal trends in utilizations, response times, and domain-related indicators such as transaction failure rates and transaction success rates. Three basic goals should be met by a performance test plan. First, one should be able to verify that a system operating under constant load will have constant average performance measures and resource utilizations between load ramp up and load ramp down. Second, one should run tests with at least three load levels to verify that utilizations are linear with respect to the offered load rate or transaction arrival rate, i.e., that the Utilization Law is satisfied. Third, one should determine that performance requirements are met by the system under test. Performance tests should be run long enough under a constant load to demonstrate that the system has reached equilibrium. Statistics on memory usage should be collected to enable the detection of memory leaks. Stress tests in which the system is subjected to a load large enough to cause a system to crash or nearly crash are of limited utility, because they reveal nothing about utilization trends and so cannot be used to predict load levels at which one or more components of the system will be saturated. They will only tell us if the system is capable of maintaining normal activity under a saturating load and of recovering from a crash once the intense load has abated. In our structured tests, if at least one of the hardware utilizations approaches 100% as the load is increased and all utilizations rise in constant ratios as predicted by the Forced Flow Law [6], we may be confident that a software bottleneck has not manifested itself. The range of transaction arrival rates and the nature of the transactions tested should be based on performance requirements, the anticipated size of the user base, as well as on functional requirements and use case specifications. A transaction rate should not be offered in a test if a modeling prediction shows that it would saturate the system.

Of necessity, the performance test lab may be a small version of a target production system that is not architecturally representative of it. This means that bottlenecks may arise in testing that might not arise in production or vice versa. To mitigate this risk, the performance test lab should be built in the light of how a system might be scaled up or down to meet the needs of various market segments while still meeting performance requirements and architectural needs.

We have found it worthwhile to conduct performance tests of the individual services of a service-oriented architecture before building and testing the applications that use them. This sort of early testing reduces the risk of having to diagnose the causes of performance problems and reworking the applications to use other services instead [1], [4].

3. SCALABILITY AND DESIGN CHOICES

Let us briefly attempt to describe our understanding of scalability and then provide illustrations of how it can be examined at the design stage. Load scalability is "the ability of a system to function gracefully, that is without undue delay and without unproductive resource consumption or resource contention, at light, moderate, or heavy loads, while making good use of available resources." A system "has space scalability if its memory requirements do not grow to intolerable levels as the number of items it supports increases." A system "may be said to have structural scalability if its implementation or standards do not impede the growth of the number of objects it encompasses, or … will not do so within a chosen time frame [3]."

Our definition of structural scalability was a precursor of a connection between scalability and performance requirements, namely that the performance requirements describe the dimensions, context and the extent of scalability that the system must support. For example, a small version of a system might be required to provide the processing power, secondary storage, memory, storage, bandwidth, and software needed to meet the performance requirements of ten users, while a large version of a system must provide them to meet the performance requirements of 100 users doing the same sort of work. Here, the dimension of scalability is the number of users, the context is the type of work they do, while the extent is ten users for the small system and 100 for the large one.

Impediments to load scalability include the occurrence of unproductive cycles (as in the case of busy waiting on locks to implement mutual exclusion rather than semaphores) and the inability to exploit parallel processing power because of serial or single-threaded processing of transactions that use disjoint data. Structural scalability can be constrained by the sizes of address spaces or of fixed-size sequence numbers. For example, the length of an array is constrained by the maximum bit length of its integer index. Questions about the impact of a design choice on load scalability can sometimes be answered by applying a simple analytic model over a wide variety of parameters. We have done this to justify the use of semaphores rather than locking instructions 1 implement mutual exclusion [3].

