Software Performance Engineering

Connie U. Smith, Ph.D.

Performance Engineering Services
PO Box 2640
Santa Fe, NM 87504

Abstract. Software performance engineering (SPE) is a method for constructing software systems to meet performance objectives. It uses quantitative analysis techniques to predict and evaluate performance implications of design and implementation decisions. This tutorial introduces SPE then covers the evolution of SPE. It reviews the SPE process and the methods used in the software lifecycle. It presents general principles for performance oriented design, then it introduces the quantitative techniques for predicting and analyzing performance. The conclusion reviews the status and future of SPE.

1 Introduction

Software performance engineering (SPE) is a method for constructing software systems to meet performance objectives. The process begins early in the software lifecycle and uses quantitative methods to identify satisfactory combinations of requirements and designs, and to eliminate those that are likely to have unacceptable performance, before developers begin 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.

The "performance balance" in Figure 1 depicts a balanced system: resource requirements match computer capacity, and software meets performance objectives. With SPE, analysts assess the "balance" *early in development.* If demand exceeds capacity, quantitative methods support cost-benefit analysis of hardware solutions versus software requirements or design solutions, versus a combination of the two. Developers select software or hardware solutions to performance problems before proceeding to the detailed design and implementation stages.

In this tutorial, *performance* refers to the response time or throughput as seen by the users. For real-time, or *reactive* systems, it is the time required to respond to events. Reactive systems must meet performance objectives to be correct. Other software has less stringent requirements, but responsiveness limits the amount of work processed, so it determines a system's effectiveness and the productivity of its users.

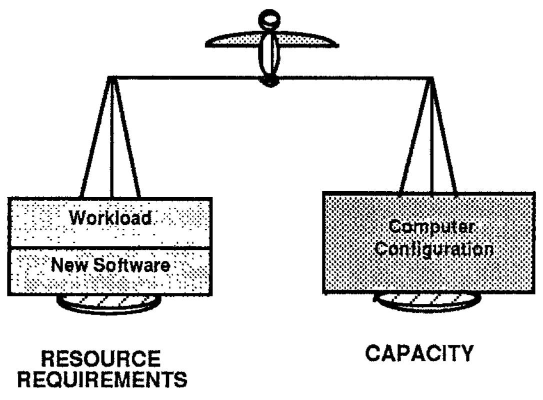

Fig. 1. Performance Balance

SPE is a *software-oriented* approach — it focuses on architecture, design, and implementation choices for managing performance. Other approaches have been proposed. For example, *system-oriented* approaches that focus on scheduling [SHA90; XU87], resource allocation, operating system executives [STAN91], total system approaches [KOPE85; KOPE91; LEVI90; POSP92; SHIN92] and so on are viable, supplemental methods for managing performance of real-time systems, but are outside the scope of the SPE approach defined here. Similarly, techniques that focus on timing requirements using temporal logic or fault-tree analysis [GABR90; JAFF91; JAHA87] are not within this SPE definition.

This tutorial first covers the evolution of SPE then it gives an overview of the SPE process and the SPE methods. It presents the general principles for performance-oriented design, then it introduces the quantitative techniques for predicting and analyzing performance. Finally, the conclusion reviews the status and future of SPE.

2 The Evolution of SPE

Performance was a principal concern in the early years of computing. Knuth's early work focused on efficient data structures, algorithms, sorting and searching [KNUT68; KNUT73]. The space and time required by programs had to be carefully managed to fit them on small machines. The hardware grew but, rather than eliminating performance problems, it made larger, more complex software feasible and programs grew into systems of programs. Software systems with strict performance requirements, such as flight control systems and other embedded systems used detailed simulation models to assess performance. Consequently creating and solving them was time-consuming, and updating the models to reflect the current state of evolving software systems was problematic. Thus, the labor-intensive modeling and assessment were cost-effective only for systems with strict performance requirements.

Authors proposed performance-oriented development approaches [BELL77; GRAH73; RIDD78; SHOL75] but most developers of non-reactive systems adopted the "fix-it-later" approach. It advocated concentrating on software correctness, deferring performance considerations to the integration testing phase and (if performance problems were detected then) procuring additional hardware or "tuning" the software to correct them. Fix-it-later was acceptable in the 1970s, but in the 1980s the demand for computer resources increased dramatically. System complexity increased while the proportional number of developers with performance expertise decreased. This, combined with a directive to ignore performance, made the resulting performance disasters a self-fulfilling prophecy. Many of the disasters could not be corrected with hardware – platforms with the required power did not exist. Neither could they be corrected with tuning – corrections required major design changes, and thus reimplementation. Meanwhile, technical advances led to the SPE alternative.

SPE uses *models* to predict performance, *tools* to formulate and solve models, and *methods* to prescibe how and when to conduct performance studies. The SPE techniques developed in the 1980s focused on models that could be solved with analytic techniques because the tools and the speed of the processors made analytic techniques more desirable than simulation techniques for early lifecycle design tradeoff studies. Consequently, the following sections on the SPE evolution focuses on these analytic techniques, tools, and methods. Later, section 2.3 presents recent developments that make other models, tools, and adaptations of the methods viable.

2.1 Modeling Foundations

In 1971, Buzen proposed modeling systems with queuing network models and published efficient algorithms for solving some important models [BUZE71]. In 1975, Baskett, et. al., defined a class of models that have efficient analytic solutions [BASK75]. The models are an abstraction of the computer systems they model, so they are easier to create than general purpose simulation models. Because they are solved analytically, they can be used interactively. Since then, many advances have been made in modeling computer systems with queuing networks, faster solution techniques, and accurate approximation techniques [JAIN90; LAZO84; MOLL89; SAUR81].

Queueing network models are commonly used to model computer systems for capacity planning. A capacity-planning model is constructed from specifications for the computer system configuration and measurements of resource requirements for each of the workloads modeled. The model is solved and the resulting performance metrics (response time, throughput, resource utilization, etc.) are compared to measured performance. The model is calibrated to the computer system. Then, it is used to study the effect of increases in workload and resource demands, and of configuration changes.

Initially, queueing network models were used primarily for capacity planning. For SPE they were sometimes used for feasibility analysis: request arrivals and resource requirements were estimated and the results assessed. More precise models were infeasible because the software could not be measured until it was implemented.

The second SPE modeling breakthrough was the introduction of analytical models for software [BEIZ78; BOOT79a; BOOT79b; BOOT80; SANG78; SMIT79a; TRIV82]. With them, software execution is modeled, estimates of resource requirements are made, and performance metrics are calculated. Software execution models yield an approximate value for best, worst, or average resource requirements (such as CPU usage or number of I/O operations). They provide an estimate for response time; they can detect response time problems, but because they do not model resource contention they do not yield precise values for predicted response time.

The third SPE modeling breakthrough was combining the analytic software models with the queueing-network system models to more precisely model execution characteristics [BGS; SMIT80b; SMIT80a]. Combined models more precisely model the execution. They also show the effect of new software on existing work and on resource utilization. They identify computer device bottlenecks and the parts of the new software with high use of bottleneck devices.

By 1980, the modeling power was established and modeling tools were available[1] [BGS; QSP; SES]. Thus, it became cost-effective to model large software systems early in their development.

2.2 SPE Methods

Early experience with a large system confirmed that sufficient data could be collected early in development to predict performance bottlenecks [SMIT82b]. Unfortunately, despite the predictions, the system design was not modified to remove them and upon implementation (approximately one year later) performance in those areas was a serious problem, as predicted. SPE methods were proposed [SMIT81] and later updated [SMIT90a] to address the reasons that early predictions of performance problems were disregarded. Key parts of the process are methods for collecting data early in software development, and critical success factors to ensure SPE success. The methods also address compatibility with software engineering methods, what is done, when, by whom, and other organizational issues. SPE methods are described in Section 3.

