Performance Validation at Early Stages of Software Development

Connie U. Smith*, Murray Woodside**

**Performance Engineering Services, Santa Fe, USA (www.perfeng.com)*
*** Carleton University, Ottawa K1S 5B6, Canada (cmw@sce.carleton.ca)*

Abstract

We consider what aspects of software performance can be validated during the early stages of development, before the system is fully implemented, and how this can be approached. There are mature and successful methods available for immediate use, but there are also difficult aspects that need further research. Ease of use and integration of performance engineering with software development are important examples. This paper describes issues in early performance validation, methods, successes and difficulties, and conclusions.

Keywords: Software performance, Requirements, Responsiveness, Workload analysis, Resource budgets.

---

1 Early Performance Validation

The IEEE defines performance and validation as follows:

***Performance*:** the degree to which a system or component accomplishes its designated functions within given constraints, such as speed, accuracy, or memory usage.

Validation: The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.

This paper focuses on the validation of the responsiveness of systems: response time, throughput, and compliance with resource usage constraints. We consider the particular issues in the early development stages (concept, requirements, and design): in pre-implementation stages complete validation is impossible because measurements of the final system are not yet available. The techniques in pre-implementation stages require construction and evaluation of models of the anticipated performance of the final system.

The result is a *model-based approach*. It is not perfect; the following problems must be addressed:

- In pre-implementation stages factual information is limited: final software plans have not been formulated, actual resource usage can only be estimated, and workload characteristics must be anticipated.
- The large number of uncertainties introduce the risk of model omissions: models only reflect what you know to model, and the omissions may have serious performance consequences.
- Thorough modeling studies may require extensive effort to study the many variations of operational scenarios possible in the final system.

- Models are not universal: different types of system assessments require particular types of models. For example the models of typical response time are different from models to assess reliability, fault tolerance, performability, or safety.

Thus, the model-based approach is not a perfect solution, but it is effective at risk reduction.

The modeling techniques must be supplemented with a *performance engineering* process (SPE) that includes techniques for mitigating these problems (Smith, [11].) SPE includes the model-based approach to performance validation as well as techniques for gathering data, prediction strategies, management of uncertainties, techniques for model validation, principles for creating responsive systems, and critical success factors.

The goal of the SPE process and the model-based approach is to reduce the risk of performance failures (rather than guarantee that they will not occur). They increase the confidence in the feasibility of achieving performance objectives and in the architecture and design choices made in early life cycle stages. They provide the following information about the new system:

- refinement and clarification of the performance requirements
- predictions of performance with precision matching the software knowledge available in the early development stage and the quality of resource usage estimates available at that time
- estimates of the sensitivity of the predictions to the accuracy of the resource usage estimates and workload intensity
- understanding of the quantitative impact of design alternatives, that is the effect of system changes on performance
- scalability of the architecture and design: the effect of future growth on performance
- identification of critical parts of the design
- identification of assumptions that, if violated, could change the assessment
- assistance for budgeting resource demands for parts of the design
- assistance in designing performance tests

The remainder of this paper covers the steps in the model-based approach for the quantitative assessment of performance characteristics of an evolving system in early development stages. It addresses the steps in the engineering process, and for each reviews methods used, experience with them, difficulties encountered, and research issues. The final section offers some conclusions on the current state of the art and state of the practice.

2 Validation for Responsiveness and Throughput

The general approach to early validation of performance is similar to any other engineering design evaluation. It is based on evaluating a model of the design, and has five steps:

1. Capture performance requirements, and understand the system functions and rates of operation,
2. Understand the structure of the system and develop a model which is a performance abstraction of the system.
3. Capture the resource requirements and insert them as model parameters
4. Solve the model and compare the results to the requirements.
5. Follow-up: interpret the predictions to suggest changes to aspects that fail to meet performance requirements.

Validation (following the definition above) comes in Step 4 from comparing the evaluation to the requirements. Step 5 creates a loop which is entered if it is necessary to improve a system so it will pass the validation.

Each step has a variety of approaches, which pose characteristic problems that have to be solved by the practitioner in each project, and which also may be problems for research. The differences between approaches taken by the present authors and others lies in the state of the design to be evaluated, the methods used to carry out the steps, and the kind of model used.

2.1 Methods, Successes and Difficulties

Each step will be considered in turn, to discuss how it can be carried out and whether there are gaps in the state of the art.