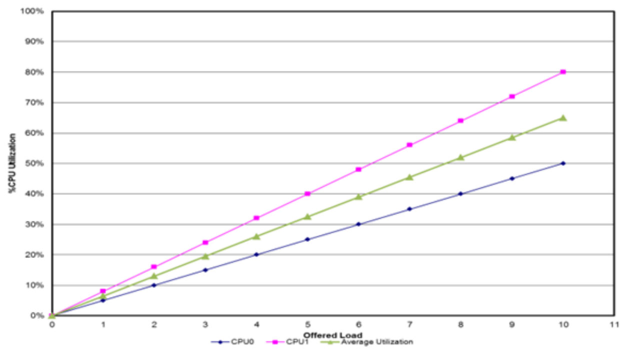

Figure 2. Unbalanced CPU loads due to serial execution via single thread.

Measurements taken during performance tests can also reveal limitations on load scalability [4]. Figure 2 illustrates contrived data representing measurements of a two-processor UNIX™-based system. The data are contrived because measurements of the actual system were not available for publication. The straight line between the upper and lower lines shows the average CPU utilization overall. Observations of the actual system were taken with *mpstat.* An examination of *ps –eLF* output taken at regular intervals revealed that activity was concentrated in two processes for which changes in the cumulative processing times corresponded to the two processor utilizations. Since cache affinity was turned on, we inferred that each process was bound to one processor, yielding test results like those shown in Figure 2. The developers explained that the more CPU-intensive process was processing disjoint sets of data sequentially. An opportunity for parallel execution had been overlooked. The maximum offered load of the system was constrained by the larger of the two CPU utilizations. Thus, the processor imbalance limited the extent of load scalability to 10 work units in unit time, while architecting the system to allow parallel execution on disjoint data sets could have increased the extent of load scalability to 12.3 units of work in unit time, in the absence of any other bottlenecks.

4. CONCLUSION

The foregoing discussion and examples illustrate how performance engineering methods can be applied to mitigate performance risk throughout the software life cycle. Clearly specified performance requirements, performance-oriented architecture reviews, and performance tests planned in the light of requirements all contribute to the mitigation of performance risk and the delivery of a product that meets performance expectations.

5. ACKNOWLEDGMENTS

The figures in this paper originally appeared in [4]. They are reproduced with the permission of the publisher. The author has benefited from frequent discussions with Alberto Avritzer and Bob Schwanke.

6. REFERENCES

[1] Avritzer, A., and A. B. Bondi. 2012. Resilience assessment based on performance testing. In *Resilience Assessment and Evaluation of Computing Systems*, edited by K. Wolter, A. Avritzer, M .Vieira, and A. van Morsel. Springer.

[2] Avritzer, A., and E. Weyuker. 1995. The automatic generation of load test suites and the assessment of the resulting software. *IEEE Trans. Softw. Eng.* 21(9), 705–716.

[3] Bondi, A. B. 2000. Characteristics of scalability and their impact on performance. In *Proc. WOSP2000,* Ottawa.

[4] Bondi, A. B. 2014. *Foundations of Software and System Performance Engineering.* Addison-Wesley, Upper Saddle River, NJ. ISBN-13: 978-0321-83382-2.

[5] Browne, J.C. 1981. Designing systems for performance. Keynote address, ACM SIGMETRICS Conference, Las Vegas, Nevada, 1981. In *Performance Evaluation Review* 10 (1), 1, 1981.

[6] Denning, P. J., and J. P. Buzen. 1978. The operational analysis of queueing network models. *ACM Computing Surveys* 10(3), 225–261.

[7] Eilperin, J.2013. CGI warned of HealthCare.gov problems a month before launch, documents show. *Washington Post,* October 29, 2013.

[8] IEEE Std 830-1998, *IEEE Recommended Practice for Software Requirements Specifications -Description.*

[9] Masticola, S., A. B. Bondi, and M. Hettich. 2005. Model-based scalability estimation in inception-phase software architecture. In *Model Driven Engineering Languages and Systems, Lecture Notes in Computer Science* 3713, 355–366.

[10] Schwaninger, C., Wuchner, E., and Kircher, M. 2004. Encapsulating crosscutting concerns in system software. Proc. Third AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software.

[11] Smith, C.U., and Williams, L.G. 2002. *Performance Solutions.* Addison Wesley, Boston, MA.