2.3 SPE Development

The 1980s brought advances in all facets of SPE. Software model advances were proposed by several authors [BEIZ84; BOOT86; ESTR86; QIN89; SAHN87; SMIT82a; SMIT90a; SMIT82c]. Martin proposed data-action graphs as a representation that facilitates transformation between performance models and various software design notations [MART88]. Opdahl and Sølvberg integrate information system models and performance models with extended specifications [OPDA92b]. Rolia extends the SPE models and methods to address systems of cooperating processes in a distributed and multicomputer environment with specific applications to Ada [ROLI92]. Woodside [WOOD86; WOOD89] proposes stochastic rendezvous networks to evaluate performance of Ada systems and Woodside and coworkers incorporate the analysis techniques into a software engineering tool [BUHR 89; WOOD91]. Opdahl [OPDA92a] describes SPE tools interfaced with the PPP (processes, phenomena, programs) CASE tool and the IMSE environment for performance modeling – both are part of the Esprit research initiative. Opdahl

[1] Many new tools are now available.

[OPDA92] develops analytic sensitivity analysis techniques to point out where refinement and parameter capture efforts should be focused, and to suggest improvements in the design specification. Lor and Berry [LOR91] automatically generate SARA (Systems ARchitects Apprentice) models from an arbitrary requirements specification language using a knowledge-based Design Assistant. Other tools that incorporate features to support SPE modeling are reported by numerous authors [BAGR91; BEIL88; GOET90; NICH90; NICH91; POOL91; SMIT91; SMIT90b].

Extensive advances have been made in computer system performance modeling techniques. A complete list of references is beyond the scope of this tutorial.

Bentley [BENT82] proposes a set of rules for writing efficient programs. A set of formal, general principles for performance-oriented design is reported by Smith [SMIT86b; SMIT88a; SMIT90a]. Software architects who are experts in formulating requirements and designs, and developers who are experts in data structure and algorithm selection, use intuition to develop their systems. The rules and principles formalize the expert knowledge developed through years of experience, for use by software developers with less experience. Additional information on the rules and principles and related work is in Section 4.

Fox [FOX87; FOX89] describes middle lifecycle SPE activities. He emphasizes that models alone are insufficient and that measurement and analysis of evolving software are essential to meeting performance objectives. Bell [BELL87] advocates techniques for *system* performance engineering. He covers middle lifecycle techniques and focuses on the overall system, not just the software.

Numerous authors relay experience with SPE [ALEX86; ALEX82; ANDE84; BELL88; BELL87; FOX87; FOX89; PATE91; SMIT85B; SMIT88B] The *Proceedings of the Computer Measurement Group Conferences* contain SPE experience papers each year [CMG].

SPE has also been adapted to real-time process control systems. When these systems must respond to events within a specified time interval, they are called *reactive systems*. Howes [HOWE90] prescribes principles for developing efficient reactive systems. Goettge applies SPE to a reactive system using a performance engineering tool with an expert system for suggesting performance improvements [BREH92; GOET90]. Williams and Smith describe the SPE process for a case study of a reactive system [SMIT93]. Sholl and Kim [SHOL86] adapt the computation-structure approach to real-time systems, and LeMer [LEME82] describes a methodology and tool. Joseph and Pandya claim a computationally efficient technique for finding the exact worst-case response time of real-time systems [JOSE86]. Valderruten and coworkers derive performance models from formal specifications [VALD92]. Chang and coworkers [CHAN89] use petri net model extensions to evaluate real-time systems. Baldasarri and coworkers integrate a petri net design notation into a CASE tool that provides performance results [BALD89]. As mentioned earlier, much of the other reported work on real-time systems is outside the scope of SPE's software-oriented work.

3 The SPE Process

Figure 2 depicts the SPE process; the following paragraphs explain the figure.

First, developers define the specific SPE assessments for the current software lifecycle 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 if

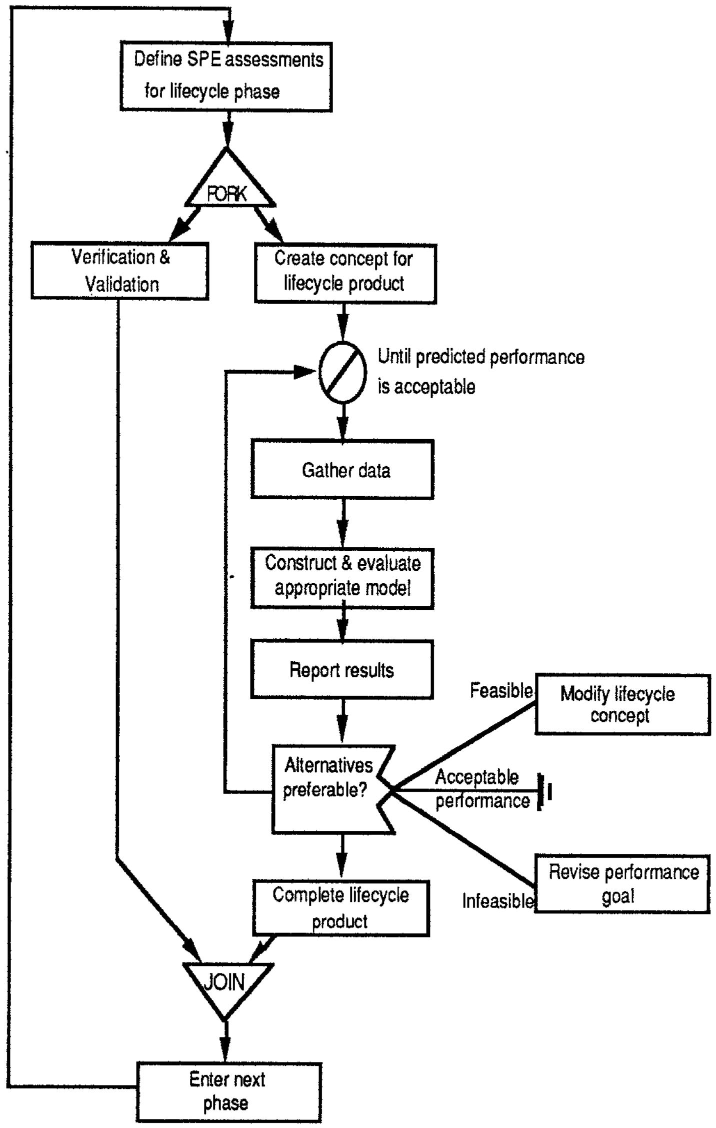
Define SPE assessments for lifecycle phase
FORK
Verification & Validation
Create concept for lifecycle product
Until predicted performance is acceptable
Gather data
Construct & evaluate appropriate model
Report results
Feasible
Modify lifecycle concept
Alternatives preferable?
Acceptable performance
Infeasible
Revise performance goal
Complete lifecycle product
JOIN
Enter next phase

Fig. 2. Software Performance Engineering Process

analysts are to determine concretely whether that objective can be met. A crisp definition of the performance objectives lets developers determine the most appropriate means of achieving objectives, and avoid spending time overachieving them.

Business systems specify performance objectives in terms of *responsiveness* as seen by the system users. Reactive systems specify timing requirements for event responses. Both the response time for a 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 lifecycle product. For early phases the "concept" is the functional architecture – the software requirements and the high-level plans for satisfying requirements. In subsequent phases the "concept" is a more detailed design, the algorithms and data structures, the code, etc. Developers apply SPE's general principles (described later) to create responsive functional architectures and designs.

Once the lifecycle concept is formulated, analysts gather data sufficient for estimating the performance of the concept proposal. First analysts need the projected system workload: how it will *typically* be used. They also need an explanation of the current design concept. Early in development, analysts use the general system architecture; later models incorporate the proposed decomposition into modules; still later, they add the proposed algorithms and data structures. Early analysis of reactive systems also examines typical usage; later analyses examine worst-case and failure scenarios. Estimates of the resource usage of the design entities complete the specifications. Section 5 provides more details on data requirements and techniques for gathering specifications.

