Designing High-Performance Distributed Applications Using Software Performance Engineering: A Tutorial

Connie U. Smith, Ph.D.

Performance Engineering Services,
PO Box 2640, Santa Fe, New Mexico, 87504-2640 USA
Telephone (505) 988-3811

December 1996

Appears in *Proceedings Computer Measurement Group*, San Diego, 1996.

Designing High-Performance Distributed Applications Using Software Performance Engineering: A Tutorial

Connie U. Smith, Ph.D.
Performance Engineering Services
PO Box 2640
Santa Fe, NM 87504
www.perfeng.com/~cusmith
(505) 988-3811

This paper reviews the purpose of Software Performance Engineering (SPE) and the steps for applying the SPE methods throughout the life cycle of new systems. It describes the data required for SPE studies and illustrates a performance walkthrough for gathering the data. It reviews the software and system execution models for assessing the performance of alternatives and for capacity planning to support the new applications.

SPE Definition

Software performance engineering (SPE) is a method for constructing software systems to meet performance objectives. The process begins early in the software life cycle and uses quantitative methods to identify satisfactory designs, and to eliminate those that are likely to have unacceptable performance, before developers invest significant time in implementation. SPE continues through the detailed design, coding, and testing stages to predict and manage the performance of the evolving software, and to monitor and report actual performance against specifications and predictions. SPE methods cover performance data collection, quantitative analysis techniques, prediction strategies, management of uncertainties, data presentation and tracking, model verification and validation, critical success factors, and performance design principles.

Performance refers to the response time or throughput as seen by the users. Responsiveness limits the amount of work processed, so it determines a system's effectiveness and the productivity of its users. Many users subconsciously base their perception of computer service more on system responsiveness than on functionality. Negative perceptions, based on poor responsiveness of new systems, seldom change after correction of performance problems.

The performance balance in Figure 1 depicts a system that fails to meet performance objectives because resource requirements exceed computer capacity. With SPE, analysts detect these problems early in development and use quantitative methods to support cost-benefit analysis of hardware solutions versus software requirements or design solutions, versus a combination of the two. Developers implement software solutions before problems are manifested in code; organizations implement hardware solutions before testing begins.

Is SPE necessary? Isn't hardware fast enough and cheap enough to resolve performance problems? Surprisingly, the use of state-of-the-art hardware and software technology dramatically *increases* the risk of performance failures. This seems counter-intuitive — one would expect increased performance — but the newness of the products combined with the developers inexperience with the new environment leads to problems. This is particularly true for distributed systems, such as client/server, because the myriad of

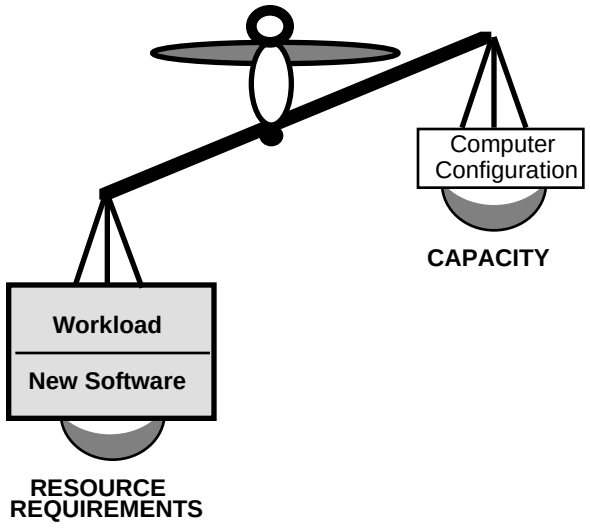


Figure 1. Performance "balance"