Step 1. Capture performance requirements

Sometimes the requirements are clear, as in embedded systems where factors in the outer system determine deadlines for certain types of responses. In systems with human users they are rarely clear, because satisfaction is a moving target. As systems get better, users demand more of them, and hope for faster responses.

In most systems there are several types of response, with different requirements. Users can be asked to describe performance requirements, to identify the types and the acceptable delays and capacity limitations. An experienced analyst must review their responses with the users, because they may be unnecessarily ambitious (and thus potentially costly), or else not testable. This review may have to be repeated as the evaluation proceeds and the potential costs of achieving the goals becomes clearer.

Experience in obtaining requirements is not well documented, however it seems clear that they are often not obtained in any depth, or checked for realism, consistency and completeness, and that this is the source of many performance troubles with products. One problem is that users expect the system to modify how they work, so they do not really know precisely how they will use it. Another is that while developers have a good notion of what constitutes well-defined functional requirements they are often naive about performance requirements.

As also for other non-functional requirements, practices for obtaining performance requirements are poorly developed.

Research is needed on questions such as tests for realism, testability, completeness and consistency of performance requirements; on methodology for capturing them, preferably in the context of a standard software notation such as UML, and on the construction of performance tests from the requirements.

Step 2. Understanding the system and building a model:

In this step an execution model is created, representing the functions the software must perform, and the design intentions. It traces scenarios through typical execution paths (a path model in the terms of [16]), and it may take different forms.

- The execution graphs of [11] represent the sequence of operations, including precedence, looping, choices, and forking/joining of flows. This is representative of a large family of models such as SP [14], task graphs (widely used in embedded and hard-real-time systems), activity diagrams, etc. This form has been most useful for capturing scenarios which represent different types of response. Automated tools such as SPEED [12] and HIT [2] capture the data and reduce it to formal workload parameters.
- Communicating state machines are captured in software design languages such as UML, SDL and ROOM, and in many CASE tools, to capture sequence. Some efforts have been made to capture performance data in this way ([18], [5]) by capturing scenarios out of the

operation of the state machines, or [13] by annotating the state machine and simulating the behaviour directly.

- Petri nets and stochastic process algebras are closely related to state machines and also represent behaviour through global states and transitions. Solutions are by Markov chain analysis in the state space, or by simulation.
- Annotated code represents the behaviour by a code skeleton, with abstract elements to represent decisions and operations, and performance annotations to convey parameters. The code may be executed on a real environment, in which case we may call it a performance prototype, or by a simulator (a performance simulation) [1], or it may be reduced to formal workload parameters [8].
- Component-based system assembly models, with components whose internal behaviour is well understood, has been used to create simulation models automatically in SES Strategizer (documented at www.ses.com), and to create analytic layered queueing models [10] [17].

A follow-on to each of the above approaches to describing the workload behaviour, is the creation of a performance evaluation model which is normally different from the behaviour model. Demands captured in an execution graph can be converted into parameters of a queueing model or a layered queueing model. State-based models can be converted into a Markov or Semi-Markov Chain, for numerical solution. Any of the models can be converted to a simulation.

Many experiences of applying the above methods have been reported. The methods all seem to work, provided they are applied by experienced analysts. In each case their users are positive about success, but penetration to others has been slow.

There are difficulties at this stage due to time and effort on one hand, and expertise and judgement on the other. Model building is labour intensive and costly. It requires input from valuable people with the central insights into the design problem. The cost of modelling is one of the forces behind the perpetuation of the *fix-it-later* philosophy. Also, the judgements needed to create performance abstractions are difficult for developers to make. Some degree of abstraction is essential, but how much? This is an element of the art of performance modelling, which is not well diffused among software developers. There is no common language for model capture, which makes it difficult to communicate the concerns of the analyst, and the model itself, to others within a project.

Some users are unhappy with constraints on what can be expressed in a given form of model. The abstractions used in a form of model may make it impossible to describe some features of the execution. Designers are creative at finding new behaviour patterns with greater flexibility, but higher performance costs (examples might be inheritance, or mobile agents). Some modelling techniques do not capture these patterns easily.

Research problems in model building include a flexible expressive model for capture of behaviour, automated capture of behaviour from legacy components, and modelling within frameworks familiar to the designer, such as UML.