Because the precision of the model results depends on the precision of the resource estimates, and because these are difficult to estimate very early in software development, a *best and worst-case analysis strategy* is integral to the methodology. Analysts 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, they seek feasible alternatives. If the worst-case performance is satisfactory, they proceed with development. If the results are between the two extremes, models identify critical components whose resource estimates have the greatest effect, and focus turns to 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.

An overview of the construction and evaluation of the performance models follows in Section 5. 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, reports show 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* before the lifecycle product is complete. If none is feasible as, for example, when the modifications would cause an unacceptable delivery delay, developers 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. Reports compare the model specifications for the workload, software structure, execution structure, and resource requirements to actual usage and software characteristics. If necessary, analysts calibrate the model by adjusting model parameters until they represent the system behavior for a variety of loading conditions. They also examine discrepancies to update the performance predictions, and to identify the reasons for differences – to prevent similar problems occurring in the future. They produce reports comparing system execution model results (response times, throughput, device utilization, etc.) to measurements. Analysts study discrepancies, identify error sources, and calibrate the model as necessary. Model verification and validation (V&V) should begin early and continue throughout the lifecycle. In early stages, focus is on key performance factors; prototypes or benchmarks provide more precise specifications and measurements as needed. The evolving software becomes the source of the model verification and validation data.

This discussion outlined the steps for one design-evaluation "pass." The steps repeat throughout the lifecycle. For each pass the goals and the evaluations change somewhat. For additional information on SPE methods for lifecycle stages and the questions to be considered refer to [SMIT90a; SMIT93].

4 General Principles for Creating Responsive Systems

Engineering new software systems is an iterative process of refinement. At each refinement step, engineers *understand* the problem, *create* a hypothetical solution, describe or *represent* the proposed product of the creation step, and then *evaluate* that products' appropriateness. The evaluation assesses a design's *correctness* (adherence to specification), *feasibility* (cost, time, and technology to implement), and the *preferability* of one solution over another (simplicity, maintainability, usability, and so on). *Responsiveness* may be a correctness assessment (for reactive systems), or a feasibility assessment (will the architecture support the performance requirements), or a preferability assessment (when other requirements are met, engineers and users prefer a more responsive alternative).

Several software engineering methods advocate a software design process with these steps [ALFO85; BALD89; BELL77; RIDD78; WINC82]. Lampson [LAMP84] presents an excellent collection of hints for computer system design that addresses effectiveness, efficiency, and correctness. His efficiency hints are the type of folklore that has until recently been informally shared. Kopetz [KOPE86] presents principles for constructing real-time process control systems; some address responsiveness – all address performance in the more general sense.

Alter [ALTE79] and Kant [KANT81] take a different approach; they use program optimization techniques to generate efficient programs from logical specifications. Search techniques identify the best strategy from various alternatives for choices such as data set organizations, access methods, and computation aggregations.

Authors address program efficiency from three perspectives: efficient algorithms and data structures [BENT82; KNUT68; KNUT73]; efficient coding techniques [JALI77; MCNE80; VANT78]; and techniques for tuning existing programs to improve efficiency [BENT82; FERR78; FERR83b; KNUT71]. The program efficiency techniques evolved first. Later, the techniques evolved to address large-scale *systems* of programs in early life-cycle stages when developers seek requirements and design specifications that will lead to *systems* with acceptable responsiveness [SMIT86b; SMIT88a; SMIT90a]. During early stages, it is seldom the efficiency of machine resource usage that matters; it is the system *responsiveness*. Another distinction between system design and program tuning approaches is that program tuning transforms an inefficient program into a new "equivalent" program that performs the same function more efficiently. In system design, developers can transform *what* the software is to do as well as how it is to be done.

SPE uses both the early lifecycle system principles and the later lifecycle program design techniques. Section 4.1 gives a summary of general principles that apply to the requirements and design creation steps to identify alternatives that are likely to meet responsiveness objectives and to refine concepts that require improved responsiveness. Section 4.2 cites sources of principles that apply to implementation steps to identify desirable algorithms and data structures.

To experienced performance analysts, Lampson's hints, and these synthesis principles are not revolutionary new prescriptions. They are, however, a generalization and abstraction of the "expert knowledge" that performance specialists use in building or tuning systems.

The principles *supplement* performance assessment rather than replace it. Performance improvement has many tradeoffs – a local performance improvement may adversely affect overall performance. The quantitative methods covered in section 5 provide the data required to evaluate the net performance effect to be weighed against other aspects of correctness, feasibility, and preferability.

4.1 Early Lifecycle Performance Principles

Smith [SMIT90a] defines the following seven principles. A quantitative analysis of the performance results of three of them are in Smith [SMIT86b]. Both Smith [SMIT88a] and Smith [SMIT90a] give extensive explanations and examples for each principle.

Fixing-Point Principle. Fixing connects the desired action or function to the instructions that accomplish the action. Fixing also connects the desired result to the data used to produce it. The fixing "point" is a point in time. The latest fixing point is during execution immediately before the instructions or data are required. Fixing could establish connections at several earlier points: earlier in the execution, at program initiation time, at compilation time, or outside the software system.

Suppose users need summary data of multiple account detail records. The latest fixing point summarizes the data at the time users request summary-data screens; earlier fixing updates the summary data as account detail records arrive. The principle is as follows:

> *Fixing-Point Principle:* For responsiveness, fixing should establish connections at the earliest feasible point in time, such that retaining the connection is cost-effective.

It is cost-effective to retain the connection when the savings realized with it offset the retention cost. In the summary-data example, the retention cost is the cost of the storage to hold summary data. Assume that the data is saved for other purposes anyway, then there is no additional retention cost. The operational costs are roughly the same – to summarize upon request, the software must locate and read each detail record, then write one summary record. To update summary data as detail records arrive, there is one locate and one write per detail record. Thus, in this case early fixing is warranted because the responsiveness is better – users do not have to wait for the summary-data calculation.

Locality-Design Principle. Locality refers to the closeness of desired actions, functions, and results to physical resources. For example, if a desired screen result is identical to the physical database row that produces it, the locality is good. According to Webster, close means "being near in *time, space, effect* (that is, purpose or intent), or *degree* (that is, intensity or extent)."

The dictionary specification for *close* leads to four types of locality design for performance engineering. These are illustrated in the following example. Consider the logical task to sort a list of names. *Temporal locality* is better if the names are all sorted at the same time rather than sorting a few, and frequently adding a few more names to the list and sorting again. *Spatial locality* is better if the names are near the processor that conducts the sort, such as in the processor's local memory, rather than on a disk drive attached to a different machine. The sort can execute on different types of physical processors; *effectual locality* is better if the processor can sort long character strings directly, rather than breaking strings into smaller "processor-size" pieces (such as words or bytes). *Degree locality* is better if the entire list of names fits in memory rather than requiring intermediate storage on disk. The principle is as follows:

> *Locality Design Principle:* Create actions, functions, and results that are "close" to physical computer resources.

Processing Versus Frequency Tradeoff Principle. This principle addresses the amount of work done per processing request, and its impact on the number of requests made. The "tradeoff" looks for opportunities to reduce requests by doing more work per request and vice versa. The principle is as follows:

Processing versus Frequency Tradeoff Principle: Minimize the processing times frequency product.

When software adds many rows to a database, two design alternatives are (1) to execute the database load commands once per row; and (2) to collect the changes then execute the database load command once for the entire batch. The processing versus frequency tradeoff principle compares the total cost of the alternatives. If the software executes on a client platform, and the database resides on a server, the communication overhead processing is part of the total cost.

Shared-Resource Principle. Computer system resources are limited; thus, processes compete for their use. Some resources may be *shared:* Multiple processes can use the resource at the same time. Other resources require *exclusive use:* Processes take turns — each process has exclusive use of the resource, one at a time. Exclusive use affects performance in two ways: the additional processing overhead to schedule the resource and the possible contention delay to gain access to the resource. The contention delay depends on how many processes request exclusive use and the time they hold the resource. The shared-resource principle is of the *synergistic type:* it improves overall performance, through cooperation to reduce contention delays, rather than by reducing individual processing like the first three *independent-type* principles. The general principle is as follows:

Shared-Resource Principle: Share resources when possible. When exclusive access is required, minimize the sum of the holding time and the scheduling time.