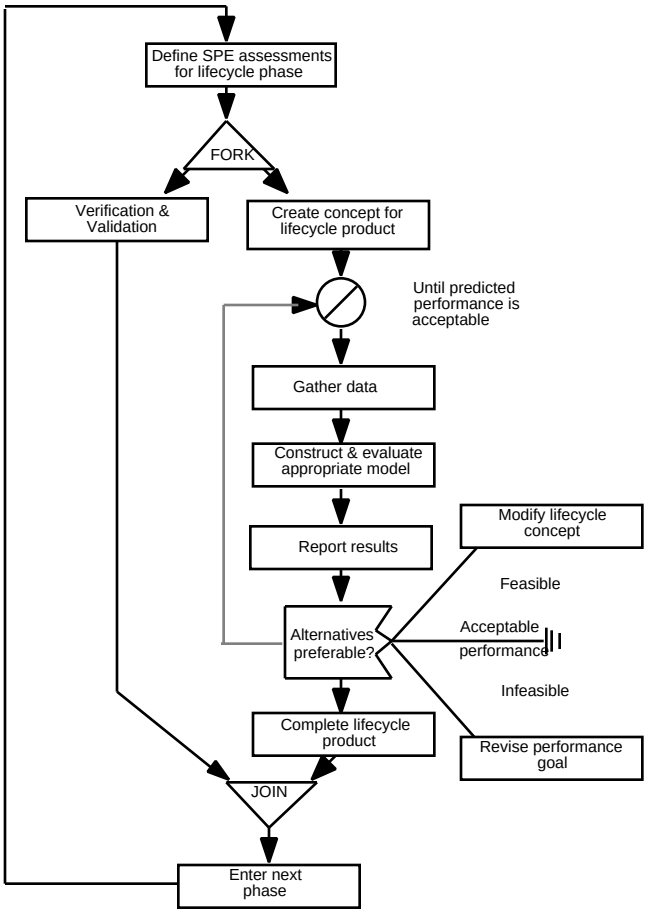


Figure 2. SPE Process

inter-related performance factors and the complexity of the design choices make intuitive performance design decisions difficult if not impossible.

SPE techniques have been used successfully for more than 15 years. This paper summarizes these techniques for readers who would like to learn to apply them. It gives an overview of SPE, reviews the steps in the SPE process, defines the data needed to assess performance in early life cycle stages, and describes the models to predict performance. These SPE techniques can be used with all types of systems; however this paper specifically addresses distributed systems because SPE techniques are particularly beneficial for them. The paper does not provide comprehensive coverage of the SPE related work. For more information about related work see [SMIT94a]. For other papers that provide examples of SPE for distributed systems see [HEID93; SMI94d; SWIN92].

SPE Process

SPE augments other software engineering methodologies, it does not replace them. With SPE, analysts apply quantitative methods throughout development to confirm that performance is satisfactory. SPE prescribes principles for creating responsive software, specifications necessary for evaluation, procedures for obtaining performance specifications, and guidelines for the types of evaluations to be conducted at each developmental stage. It incorporates models for representing and predicting software system performance, and a set of analysis methods. The SPE process is depicted in Figure 2; the steps are described in the following paragraphs.

The first step defines the SPE assessments for the current life cycle phase. Assessments determine whether planned software meets its performance objectives, such as acceptable response times, throughput thresholds, or constraints on resource requirements. A specific, quantitative objective is vital for analysts to determine concretely whether or not that objective can be met. It is essential to define the performance objectives and the expected usage patterns in order to determine the most appropriate means of achieving them, and avoid devoting time to unintentionally overachieving them.

We specify the performance objectives in terms of *responsiveness* as seen by the system users. Both the response time for an interactive task and the number of work units processed in a time interval (throughput) are measures of responsiveness. Responsiveness does not necessarily imply efficient computer resource usage. Efficiency is addressed only if critical computer resource constraints must be satisfied.

After defining the goals, designers create the concept for the life cycle product. For early phases the concept is the functional architecture -- the software requirements and the high-level plans for satisfying them. In subsequent phases the concept is a more detailed design, the algorithms and data structures, the code, etc. Principles for creating responsive software are covered in [SMIT88a; SMIT90a]

Once the life cycle concept is formulated, we gather data sufficient for estimating its performance. First we need the projected system workload: how it will *typically* be used. Then we need an explanation of the current design concept. Early in development, we use the general system architecture; later we add the proposed decomposition into object classes or modules; still later, we incorporate the proposed algorithms and data structures. We also need estimates of the resource usage of the design entities. The *SPE Data* section provides an overview of data requirements and techniques for gathering specifications.