Step 3. Capture resource demands

Within the structure captured at the last step, the actual demand parameters for CPU time, disk operations, network services and so forth are estimated and inserted at this step. In fact the parameters are often captured at the same time as the structure, but the discussion is separated because the factors governing this step are sometimes different.

There seem to be four different sources of actual values for execution demands:

- measurements on parts of the software which are already implemented, such as basic services, existing components, a design prototype or an earlier version [15],
- compiler output from existing code,

- demand estimates (CPU, I/O, etc.) based on designer judgement and reviews,
- "budget" figures, estimated from experience and the performance requirements, may be used as demand *goals* for designers to meet (rather than as estimates for the code they will produce) (see the next section for more discussion).

The sources at the top of the list are only useful for parts of the system that are actually running, so early analysis implies at least some of the numbers will be estimated.

There has been considerable experience with estimation using expert designer judgement [11], and it works well provided it has the participation (and encouragement) of a performance expert on the panel. Few designers are comfortable in setting down inexact estimates, and on their own they have a tendency to chase precision, for example by creating prototypes and measuring them. Nonetheless the estimates are useful even if they turn out later to have large errors, since the performance prediction is usually sensitive to only a few of the values.

Cache behaviour has important effects on CPU demands, which may differ from one run to another. For embedded and hard-real-time systems, where an exact worst-case value is needed, algorithms have been developed to determine the worst-case impact of caching on execution of code.

Parameter estimation is an important area, yet it is difficult to plan research that will overcome the difficulties. Systems based on re-using existing components offer an opportunity to use demand values found in advance from performance tests of their major functions, and stored in a demand data base. Systems based on patterns offer a similar possibility, but if the pattern is re-implemented it has to be re-measured.

Step 4. Solve the model

When we consider solving the model, some advantages of one model formalism over another become apparent. Because of uncertainties in parameter values and patterns of usage it may be necessary to solve the model many times, which gives a reward to fast model solvers.

Queueing network models are relatively lightweight and give basic analytic models which solve quickly. They have been widely used ([11], [8]). However the basic forms of queueing network model are limited to systems that use one resource at a time. Extended queueing models, that describe multiple resource use, are more complex to build and take longer to solve. *Layered queueing* is a framework for extended queueing models that can be built relatively easily, and which incorporates many forms of extended queueing systems [10], [17].

Bottleneck or bounds analysis of queueing networks is even faster than a full solution and has been widely used. However, when there are many classes of users or of workloads (which is the case in most complex systems) the bounds are difficult to calculate and interpret.

The difficulty with queueing models is expressiveness; they limit the behaviour of the system to tasks providing service to requests in a queue. They cannot express the logic of intertask protocols, for instance.

Petri nets and other state-based models can capture logically convoluted protocol interactions that cannot be expressed at all in queueing models. They can in principle express any behaviour whatever. Further, some of them, such as stochastic process algebra, include mechanisms for combining subsystems together to build systems.

Their defect is that they require the storage of probabilities over a state space that may easily have millions of states, growing combinatorially with system size. They do not, and cannot scale up. However the size of system that can be solved has grown from thousands of states to millions, and interesting small designs can be analyzed this way.

Embedded and hard-real-time systems often use a completely different kind of model, a schedulability model, to verify if a schedule can be found that meets the deadlines. These are considered briefly below.

Simulation is the always-available fallback from any formulation. Some kinds of models are designed specifically for simulation, such as the performance prototype from code. Simulation models are heavyweight in the execution time needed to produce accurate results, and can give meaningless results in the hands of non-experts who do not know how to assess their precision or their accuracy. When models can be solved by both extended queueing techniques and by simulation, the simulation time may easily be three orders of magnitude greater. However simulation can always express anything that can be coded, so it never limits the modeller in that way.

The unlimited expressiveness of simulation is both a hazard and a benefit. A colleague (Bill Sanders) put it nicely, that "the comfort is in the details", and developers appreciate being able to represent detail whenever they like. However the temptation is to include excessive detail which can burden and confuse the analysis. It would be useful to have criteria to identify detail that is unnecessary.

Research into modelling technology, including efficient solution techniques, is lively and broadly based. It would be interesting to be able to combine the speed of queueing techniques with state-based techniques to take advantage of the strengths of both, for analytic estimation.