Resource sharing minimizes both the overhead for scheduling and the wait to gain access (there may be a wait if another process already has exclusive access even though the requester is willing to share). A database organization that keys on date and clusters transactions entered on the same date, does *not* promote "sharing when possible" because all additions during the day must lock the same portion of the database.

Parallel Processing Principle. Processing time can sometimes be reduced by partitioning a process into multiple concurrent processes. The concurrency can either be real concurrency in which the processes execute at the same time on different processors, or it can be apparent concurrency in which the processes are multiplexed on a single processor. For real concurrency, the processing time is reduced by an amount proportional to the number of processors. Apparent concurrency is more complicated: Although some of the processing may be overlapped (one process may use the CPU while another accesses the disk), each process may experience additional wait time. Both real and apparent concurrency require processing overhead to manage the communication and coordination between concurrent processes. The principle is as follows:

Parallel Processing Principle: Execute processing in parallel (only) when the processing speedup offsets communication overhead and resource contention delays.

The parallel processing principle is another synergistic principle. SPE performance models assess speedup, contention delays, and communication delays.

Centering Principle. The five previous principles provide guidance for creating software requirements and designs. Their application improves the performance of the part of the system to which they are applied. Centering is different in that it leverages performance by focusing attention on the parts of software systems that have the greatest impact on responsiveness.[2] Centering identifies the subset (≤ 20%) of the *system functions* that will be used most (≥ 80%) of the time. These frequently used functions are the *dominant workload functions*.[3] Performance enhancements made to these key areas of the software system thus greatly affect the overall responsiveness of the system. The principle is as follows:

> *Centering principle:* Identify the dominant workload functions and minimize their processing.

That is, create special, streamlined execution paths for the dominant workload functions that are customized and trimmed to include only processing that *must* be part of the function. Use the five previous principles to minimize processing of the special paths. Create separate transactions for the workload functions that are used less frequently.

Design the dominant workload functions first to increase the liklihood that the data organization matches the access patterns. Even though the dominant workload functions are usually trivial transactions – not the essence of the software design – they will likely have the greatest effect on the overall responsiveness of the system. Thus, they require early resolution.

Instrumenting Principle. "Instrumenting software" means inserting code at key probe points to enable the measurement of pertinent execution characteristics. The principle is:

> *Instrumenting Principle:* Instrument systems *as you build them* to enable measurement and analysis of workload scenarios, resource requirements, and performance goal achievement.

This principle does not in itself improve performance, but is essential to *controlling* performance. It has its foundations in engineering, particularly process control engineering. Their rule of thumb is "if you can't measure it, you can't control it."

---

[2] Centering is based on the folkloric "80-20 rule" for the execution of code within programs (which claims that ≤ 20 percent of a program's code accounts for ≥ 80 percent of its computer resource usage).

[3] The dominant workload functions also cause a subset (≤ 20%) of the programs or modules in the software system to be executed most (≥ 80%) of the time, and the code within modules, and so forth.

Data collection mechanisms are part of the system requirements and design; it is much more difficult to add it after implementation. This is because of limitations in instrumenting technology - most external measurement tools collect system-level data, such as program execution time, rather than functional data such as end-to-end response time (for business units of work that require multiple transactions). To collect functional data, programmers must insert code to call system timing routines, and write key events and pertinent data to files for later analysis. *Define these probe points when defining the functions.*

Heuristics for Real-Time Systems Design. Howes develops design heuristics for "demonstrably efficient" real-time systems to be implemented in Ada. In Howes and Weaver [HOWE89] the authors introduce the

> *Structuring Principle of Physical Concurrency:* Introduce concurrency only to model physically concurrent objects or processes.

This is a specific instance of the earlier, more general parallel processing principle. This principle provides guidance to real-time system developers for the efficient use of Ada rendezvous. It is an alternative to maximum concurrency which advocates concurrency as a goal in itself, and conceptual concurrency which uses concurrency to simplify a design. Their paper compares the efficiency of the three concurrency approaches.

Howes [HOWE90] introduces the

> *Tuning Principle:* Reduce the mean service time of cyclic functions.

This is also a specific instance of the more general centering and processing versus frequency principles. It guides real-time system developers to identify the functions that execute at regular, specific time intervals and minimize their processing requirements.

Howes' 1990 paper applies both principles to a small design problem and compares the results to those achieved with other design heuristics that do not address performance. The paper's conclusion indicates that the investigation of other performance principles is underway.

Summary. A quantitative analysis of the performance effect of the fixing-point, processing versus frequency tradeoff, and centering principles is in Smith [SMIT86b]. While these three can be evaluated with simple, back-of-the-envelope calculations, the others require more sophisticated models. Section 4 reviews the quantitative methods for evaluating responsiveness of alternatives.

4.2 Implementation principles

Implementation principles apply during the detail-design stage to guide the selection and implementation of proper data structures and algorithms for the critical modules.

The *execution environment* defines 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 modeling tools automatically provide much 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, analysts need: the approximate number of instructions executed (or CPU time required); the number of physical I/O's; and use of other key devices such as communication lines (terminal I/O's 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 the number of database calls and their characteristics are usually sufficient. Early lifecycle requirements are tentative, difficult to specify, and likely to change, so SPE uses upper and lower bound estimates to identify problem areas or software components that warrant further study to obtain more precise specifications.

The number of times a component executes or its resource usage may vary significantly. To represent the variability, analysts identify the factors that cause the variability, use a data dependent variable to represent each factor, and specify the execution frequency and resource usage in terms of the data dependent variables. Later, the models study the performance sensitivity to various parameter values.

It is seldom possible to get precise information for all these specifications early in the software's lifecycle. Rather than waiting to model the system until it is available (i.e., in detailed design or later), SPE suggests gathering guesses, approximations, and bounded estimates to begin, then augmenting the models as information becomes available. This approach has the added advantage of focusing attention on key workload elements to minimize their processing (as prescribed by the "centering principle" in section 3). Otherwise, designers tend to postpone these important performance drivers in favor of designing more interesting but less frequently used parts of the software.

Because one person seldom knows all the information required for the software performance models, *performance walkthroughs* provide most of the data [SMIT90a]. Performance walkthroughs are closely modeled after design and code walkthroughs. In addition to software specialists who contribute software plans they bring together users who contribute workload and scenario information, and technical specialists who contribute computer configuration and resource requirements of key subsystems such as DBMS and communication paths. The primary purpose of a performance walkthrough is data gathering rather than a critical review of design and implementation strategy.

These topics are covered by a variety of data structure texts (see, for example, [KNUT68; KNUT73; SCHN78; STAN80]). Bentley [BENT82] provides a systematic methodology and specific efficiency rules for implementing the data structures and algorithms.

5 Quantitative Methods for SPE

The quantitative methods prescribe the data required to conduct the performance assessment and techniques for gathering the data; the performance models and techniques for adapting them to the system evolution; and techniques for verification and validation of the models.

5.1 Data Requirements

To create a software execution model analysts need: *performance goals, workload definitions, software execution characteristics, execution environment descriptions,* and *resource usage estimates.* An overview of each follows.

Performance goals, precise, quantitative metrics, are vital to determine concretely whether or not performance objectives can be met. For business 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. Goal specifications 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 90th percentile, and so on. For reactive systems, timing constraints specify the maximum time between an event and the response. Some reactive systems have throughput goals for the number of events processed in a time interval.

Workload definitions specify the key scenarios of the new software. For on-line transactions scenarios initially specify the transactions expected to be the most frequently used. Later in the lifecycle, scenarios also cover resource intensive transactions. 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. Reactive systems represent scenarios of time-critical functions, and worst-case operating conditions.