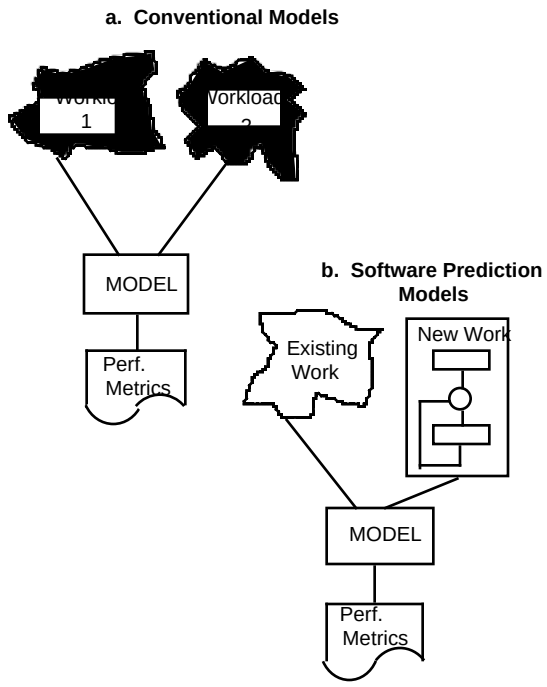


Figure 3. SPE Model Overview

Because the precision of the model results depends on the precision of the resource estimates, and because these are difficult to estimate early in software development, a best and worst-case analysis strategy is integral to SPE. We use estimates of the lower and upper bound when there is high uncertainty about resource requirements. Using them, the analysis produces an estimate of both the best and worst-case performance. If the best-case performance is unsatisfactory, we seek feasible alternatives. If the worst-case performance is satisfactory, we proceed with development. If the results are somewhere in between, we identify critical components whose resource estimates have the greatest effect, and focus attention on obtaining more precise data for them. A variety of techniques provide more precision, such as further refining the design concept and constructing more detailed models, or constructing performance benchmarks and measuring resource requirements for key elements.

To predict software performance, analysts construct a model of the software execution that can be solved for the indicative performance metrics. The models are similar to those used for conventional performance evaluation studies. In conventional studies of existing systems, analysts model systems to predict the effect of workload or configuration changes. The conventional modeling procedure depicted in Figure 3a is as follows: study the computer system; construct a model (either a queueing network or a simulation model); measure current execution patterns; characterize workloads; develop model input parameters; validate the model by solving it and comparing the model results to observed and measured data for the computer system; and calibrate the model until its results match the measurement data. Then we use the conventional model to evaluate changes to the computer system by modifying the corresponding workload parameters, the computer system resource parameters, or both. After each change, we compare the model results to the performance goal. We repeat the change-evaluate process until we identify the desired performance solution. Practitioners rely on these models for computer capacity planning. The model precision is sufficient to predict future configuration requirements. They are widely used, and they work; so we use them as the basis for SPE.

The SPE performance models in Figure 3b are similar; however, because the software does not yet exist, it is not possible to measure the workload parameters. We first model the workload explicitly.

Two models satisfy the modeling requirements: the *software execution model* and the *system execution model*. The software execution model represents key facets of software execution behavior; its solution yields workload parameters for the system execution model, which closely resembles the conventional models. An overview of the construction and evaluation of the performance models follows in the *Model Overview* section.

If the model results indicate that the performance is likely to be satisfactory, developers proceed. If not, analysts report quantitative results on the predicted performance of the original design concept. If alternative strategies would improve performance, they report the alternatives and their expected quantitative improvements. Developers review the findings to determine the cost-effectiveness of the alternatives. If a feasible and cost effective alternative exists, developers modify the concept (architecture design, or implementation decision) before the life cycle product is complete. If none is feasible as, for example, when the modifications would cause an unacceptable delivery delay, we explicitly revise the performance goal to reflect the expected degraded performance.