Step 5. Interpret the model, and identify improvements if if performance is unsatisfactory

If the requirements are not met, the analysis will often point to changes that make it satisfactory. These may be changes to the execution platform, to the software design, or to the requirements. To diagnose changes one must trace the causes of unsatisfactory delays and throughputs, back to delays and bottlenecks inside the system. Sometimes the causes are obvious, for example a single bottleneck which delays all responses and throttles the throughput. Sometimes they are more subtle, such as a network latency that blocks a process, making it busy for long unproductive periods and creating a bottleneck there. Examples are given in [11] and [3] . Woodside describes the process of tracing causality in [16].

Changes to the software design may be to reduce the cost of individual operations, or to reduce the number of repetitions of an operation, as described in [11]. Larger scale changes may be to change the process architecture or the object architecture, to reduce overhead costs or to simplify the control path. If a single response class, distinct from other processing, is unsatisfactory, then it may be possible to speed it up by using priorities provided it is executed by a separate process.

If the bottleneck is a processing device (processor, network card, I/O channel, disk, or attached device of some kind) then the analysis can be modified to consider more powerful devices. If the cost of adapting the software or the environment is too high, one should finally consider the possibility that the requirements are unrealistic and should be relaxed.

2.2 Summary

Practical methods are known to handle all these steps, however there are difficulties which inhibit their use in many development projects. Two major, broad problems are:

- the high skill level needed to apply them. (Thus, simpler methods or easy-to-use tools should be a goal of research.)
- the separation of performance engineering from development methods. (Thus, ways to integrate the two should be sought.)

Underdeveloped technical areas related to models include performance requirements analysis, resource demand capture for components, and tracing causality of performance shortfalls in complex systems.

3 Validation of Resource Budgets

Before a new product is designed there are no figures for resource demands (that is, for the CPU execution demand and the demands for other operations using disks, networks, etc.). Rather than considering the prediction of performance based on demand estimates, it may be better to plan budgets for resource demands for all the operations, and use the validation to check that they will give the required performance. The budgets then become targets to be met by the designers of the subsystems. In this approach the performance requirements are decomposed into budgets for subsystems, and the validation step validates the decomposition.

The methods just described can also be used, without change, to validate budgets. The budgeted resource demands are inserted into the model at Step 3, and the evaluation is carried out. If the validation does not pass, then the adjustments to be considered at Step 5 also include changes to the budgets for some operations.

4 Other Aspects of Early Software Validation

Other aspects of software should also be validated at an early stage, however limitations of space and experience preclude a full discussion here. Examples are:

- Reliability, availability, performability, are approached by techniques very similar in principle to those for responsiveness, using models which include failure and repair mechanisms, and their parameters such as mean-time-to-failure and mean-time-to-repair. The same five steps are applied, but using these different models. An example is given in [4], and other examples may be found in the proceedings of the conference [6]. Many of the same difficulties occur in this area, as in performance validation, and successes are limited to systems of modest size.

- Schedulability is similar to responsiveness, in that it deals with completing responses in time, but it uses different techniques and is concerned with systems that fail if a single response is late. These are sometimes called hard real-time systems. Examples of techniques may be found in the proceedings of the conference [9] or the journal [7]. Excellent techniques exist for single processors, and strong heuristics for more complex systems. However many aspects of modern computing technology (caching, networks) conspire against deterministic behaviour.

5 Conclusions

An ideal performance validation methodology would:

- be capable of dealing with multiple alternative deployments of the same software
- be selectable, so one can apply low-cost, less powerful techniques when performance is not critical, and then step up to more powerful analysis if needed later,
- allow the assessment effort to be incremental, so that additional effort gives additional insight,

- be adaptable so as the design evolves the original models are extended and the predictions updated,
- be capable of tracking the design as it develops,
- incorporate analysis techniques and process steps to address other aspects of performance: reliability, safety, etc.

Practitioners adapt SPE to address many of these issues in the course of their studies. Unfortunately, the state of the practice is still to apply ad-hoc techniques for each type of system. The SPE process has not yet evolved to prescribe an exhaustive set of procedures based on the type of system. The process is largely practiced by performance specialists. It needs to be better integrated with the software development process with specific SPE deliverable at each stage in the life cycle.

The model-based approach has been successfully applied to validate performance in a variety of systems. The two major difficulties which inhibit its use in many development projects are:
- The high skill level needed to apply them,
- the separation of performance engineering from development methods.