Software execution characteristics identify components of the software system to be executed for each workload scenario. The software execution scenario identifies: software 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. Reactive systems initially specify the likely execution paths, and later add less likely, but feasible execution paths.

5.2 Performance Models

Two models provide the quantitative data for SPE: the *software execution model* and the *system execution model*. The software execution model represents key facets of software execution behavior. The model solution quantifies the computer resource requirements for each workload scenario. The system execution model represents computer system resources with a network of queues and servers. The model combines the workload scenarios and quantifies overall resource utilization and consequent response times of each scenario.

Execution graph models are one type of software execution model. They are not the only option, but are convenient for illustration. A graph represents each workload scenario. 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 [SMIT90a].

A static analysis of the graphs yields mean, best- and worst-case execution times of the scenarios. The static analysis makes the optimistic assumption that there are no other workloads on the host configuration competing for resources. Simple, graph-analysis algorithms provide the static analysis results [BEIZ78; BOOT79a; BOOT79b; SMIT90a].

Next, the system execution model solution yields the following additional information:

- The effect of resource contention on response times, throughput, device utilizations and device queue lengths.
- Sensitivity of performance metrics to variations in workload composition.
- Effect of new software on service level objectives of other existing systems.
- Identification of bottleneck resources.
- Comparative data on performance improvements to the workload demands, software changes, hardware upgrades, and various combinations of each.

To construct and evaluate the system execution model, analysts first represent the key computer resources with a network of queues, then use environment specifications to specify device parameters (such as CPU size and processing speed). The workload parameters and service requests come from the resource requirements computed from the software execution models. Analysts solve the model and check for reasonable results, then examine the model results. If the results show that the system fails to meet performance objectives, analysts identify bottleneck devices and correlate system execution model results with software components. After identifying alternatives to the software plans or the computer configuration, analysts evaluate the alternatives by making appropriate changes to the software or system model and repeating the analysis steps.

System execution models based on the network of queues and servers can be solved with analytic techniques in a few seconds. Thus analysts can conduct many tradeoff studies in a short time. Analytic solution techniques generally yield utilizations within 10 percent, and response times within 30 percent of actual. Thus, they are well-suited to early lifecycle studies when the primary objective is to identify feasible alternatives and rule out alternatives that are unlikely to meet performance goals. The resource usage estimates that lead to the model parameters are seldom sufficiently precise to warrant the additional time and effort required to produce more realistic models. Even reactive systems benefit from this intermediate step – it rules out serious problems before proceeding to more realistic models.

Early studies typically use simple SPE models. The SPE goal is to find problems with the simplest possible model. Experience shows that simple models can detect serious performance problems very early in the development process. The simple models isolate the problems and focus attention on their resolution (rather than on assumptions used for more realistic models). After they serve this primary purpose, analysts augment them as the software evolves to make more realistic performance predictions.

SPE calls for matching the modeling effort to the precision of the available data. The effort depends on the sophistication of the modeling tools available and their solution speed. As new tools become available and processor speeds increase, the shift from simple, optimistic models to more detailed, realistic models is adapted accordingly. Likewise, the shift from analytic to simulation solutions is adapted accordingly. Analytic solutions produce mean-value results which may not adequately reflect a system's performance if it has periodic problems or unusual execution characteristics.

Advanced system execution models are usually appropriate when the software reaches the detail-design lifecycle stage. Even when it is easy to incorporate the additional execution characteristics earlier, it is better to defer them to the advanced system execution model. It is seldom easy, however, and the time to construct and evaluate the advanced models usually does not match the input data precision early in the lifecycle.

The modeling difficulty arises from several sources. They are described next. First, advances in modeling technology follow the introduction of new hardware and operating system features. So accurate models of new computer system resources are active research topics. For example, I/O subsystems may have channel reconnect, path selection when multiple channel paths are available, disk cache controllers, etc. Models of these resources are evolving.

The second modeling difficulty is that special software features such as lock-free, acquire-release, etc., require *passive resources* – resources that are required for processing, but which do no work themselves. They are held while the software uses one or more active resources; the queueing delays for active resources influence the duration of passive resource usage. The impact of these passive resource delays is two-fold. The response of jobs forced to wait on the passive resource will be slower, but because these waiting jobs do not use active resources while they wait, other jobs may execute at a faster rate due to the decreased contention. Passive resource delays are difficult to quantify with analytic queueing network models: quantitative data for queueing delays requires a queue-server node, but its service time depends on the time spent in other queue-servers.

A third modeling difficulty is that computer system environment characteristics (such as distributed processing, parallel processing, concurrent and multi-threaded software, and memory management overhead) challenge model technology. They either use passive resources, have complex model topologies, or tightly couple the models of software and system execution.

These three facets of execution behavior are represented in the advanced system execution model. It augments the elementary system execution model with additional types of constructs. Then procedures specify how to calculate corresponding model parameters from software models, and how to solve the advanced models. SPE methods specify "checkpoint evaluations" to identify those aspects of the execution behavior that require closer examination [SMIT90a].

Recent modeling research results have simplified the analysis of advanced system execution models with passive resources [ROLI92; WOOD89; WOOD91]. Details of these models are beyond the scope of this tutorial. Information on system and advanced system execution models is available in books [JAIN90; LAZO84; MOLL89; SAUR81] as well as other publications. The *Proceedings of the ACM SIGMETRICS Conferences* and the *Performance Evaluation Journal* report recent research results in advanced systems execution models. The *Proceedings of the Performance Petri Net Conferences* also report relevant results.

5.3 Verification and Validation

Another vital part of SPE is continual verification of the model specifications and validation of model predictions (V&V). It begins early, particularly when model results suggest that major changes are needed. The V&V effort matches the impact of the results and the importance of performance to the project. When performance is critical, or major software changes are indicated, analysts identify the critical components, implement or prototype them, and measure. Measurements verify resource usage and path execution specifications and validate model results.

Early V&V is important even when predicted performance is good. Performance specialists influence the values that developers choose for resource usage estimates, and specialists tend to be optimistic about how functions will be implemented and about their resource requirements. Resource usage of the actual system often differs significantly from the optimistic specifications.

Performance engineers interview users, designers, and programmers to confirm that usage will be as expected, and that designed and coded algorithms agree with model assumptions. They adjust models when appropriate, revise predictions, and give regular status reports. They also perform sensitivity analyses of model parameters and determine thresholds that yield acceptable performance. Then, as the software evolves and code is produced, they measure the resource usage and path executions and compare with these thresholds to get early warning of potential problems. As software increments are deployed, measurements of the workload characteristics yield comparisons of specified scenario usage to actual and show inaccuracies or omissions. Analysts calibrate models and evaluate the effect of model changes on earlier results. As the software evolves, measurements replace resource estimates in the verification and validation process.

V&V is crucial to SPE. It requires the comparison of multiple sets of parameters, for heavy and light loads, to corresponding measurements. The model precision depends on how closely the model represents the key performance drivers. It takes constant vigilance to make sure they match.

6 Lifecycle SPE

The previous sections outline the SPE steps: define objectives, apply principles to formulate performance-oriented concepts, gather data, model, evaluate, and measure to verify model fidelity, validate model predictions, and confirm that software meets performance objectives. The steps are repeated throughout the lifecycle. The goals and the evaluation of the objectives change somewhat for each pass.

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.

During later lifecycle stages performance engineers adapt the SPE methods to the lifecycle stage. As more information about the proposed software becomes available, the quantitative methods produce more precise performance metrics and support implementation trade-off decisions. Performance measurements confirm that the software performs as expected and augment earlier model estimates. Refer to [SMIT93] for more information on activities in later lifecycle stages.