A vital and on-going activity of the SPE process is to verify that the models represent the software execution behavior, and validate model predictions against performance measurements. Compare the model specifications for the workload, software structure, execution structure, and resource requirements to

discrepancies to update the performance predictions, and to identify the reasons for differences -- to prevent similar problems occurring in the future. Similarly, compare system execution model results (response times, throughput, device utilization, etc.) to measurements. Study discrepancies, identify error sources, and calibrate the model as necessary. Begin the model verification and validation early and continue throughout the life cycle. In early stages, focus on key performance factors; use prototypes or benchmarks to obtain more precise specifications and measurements as needed. The evolving software becomes the source of the model verification and validation (V&V) data.

This discussion outlined the steps for one design-evaluation pass. We repeat the steps throughout the life cycle. For each pass the goals and the evaluation of the objectives change somewhat. The *Life Cycle SPE* section discusses the life cycle stages and the questions to be considered.

SPE Data

ln order to create a software execution model you need five types of data: performance goals, workload definitions, software execution characteristics, execution environment descriptions, and resource usage estimates. An overview of each follows.

Precise quantitative *performance goals* are vital in order to concretely determine whether or not performance objectives can be met. For database applications, both on-line performance goals and batch window objectives must be met. For on-line transactions, the goals specify the response time or throughput required, and define the external factors that impact goal attainment, such as the time of day, the number of concurrent users, whether the goal is an absolute maximum, a 90^{th} percentile, etc. For distributed applications, goals may specify the total time to complete a user task including the time that may be required to access a remote system.

Workload definitions specify the key scenarios of the new software. For on-line transactions they initially specify the transactions expected to be the most frequently used. Later they also include resource intensive transactions. Of all functions provided by new systems, more than 80% of the users' interactions typically invoke less than 20% of the functions. Consider your use of Automated Teller Machines (ATM's). Of all possible transactions available to you, more than 80% of your requests likely use only one or two of them. On-line workload definitions identify the key scenarios and specify their workload intensity: the request arrival rates, or the number of concurrent users and the time between their requests (think time). Batch workload definitions identify the programs on the critical path, their dependencies, and the data volume to be processed.

Software execution characteristics identify components of the software system to be executed for each workload scenario. Of all possible software components, less than 20% are likely executed more than 80% of the time. The software execution scenario identifies:

- components most likely to be invoked when users request the corresponding workload scenario;
- the number of times they are executed or their probability of execution;
- and their execution characteristics, such as the database tables used, and screens read or written.

In distributed systems the workload scenarios may require software execution on multiple platforms as well as network transmission. The software execution characteristics identify all such processing requirements; then SPE analysis strategies focus on the key software processing requirements for the initial evaluations.

The *execution environment descriptions* define the computer system configuration, such as the CPU, the operating system, and the I/O subsystem characteristics. It provides the underlying queueing network model topology and defines resource requirements for frequently used service routines. This is usually the easiest information to obtain. Performance measurement tools provide most of it, and most capacity and performance analysts are familiar with the requirements.

Resource usage estimates determine the amount of service required of key devices in the computer system configuration. For each software component executed in the workload scenario you need:

- the approximate number of instructions executed (or CPU time required),
- the number of physical I/Os,
- use of other key devices such as communication lines (number of messages and amount of data), memory (temporary storage, map and program size), etc.

For database applications, the database management system (DBMS) accounts for most of the resource usage, so you need the number of database calls and their characteristics. For distributed systems, you need resource requirements on each platform and the demand for communication services. Early life cycle requirements are tentative, difficult to specify, and likely to change, so you need upper and lower bound estimates to identify problem areas or software components that warrant further study to obtain more precise specifications.

You may find that the number of times a component executes or its resource usage varies significantly. To represent the variability: identify the factor(s) which cause the variability, use a data dependent variable to represent the key factors, and specify the execution frequency and resource usage in terms of the data dependent variables. For example, in a database environment the number of times some transaction components execute may depend on the number of rows qualified in a SELECT. Use a parameter, *N*, to represent the number qualified. Then possible execution frequencies may be *N* or *2N* or *.3N*, etc. Similarly, the number of instructions executed within a component could be *500N*. Later, study the performance sensitivity to various parameter values.