Underdeveloped technical areas related to models include performance requirements analysis, resource demand capture for components, and tracing causality of performance shortfalls in complex systems.

Is it possible to validate performance in early development stages with a model-based approach? Models can identify performance problems but they cannot prove absolutely the absence of problems. It is not possible to completely validate the model until the final system can be measured. Perhaps *verification* is a better concept to apply to performance:

Verification: The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase (IEEE)

If quantitative performance objectives can be set for each stage, it should be possible to measure the product for that stage and determine whether the objective has been met.

Most SPE studies adjust the performance models to match the software as it evolves and new decisions are made. Perhaps a better approach is to "build to the model," that is, if the model predicts that if the system is constructed in this way, with these resource usage requirements, etc., then it will work – then the SPE process should verify that the evolving system matches the specifications of the early models. This strategy is particularly important for safety-critical systems.

References

[1] R. L. Bagrodia, C.-C. Shen, "MIDAS: Integrated Design and Simulation of Distributed Systems", IEEE Trans. on Software Engineering, v 17, n 10, pp 1042--1058, Oct. 1991.

[2] H. Beilner, J. Mater, N. Weissenberg, "Towards a Performance Modelling Environment: News on HIT", *Proc. Int. Conf on Modeling Techniques and Tools for Computer Performance Evaluation*, 1989, pp 57--75.

[3] J. Dilley, R. Friedrich, T. Jin and J.A. Rolia, "Measurement Tools and Modeling Techniques for Evaluating Web Server Performance", *Proc. 9th Int. Conf. on Modelling Techniques and Tools*, St. Malo, France, June, 1997

[4] H. Duggal, M. Cukier, W. Sanders, “Probabilistic Verification of a Synchronous Round-based Consensus Protocol”, Proc. 16th Symp. on Reliable Distributed Systems, Durham, NC, October 1997.

[5] H. El-Sayed, D. Cameron, M. Woodside, "Automated Performance Modeling from Scenarios and SDL Designs of Telecom Systems", *Proc. of Int. Symposium on Software Engineering for Parallel and Distributed Systems (PDSE98),* Kyoto, April 1998.

[6] IPDS, Proc. IEEE Int. Computer Performance and Dependability Symposium

[7] Journal of Real-Time Systems, Kluwer, New York.

[8] D.A. Menasce and H. Gomaa, "On a Language Based Method for Software Performance Engineering of Client/Server Systems", *Proc. of First International Workshop on Software and Performance (WOSP98)*, pp 63-69, Oct. 1998.

[9] Proc. of the Real-Time Systems Symposium, IEEE Computer Society, held annually since 1979.

[10] J.A. Rolia and K.C. Sevcik, "The Method of Layers", *IEEE Trans. on Software Engineering*, v 21 n 8 pp 689-700, Aug. 1995.

[11] C.U. Smith, *Performance Engineering of Software Systems*, Addison-Wesley, 1990

[12] C.U. Smith and L.G. Williams, “Performance Engineering Evaluation of CORBA-based Distributed Systems with *SPE•ED*”, *Proc. 10th Int. Conf. On Modelling Tools and Techniques*, Palma de Mallorca, 1998..

[13] M. Steppler, "Performance Analysis of Communication Systems Formally Specified in SDL", *Proc. of First International Workshop on Software and Performance (WOSP98*), pp 49-62, Oct. 1998.

[14] V. Vetland, P. Hughes, A. Solvberg, "A Composite Modelling Approach to Software Performance Measurement*", Proc. ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, Santa Clara, CA, pp 275-276, May 1993.

[15] V. Vetland, *Measurement-Based Composite Computational Work Modelling of Software*, PhD. thesis, University of Trondheim, Aug 1993.

[16] C.M. Woodside, "A Three-View Model for Performance Engineering of Concurrent Software", *IEEE Transactions on Software Engineering*, v 21, n 9, pp 754-767, Sept. 1995.

[17] C.M. Woodside, J.E. Neilson, D.C. Petriu, S. Majumdar, "The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-like Distributed Software", *IEEE Transactions on Computers*, v 44, n 1, pp 20 - 34, Jan 1995

[18] C.M. Woodside, C. Hrischuk, B. Selic and S. Bayarov, "A Wideband Approach to Integrating Performance Prediction into a Software Design Environment*", Proc. of First International Workshop on Software and Performance (WOSP98)*, pp 31 - 41, Oct. 1998.