7 Status and Future of SPE

Since computers were invented, the attitude persists that the next hardware generation will offer significant cost-performance enhancements, so it will no longer be necessary to worry about performance. There was a time, in the early 1970's, when computing power exceeded demand in most environments. The cost of achieving performance goals, with the tools and methods of the era, made SPE uneconomical for many batch systems – its cost exceeded its savings. Today's methods and tools make SPE the appropriate choice for many new systems. Will tomorrow's hardware solve all performance problems and make SPE obsolete? It has not happened yet. Hardware advances merely make new software solutions feasible, so software size and sophistication offset hardware improvements. There is nothing wrong with using more powerful hardware to meet performance objectives, but SPE suggests evaluating all options early, and selecting the most effective one. Thus hardware may be the solution, but it should be explicitly chosen – early enough to enable orderly procurement. SPE still plays a role.

The three primary elements in SPE's evolution are the *models* for performance prediction, the *tools* that facilitate the studies, and the *methods* for applying them to systems under development. With these the *use* of SPE increases, and new design *concepts* develop that lead to high-performance systems. Future evolution in each of these five areas will change the nature of SPE but not its underlying philosophy to build performance into systems. The following paragraphs speculate on future developments in each of these areas.

Both research and development will produce the *tools* of the future. We seek better integration of the models and their analysis with software engineering tools such as specification languages, CASE tools, and automatic program generators. Then software changes automatically update prediction models. Simple models can be transparent to designers – designers could click a button while formulating designs and view automatically generated performance predictions. Expert systems could suggest performance improvements [GOET90]. Visual user interfaces could make analysis and reporting more effective. Software measurement tools could capture, reduce, interpret, and report data at a level of detail appropriate for designers. Measurement tools could automatically generate performance tests and compare specifications to measurements, compare predictions to actual performance, and report discrepancies. Each of these tools could interface with an SPE database to store evolutionary design and model data and support queries against it.

While simple versions of each of these tools are feasible with today's technology, research must establish the framework for fully functional versions. For example, if a CASE tool supports data flow diagrams and structure charts, how does software automatically convert them to software models? How should performance models integrate with specification languages or automatic program generators – should one begin with models and generate specifications or code from them, or should one write the specifications and let underlying models select efficient implementations, or some other combination? How can expert systems detect problems? Can software automatically determine from software execution models where instrumentation probes should be inserted? Can software automatically reduce data to appropriate levels of detail? Can software automatically generate performance tests? Each of these topics represents extensive research projects.

Performance models currently have limits in their ability to analytically solve models of tomorrow's complex environments. Models of extensive parallel or distributed environments must be hand-crafted, with many checkpoint evaluations tailored to the problem. More automatic solutions are desired. Secondly, the analytic queueing network models yield only mean value results. Analysts need to quickly and easily model transient behavior to study periodic behavior or unusual execution characteristics. For example, averaged over a 10-hour period, locking effects may be insignificant, but there may be short 1 minute intervals in which locks cause all other active jobs to "log jam," and it may take 30 minutes for the log jam to clear. Mean value results do not reflect these after-effects; transient behavior models could. Petri nets and simulation offer more of the desired capabilities than analytical methods. Finally, as computer environments evolve, model technology must also develop. Thus, research opportunities are rich in software and computer environment models.

Technology transfer suggests that the *use of SPE* is likely to spread. More literature documenting SPE experiences is likely to appear. As it is applied to new, state-of-the-art software systems, new problems will be discovered that require research solutions. Future SPE applications will require skills in multiple domains and offer many new learning opportunities. As quantitative models evolve, so will the use of SPE for new problem domains. For example, models of software reliability have matured enough to be integrated with other SPE methods. Similarly, models can also support hardware-software codesign and enable software versus hardware implementation choices early in development [FRAN85].

The *concepts for building high performance systems* will evolve as SPE use spreads. Experience will lead to many examples illustrating the difference between high and low- performance software that can be used to educate new software engineers. SPE quantitative techniques should be extended to build in other quality attributes, such as reliability, availability, testability, maintainability, etc. Research in these areas is challenging – for example: Do existing software metrics accurately represent quality factors? Can one develop predictive models? What design data will drive the models? What design concepts lead to improved quality?

SPE methods should undergo significant change as its usage increases. The methods should be better integrated into the software development process, rather than an add-on activity. SPE should become better integrated into capacity planning as well. As they become integrated, many of the pragmatic techniques should be unnecessary (how to convince designers there is a serious problem, how to get data, etc.). The nature of SPE should then change. Performance walkthroughs will not be necessary for data gathering; they may only review performance during the course of regular design walkthroughs. The emphasis will change from finding and correcting design flaws to verification and validation that the system performs as expected. Additional research into automatic techniques for measuring software designs is needed, for calibrating models, and for reporting discrepancies.

SPE methods need to evolve from a general methodology with numerous examples (many in the business systems area) to a more exhaustive set of procedures based on system types. For a particular type of system: Is a standard set of performance requirements appropriate? Which performance metrics are relevant? Which design principles and rules of thumb are most important? What specific SPE steps should be conducted at each lifecycle stage? Which modeling techniques and tools are most appropriate to represent pertinent system characteristics? What specific measurements are needed and which tools provide the data?

For certain real-time systems, particularly those with mission-critical or safety-critical performance requirements, SPE procedures must be rigorously defined. The SPE results must also be defined and *reviewable* so inspectors can determine that SPE was properly conducted and the system will meet its performance requirements [SMIT92b].

Thus, the research challenges for the future are to extend the quantitative methods to model the new hardware-software developments, to extend hardware-software measurement technology to support SPE, and to develop interdisciplinary techniques to address the more general definition of performance. The challenges for future technology transfer are to automate the sometimes cumbersome SPE activities, and to evolve SPE to make it easy and economical for future environments.

Acknowledgement
This tutorial is an excerpt from the article "Software Performance Engineering" that appears in *The Encyclopedia of Software Engineering*, John Wiley and Sons, 1993.

BIBLIOGRAPHY

[ALEX86] C.T. Alexander, "Performance Engineering: Various Techniques and Tools," *Proceedings Computer Measurement Group Conference*, Las Vegas, NV, Dec. 1986, pp. 264-267.

[ALEX82] William Alexander and Richard Brice, "Performance Modeling in the Design Process," *Proceedings National Computer Conference*, Houston, TX, June 1982.

[ALFO85] M. Alford, "SREM at the Age of Eight: The Distributed Computing Design System," *IEEE Computer*, vol. 18, no. 4, April 1985.

[ALTE79] S. Alter, "A Model for Automating File and Program Design in Business Application Systems," *Communications of the ACM*, vol. 22, no. 6, June 1979, pp. 345-353.

[ANDE84] Gordon E. Anderson, "The Coordinated Use of Five Performance Evaluation Methodologies," *Communications of the ACM*, vol. 27, no. 2, Feb. 1984, pp. 119-125.

[BAGR91] R.L. Bagrodia and C. Shen, "MIDAS: Integrated Design and Simulation of Distributed Systems," *IEEE Transactions on Software Engineering*, vol. 17, no. 10, Oct. 1991, pp. 1042-58.

[BALD89] M. Baldassari, et al., "PROTOB: A Hierarchical Object-Oriented CASE Tool for Distributed Systems," *Proceedings European Software Engineering Conference - 1989*, Coventry, England, Sept. 1989.

[BASK75] F. Baskett, et al., "Open, Closed, and Mixed Networks of Queues with Different Classes of Customers," *Journal of the ACM*, vol. 22, no. 2, Apr. 1975, pp. 248-260.

[BEIL88] Heinz Beilner, J. Mäter, and N. Weissenburg, "Towards a Performance Modeling Environment: News on HIT," *Proceedings 4th International Conference on Modeling Techniques and Tools for Computer Performance Evaluation*, Plenum Publishing, 1988.

[BEIZ78] Boris Beizer, *Micro-Analysis of Computer System Performance*, New York, NY, Van Nostrand Reinhold, 1978.

[BEIZ84] Boris Beizer, "Software Performance," in *Handbook of Software Engineering*, C.R. Vicksa and C.V. Ramamoorthy, ed., New York, NY, Van Nostrand Reinhold, 1984, pp. 413-436.