It is seldom possible to get precise data for all these specifications early in the software life cycle. Do not wait to model the system until it is available (i.e., in detailed design or later). Gather guesses, approximations, and bounded estimates to begin then augment the models as data becomes available. For example, during the requirements analysis phase you can identify some key database tables but may not know their contents. You can get a very general description of high use screens, but may not know all the fields they will ultimately contain. Start with these approximations – if performance is a problem at this stage it must be corrected early, before more tables and other processing is added. This approach has the added advantage of focusing attention on key workload elements to minimize their processing (as prescribed by the centering principle in [SMIT88a; SMIT90a]). Otherwise, designers tend to postpone these important performance drivers in favor of designing more complex but less frequent parts of the software.

Data Gathering Example

SPE suggests techniques for gathering SPE data in a performance walkthrough. This section illustrates the process with a simple example. To illustrate what transpires in a walkthrough, consider a database example. The users of the system are represented by a manager of customer service representatives (CSR manager) who is knowledgeable of the future transactions and the characteristics of the customer inquiries handled by the service representatives. The designers and implementors are the software specialists who will develop the application programs. They first discuss the SPE purpose: to determine whether the transaction response time will enable the customer service representatives to handle incoming calls. Some discussion follows to quantify the current and future response times, the peak hours, number of customer service representatives, rate and nature of incoming calls, etc.

The CSR manager then describes the system requirements, questions typically posed by callers, and specific transaction usage scenarios of the customer service representatives. The performance engineer asks questions such as how many customers are in the database, which screens or transactions are typically used for each, etc. Together they identify 4 high use transactions and an initial response time goal of under 1 second. It is refined to specify the characteristics of the peak hours, rate of calls, number of on-line terminals, etc. when the one-second goal must be met.

Next, the designers and implementors describe the database characteristics, and the software system.

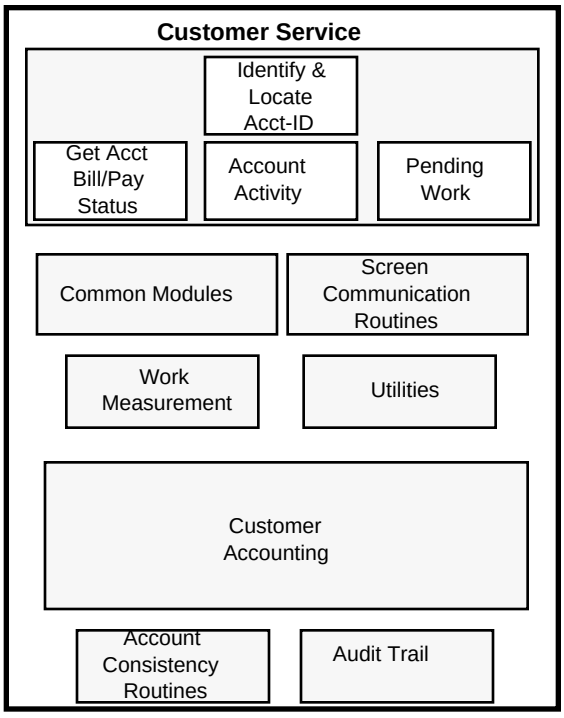


Figure 4. Major Subsystems

Table 1. Primary Tables for DB Example

Table Name	No. of Rows	Bytes per Row	Description
Acct-ID	1.5 Million	100	Cross reference account to unique identifier
Debit	36 Million	40	All items billed for last 6 months
Credit	18 Million	40	All payments during last 6 months
Collection History	150,000	60	Collection notices generated during last 6 months
Activity	200,000	25	Transaction usage information

Figure 4 gives an overview: Some of the primary tables are in Table 1. The major subsystems and support routines are in Figure 4. The "customer service" box shows the 4 high use transactions which comprise the initial workload scenarios. Next, the designers identify the components to be executed (the execution structure) for the specified scenarios. The performance analyst interprets the information presented and responds by describing a software execution model of the derived execution structure.

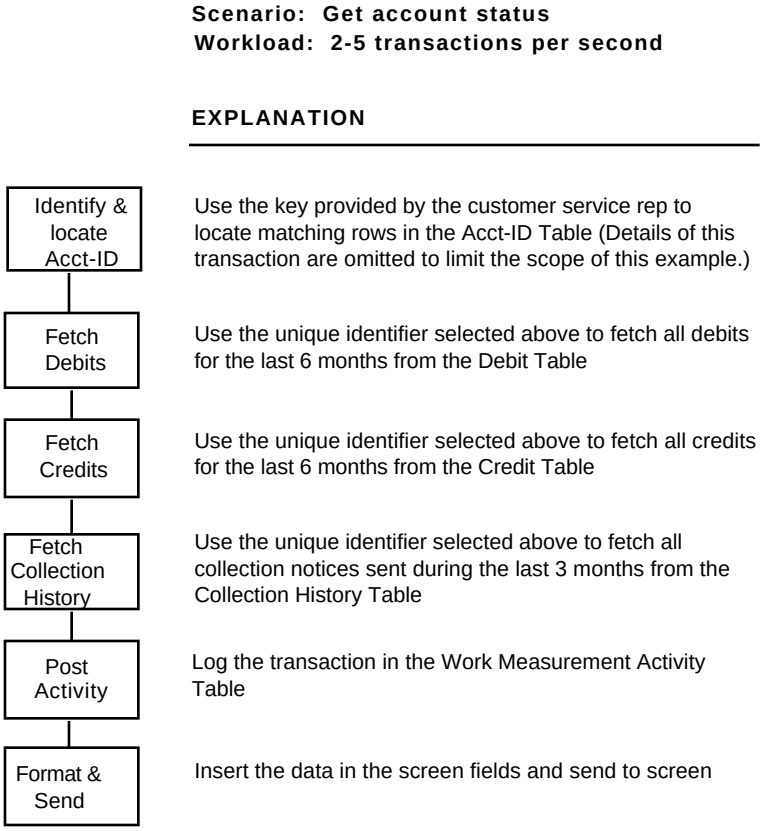


Figure 5. Software Execution Structure

Figure 5 illustrates the processing steps in a user scenario. This example shows the processing that executes on a single system. In a distributed system, you will have a similar scenario for the key processing steps on each processor.

The execution environment is already known although a system specialist may be consulted to resolve configuration questions that arise during the walkthrough. Therefore, after the execution structures are derived, the walkthrough participants estimate the resource usage specifications, taking into consideration the CSR manager's knowledge about the workload, the software specialist's knowledge of the processing, and the performance analyst's knowledge of the environment. Figure 6 associates resource estimates with each software component. CPU estimates came from discussing the processing required and relating it to existing transactions for which measurement data is available. Lower bound I/O estimates assume the top two index levels are already in memory, and assume a clustering index which places all needed data records on one page. Upper bound estimates assume only the root index node is in memory, and all data rows are on separate pages.

Experienced performance engineers may suspect performance problems, but they are not obvious to other participants, so we defer discussing alternatives. We repeat the process to develop scenarios for the other high use transactions. The meeting concludes with a summary of the findings, and schedules the results presentation and next walkthrough.

Model Overview

We use graphs to represent the workload scenarios. Nodes represent functional components of the software, arcs represent control flow. The graphs are hierarchical with the lowest level containing complete information on estimated resource requirements. Figure 6 is a simple software execution model. See [SMIT90a] for a definition of software execution models, how to represent processing steps with nodes and how to evaluate models.