[BELL88] Thomas E. Bell, guest editor, *Computer Measurement Group Transactions*, Spring, 1988.

[BELL77] Thomas E. Bell, D.X. Bixler, and M.E. Dyer, "An Extendible Approach to Computer-aided Software Requirements Engineering," *IEEE Transactions on Software Engineering*, vol. 3, no. 1, Jan. 1977, pp. 49-59.

[BELL87] Thomas E. Bell and A.M. Falk, "Performance Engineering: Some Lessons From the Trenches," *Proceedings Computer Measurement Group Conference*, Orlando, FL, Dec. 1987, pp. 549-552.

[BENT82] Jon L. Bentley, *Writing Efficient Programs*, Englewood Cliffs, NJ, Prentice-Hall, 1982.

[BGS] BGS Systems, Inc., 128 Technology Center, Waltham, MA 02254, (617)891-0000.

[BOOT79a] Taylor L. Booth, "Performance Optimization of Software Systems Processing Information Sequences Modeled by Probabilistic Languages," *IEEE Transactions on Software Engineering*, vol. 5, no. 1, Jan. 1979, pp. 31-44.

[BOOT79b] Taylor L. Booth, "Use of Computation Structure Models to Measure Computation Performance," *Proceedings Conference on Simulation, Measurement, and Modeling of Computer Systems*, Boulder, CO, Aug. 1979.

[BOOT86] Taylor L. Booth, R.O. Hart, and Bin Qin, "High Performance Software Design," *Proceedings Hawaii International Conference on System Sciences*, Honolulu, HI, Jan. 1986, pp. 41-52.

[BOOT80] Taylor L. Booth and C.A. Wiecek, "Performance Abstract Data Types as a Tool in Software Perfromance Analysis and Design," *IEEE Transactions on Software Engineering*, vol. 6, no. 2, Mar. 1980, pp. 138-151.

[BREH92] Eric W. Brehm, Robert T. Goettge, and Frederick W. McCaleb, "START/ES — An Expert System Tool for System Performance and Reliability Analysis," *Proceedings Modelling Techniques and Tools for Computer Performance Evaluation*, Rob Pooley and Jane Hillston, ed., Edinburgh Scotland, September 1992, pp. 151-165.

[BUHR 89] R.J. Buhr, et al., "Software CAD: A Revolutionary Approach," *IEEE Transactions on Software Engineering*, vol. 15, no. 3, Mar. 1989, pp. 234-249.

[BUZE71] Jeffrey P. Buzen, "Queueing Network Models of Multiprograming," Ph.D. Dissertation, Harvard University, 1971.

[CHAN89] C.K. Chang, et al., "Modeling a Real-Time Multitasking System in a Timed PQ Net," *IEEE Transactions on Software Engineering*, vol. 6, no. 2, March 1989, pp. 46-51.

[CMG] Computer Measurement Group, 414 Plaza Dr. Suite 209, Westmont, IL 60559, (708)655-1812.

[ESTR86] G. Estrin, et al., "SARA (System ARchitects' Apprentice): Modeling, Analysis, and Simulation Support for Design of Concurrent Systems," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 2, Feb. 1986, pp. 293-311.

[FERR78] Domenico Ferrari, *Computer Systems Performance Evaluation*, Englewood Cliffs, NJ, Prentice-Hall, 1978.

[FERR83b] Domenico Ferrari, Giuseppe Serazzi, and Alessandro Zeigner, *Measurement and Tuning of Computer Systems*, Englewood Cliffs, NJ, Prentice-Hall, 1983.

[FOX87] Gregory Fox, "Take Practical Performance Engineering Steps Early," *Proceedings Computer Measurement Group Conference*, Orlando, FL, Dec. 1987, pp. 992-993.

[FOX89] Gregory Fox, "Performance Engineering as a Part of the Development Lifecycle for Large-Scale Software Systems," *Proceedings 11th International Conference on Software Engineering*, Pittsburgh, PA, May 1989, pp. 85-94.

[FRAN85] Geoff A. Frank, Connie U. Smith, and John L. Cuadrado, "Software/Hardware Codesign with an Architecture Design and Assessment System," *Proceedings Design Automation Conference*, Las Vegas, NV, 1985.

[GABR90] Armen Gabrielian and Matthew K. Franklin, "Multi-Level Specification and Verification of Real-Time Software," *Proceedings Twelfth International Conference on Software Engineering*, Nice, France, Apr. 1990, pp. 52-62.

[GOET90] Robert T. Goettge, "An Expert System for Performance Engineering of Time-Critical Software," *Proceedings Computer Measurement Group Conference*, Orlando FL, 1990, pp. 313-320.

[GRAH73] R.M. Graham, G.J. Clancy, and D.B. DeVaney, "A Software Design and Evalation System," *Communications of the ACM*, vol. 16, no. 2, Feb. 1973, pp. 110-116.

[HOWE90] Norman R. Howes, "Toward a Real-Time Ada Design Methodology," *Proceedings Tri-Ada 90*, Baltimore, MD, Dec. 1990.

[HOWE89] Norman R. Howes and Alfred C. Weaver, "Measurements of Ada Overhead in OSI-Style Communications Systems," *IEEE Transactions on Software Engineering*, vol. 15, no. 12, Dec. 1989, pp. 1507-1517.

[JAFF91] Matthew S. Jaffe, et al., "Software Requirements Analysis of Real-Time Process Control Systems," *IEEE Transactions on Software Engineering*, vol. 17, no. 3, Mar. 1991, pp. 241-258.

[JAHA87] Farnam Jahanian and Aloysius K.L. Mok, "A Graph-Theoretic Approach for Timing Analsis and its Implementation," *IEEE Transactions on Computers*, vol. C-36, no. 8, Aug. 1987, pp. 961-975.

[JAIN90] R. Jain, *Art of Computer Systems Performance Analysis*, New York, NY, John Wiley, 1990.

[JALI77] Paul J. Jalics, "Improving Performance The Easy Way," *Datamation*, vol. 23, no. 4, Apr. 1977, pp. 135-148.

[JOSE86] M. Joseph and P. Pandya, "Finding Response Times in a Real-Time System," *The Computer Journal*, vol. 29, no. 5, 1986, pp. 390-395.

[KANT81] Elaine Kant, *Efficiency in Program Synthesis*, Ann Arbor, MI, UMI Research Press, 1981.

[KNUT68] Donald E. Knuth, *The Art of Computer Programming, Vol.1: Fundamental Algorithms*, Reading, MA, Addison-Wesley, 1968.

[KNUT71] Donald E. Knuth, "An Empirical Study of FORTRAN Programs," *Software Practice & Experience*, vol. 1, no. 2, Apr. 1971, pp. 105-133.

[KNUT73] Donald E. Knuth, *The Art of Computer Programming, Vol.3: Sorting and Searching*, Reading, MA, Addison-Wesley, 1973.

[KOPE86] Herman Kopetz, "Design Principles of Fault Tolerant Real-Time Systems," *Proceedings Hawaii International Conference on System Sciences*, Honolulu, HI, Jan. 1986, pp. 53-62.

[KOPE85] H. Kopetz and W. Merker, "The Architecture of Mars," *Proceedings FTCS 15*, Ann Arbor, MI, IEEE Press, June 1985, pp. 274-279.

[KOPE91] H. Kopetz, et al., "The Design of Real-Time Systems: From Specification to Implementation and Verification," *Software Engineering Journal*, 1991, pp. 72-82.

[LAMP84] Butler W. Lampson, "Hints for Computer System Design," *IEEE Software*, vol. 2, no. 1, Feb 1984, pp. 11-28.

[LAZO84] Edward D. Lazowska, et al., *Quantitative System Performance: Computer System Analysis Using Queuing Network Models*, Englewod Cliffs, NJ, Prentice-Hall, Inc., 1984.