First, we use software execution models for a static analysis to derive the mean, best- and worst-case

Scenario: Get account status
Workload: 2 - 5 transactions per second

	CPU (sec.)	DB Action &Table	DBI/Os	File I/Os
Identify & locate Acct-ID	.05 - .3	SEL Acct-ID	2-24	3
Fetch Debits	.05 - .1	SEL Debit	3-15	2
Fetch Credits	.05 - .1	SEL Credit	3-9	0
Fetch Collection History	.05 - .1	SEL Coll-Hist	2-3	0
Post Activity	.01 - .05	ADD Activity	0-1	0
Format & Send	.001 - .01		0	3

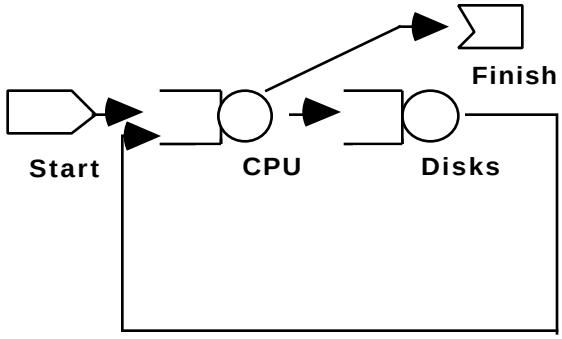
Figure 6. Scenario Resource Requirements

response times. The static analysis makes the optimistic assumption that there are no other jobs on the host configuration competing for resources. We compute the resource requirements for the lowest hierarchical level of detail, then use the result for the corresponding node at the next higher level of the graph. We repeat the computation until, ultimately, we have the totals for the entire graph. The specific algorithms are in [SMIT90a]. For the simple scenario in Figure 6, the totals are: CPU = .211-.660, DB I/Os = 10-52, File I/Os = 8. If an I/O takes an average of 25 ms, the best case elapsed time for the scenario is:

.211 + (10 x .025) + (8x.025) = .661 sec.

Next, we use the resource requirements to specify parameters for the *system execution model* to solve for the following additional information:

- More precise metrics that account for resource contention
- Sensitivity of performance metrics to variations in workload composition
- Effect of new software on service level objectives of other systems
- Identification of bottleneck resources
- Comparative data on performance improvement options to the workload demands, software changes, hardware upgrades, and various combinations of each



BEST CASE

	Load 2 Utilization	Load 5 Utilizatio
Disks	45%	100%
CPU	42%	100%
Response time (sec)	1.2	*****

Figure 7. System Execution Model

To construct and evaluate the system execution model, we first represent the key computer resources with a network of queues. The queueing network model may include queues to represent resources on multiple processors and queues to represent communication lines. Then we use environment specifications to specify device parameters (such as CPU size and processing speed). We derive the workload parameters and service requests for new software from the resource requirements computed from the software execution models. Then we solve the model and check for reasonable results. We examine the model results. If the results show that the system fails to meet performance objectives, we identify bottleneck devices and correlate system execution model results with software components. We identify and evaluate alternatives to the software plans or the computer configuration, and evaluate the alternatives by making appropriate changes to the software or system model and repeat the analysis steps.

The simple system execution model for the database example is in Figure 7. This model represents the software execution on a single platform in the environment. It would be appropriate to model calls to a stored procedure on a server. Subsequent models add queues for additional platforms and key communication devices. Subsequent models also add workloads for existing work on this configuration – model parameters are derived from measurements as described in the *SPE Process* section. These results show that the system meets the performance goal for the lower bound on throughput, but fails for the upper bound. Hardware solutions would use more disk devices or faster devices. Software solutions would revise the scenario to execute fewer database calls. We quantify the alternatives by changing parameters in the software execution models, the system execution model, or both, and solving the models. After selecting the appropriate alternative developers proceed to the next life cycle phase.

Life Cycle SPE

The previous sections outline the SPE steps:

- define objectives,
- apply principles to formulate performance-oriented concepts,
- gather data,

- model and evaluate,
- measure to verify model fidelity, validate model predictions, and confirm that software meets performance objectives.

The steps are repeated throughout the life cycle. The goals and the evaluation of the objectives change somewhat for each pass.

Table 2 contains a synopsis of the SPE considerations in each life cycle stage. The first evaluation occurs during the requirements definition and the initial formulation of the software design; it assesses the feasibility and desirability of the functional architecture to detect infeasible plans. The requirements may be prescribed before development begins and many developers perceive them to be non-negotiable. It is nevertheless prudent to verify that the requirements can be met with reasonable cost and performance.

The next evaluation determines the computer configuration required to support the new product, that is, the power of the supporting hardware and operating system software. These are not independent issues: the design depends upon the requirements, and the configuration will vary with the design. Therefore, analysts evaluate several combinations of requirements, designs, and configurations to determine the best combination.

Table 2. Synopsis Of The Performance Engineering Considerations

Life Cycle Stage	Performance Considerations
Requirements Analysis Functional Architecture	What are typical uses? How will the software be used? What are the performance goals for these scenarios? Can the requirements be achieved with acceptable performance? Approximately how much computer power is required to support it?
Preliminary Design	Does the expected performance of this design meet specifications? Is the proposed configuration adequate? Excessive?
Detailed Design	Have changes occurred that affect earlier predictions? What is a more realistic estimate of the projected performance?
Implementation and Integration testing	How does the performance of the implementation alternatives compare? Have any unforeseen problems arisen? What are the resource requirements of the critical components? Are the performance requirements met?
Maintenance & Operation	What is the effect of the proposed modifications? What are the long-range configuration requirements to support future use?

After establishing a feasible set of requirements, the functional architecture of the software, and the supporting configuration requirements, the focus of SPE changes. Analysts identify components that are critical with respect to meeting performance goals, and implement them first. This identification of critical components maximizes the impact of the performance efforts, and yields early measurement data of actual resource requirements to produce more precise predictions.

In the middle stages of the life cycle, SPE incorporates additional design details as they evolve, and design changes as they occur. Continued analysis helps to detect problems as soon as possible. At this stage, SPE studies provide data for developers to select appropriate data structures, algorithms, and system decomposition strategies. Models incorporate operating system overhead for memory management, data management, resource management and communication. As implementation proceeds, analysts add implementation details, refine the resource requirements estimates, and use more detailed models to produce more precise predictions.
During the maintenance stage, SPE evaluates both major and minor revisions which correct defects or add functions. Minor changes use mid-life cycle techniques to incorporate changes into the models and assess their performance impact. Major revisions use early life cycle techniques to assess the feasibility and desirability of the revised requirements and functional architecture, and follow the major-revision project with standard SPE steps. Maintenance evaluations require much less effort than original studies when the SPE models are current and complete and can be used as the basis for the analysis.

System performance engineering techniques evaluate the overall end-to-end performance to ensure that performance objectives will be met when all subsystems are combined. Systems with complex combinations of software, networks, and hardware have many potential pitfalls in addition to application software choices.

Summary

This paper reviewed the purpose of SPE, the steps for applying the SPE methods throughout the life cycle of new systems, the data requirements and illustrated a performance walkthrough approach for gathering the data. The software and system execution models for assessing the performance of alternatives were described and their use throughout the life cycle was reviewed.

SPE is becoming increasingly popular for large, strategic systems to systematically manage the performance. The models are relatively easy to construct and evaluate with the performance modeling tools available today. SPE's current problems are not the technical problems of constructing and evaluating the performance models. The chief barriers are making developers and managers aware of SPE's potential, and persuading them to incorporate the methods into their development life cycle.

References

[HEID93] Ruth V. Heidel and George W. Dodson, (Editor), *Special Issue on Client/Server Computing*, CMG Transactions, no. 82, 1993.

[SMIT88a] Connie U. Smith, "Applying Synthesis Principles to Create Responsive Software Systems," *IEEE Transactions on Software Engineering*, vol. 14, no. 10, 1988, pp. 1394-1408.

[SMIT90a] Connie U. Smith, *Performance Engineering of Software Systems*, Reading, MA, Addison-Wesley, 1990.

[SMIT94a] Connie U. Smith, "Performance Engineering," in *The Encyclopedia of Software Engineering*, John Wiley and Sons, 1994.

[SMI94d] Connie U. Smith and Bernie Wong, "SPE Evaluation of a Client/Server Application," *Proc. Computer Measurement Group*, Orlando, FL, 1994.

[SWIN92] Carol Swink, "SPE in a Client/Server Environment: A Case Study," *Proceedings Computer Measurement Group Conference*, Reno, 1992, pp. 271-284.