[LEME82] Eric LeMer, "MEDOC: A Methodology for Designing and Evaluating Large-Scale Real-Time Systems," *Proceedings National Computer Conference, 1982*, Houston, TX, 1982, pp. 263-272.

[LEVI90] S. Levi and A.K. Agrawala, *Real-Time System Design*, New York, NY, McGraw-Hill, 1990.

[LOR91] K. Lor and D.M. Berry, "Automatic Synthesis of SARA Design Models from System Requirements," *IEEE Transactions on Software Engineering*, vol. 17, no. 12, Dec. 1991, pp. 1229-1240.

[MART88] Charles R. Martin, "An Integrated Software Performance Engineering Environment," Masters Thesis, Duke University, 1988.

[MCNE80] M. McNeil and W. Tracy, "PL/I Program Efficiency," *SIGPLAN Notices*, vol. 15, no. 6, June 1980, pp. 46-60.

[MOLL89] Michael K. Molloy, *Fundamentals of Performance Modeling*, MacMillan, 1989.

[NICH90] Kathleen M. Nichols, "Performance Tools," *IEEE Software*, vol. 7, no. 3, May 1990, pp. 21-30.

[NICH91] Kathleen M. Nichols and Paul Oman, "Special Issue in High Performance," *IEEE Software*, vol. 8, no. 5, 1991.

[OPDA92a] A. Opdahl, "A CASE Tool for Performance Engineering During Software Design," *Proceedings Fifth Nordic Workshop on Programming Environmental Research*, Tampere, Finland, Jan. 1992.

[OPDA92b] A. Opdahl and A. Sølvberg, "Conceptual Integration of Information System and Performance Modeling," *Proceedings Working Conference on Information System Concepts: Improving the Understanding*, 1992.

[OPDA92] Andreas L. Opdahl, "Sensitivity Analysis of Combined Software and Hardware Performance Models: Open Queueing Networks," *Proceedings Modelling Techniques and Tools for Computer Performance Evaluation*, Rob Pooley and Jane Hillston, ed., Edinburgh, September 1992, pp. 257-271.

[PATE91] M. Paterok, R. Heller, and H. deMeer, "Performance Evaluation of an SDL Run Time System - A Case Study," *Proceedings 5th International Conference on Modeling Techniques and Tools for Computer Performance Evaluation*, Torino, Italy, Feb. 1991, pp. 86-101.

[POOL91] R. Pooley, "The Integrated Modeling Support Environment," *Proceedings 5th International Conference on Modeling Techniques and Tools for Computer Performance Evaluation*, Torino, Italy, Feb. 1991, pp. 1-15.

[POSP92] Gustav Pospischil, et al., "Developing Real-Time Tasks with Predictable Timing," *IEEE Software*, vol. 9, no. 5, Sept. 1992, pp. 35-50.

[QIN89] Bin Qin, "A Model to Predict the Average Response Time of User Programs," *Performance Evaluation*, vol. 10, 1989, pp. 93-101.

[QSP] Quantitative System Performance, 7516 34th Ave N., Seattle, WA 98117-4723.

[RIDD78] W.E. Riddle, et al., "Behavior Modeling During Software Design," *IEEE Transactions on Software Engineering*, vol. 4, 1978.

[ROLI92] J.A. Rolia, "Predicting the Performance of Software Systems," University of Toronto, 1992.

[SAHN87] Robin A. Sahner and Kishor S. Trivedi, "Performance and Reliability Analysis Using Directed Acyclic Graphs," *IEEE Transactions on Software Engineering*, vol. 13, no. 10, Oct. 1987, pp. 1105-1114.

[SANG78] John W. Sanguinetti, "A Formal Technique for Analyzing the Performance of Complex Systems," *Proceedings Performance Evaluation Users Group 14*, Boston, MA, Oct. 1978.

[SAUR81] C.H. Sauer and K.M. Chandy, *Computer Systems Performance Modeling*, Englewood Cliffs, NJ, Prentice-Hall, 1981.

[SCHN78] G.M. Schneider, S.W. Weingart, and D.M. Perlman, *An Introduction to Programming and Problem Solving with Pascal*, New York, NY, John Wiley and Sons, 1978.

[SES] Scientific and Engineering Software, 4301 West Bank Drive, Bldg A., Austin, TX 78746, (512)328-5544.

[SHA90] Lui Sha and John B. Goodenough, "Real-Time Scheduling Theory and Ada," *IEEE Computer*, vol. 23, no. 4, Apr. 1990, pp. 53-62.

[SHIN92] Kang G. Shin, et al., "A Distributed Real-Time Operating System," *IEEE Software*, vol. 9, no. 5, Sept. 1992, pp. 58-68.

[SHOL86] H. Sholl and S. Kim, "An Approach to Performance Modeling as an Aid in Structuring Real-time, Distributed System Software," *Proceedings Hawaii International Conference on System Sciences*, Honolulu, HI, Jan. 1986, pp. 5-16.

[SHOL75] H.A. Sholl and Taylor L. Booth, "Software Performance Modeling Using Computation Structures," *IEEE Transactions on Software Engineering*, vol. 1, no. 4, Dec. 1975.

[SMIT79a] Connie U. Smith and J.C. Browne, "Performance Specifications and Analysis of Software Designs," *Proceedings ACM Sigmetrics Conference on Simulation Measurement and Modeling of Computer Systems*, Boulder, CO, Aug. 1979.

[SMIT80a] Connie U. Smith and J.C. Browne, "Aspects of Software Design Analysis: Concurrency and Blocking," *Proceedings ACM Sigmetrics Conference on Simulation Measurement and Modeling of Computer Systems*, May 1980.

[SMIT80b] Connie U. Smith, "The Prediction and Evaluation of the Performance of Software from Extended Design Specifications," Ph.D. Dissertation, University of Texas, 1980.

[SMIT81] Connie U. Smith, "Software Performance Engineering," *Proceedings Computer Measurement Group Conference XII*, Dec. 1981, pp. 5-14.

[SMIT82a] Connie U. Smith, "A Methodology for Predicting the Memory Management Overhead of New Software Systems," *Proceedings Hawaii International Conference on System Sciences*, Honolulu, HI, Jan. 1982, pp. 200-209.

[SMIT82b] Connie U. Smith and J.C. Browne, "Performance Engineering of Software Systems: A Case Study," *Proceedings National Computer Conference*, Houston, TX, vol. 15, June 1982, pp. 217-224.

[SMIT82c] Connie U. Smith and David D. Loendorf, "Performance Analysis of Software for an MIMD Computer," *Proceedings ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, Seattle, WA, Aug. 1982, pp. 151-162.

[SMIT85b] Connie U. Smith, guest editor, *Computer Measurement Group Transactions*, 1985.

[SMIT86b] Connie U. Smith, "Independent General Principles for Constructing Responsive Software Systems," *ACM Transactions on Computer Systems*, vol. 4, no. 1, Feb. 1986, pp. 1-31.

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

[SMIT88b] Connie U. Smith, "Who Uses SPE?," *Computer Measurement Group Transactions*, Spring 1988, pp. 69-75.

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

[SMIT90b] Connie U. Smith and Lloyd G. Williams, "Why CASE Should Extend into Software Performance," *Software Magazine*, vol. 10, no. 9, 1990, pp. 49-65.

[SMIT91] Connie U. Smith, "Integrating New and 'Used' Modeling Tools for Performance Engineering," *Proceedings 5th International Conference on Modeling*

Techniques and Tools for Computer Performance Evaluation, Torino, Italy, Feb. 1991.

[SMIT92b] Connie U. Smith, "Software Performance Engineering in the Development of Safety-Critical Systems," No.92-03, L&S Computer Technology, November, 1992.

[SMIT93] Connie U. Smith, "Software Performance Engineering," in *The Encyclopedia of Software Engineering*, John Wiley and Sons, 1993.

[SMIT93] Connie U. Smith and Lloyd G. Williams, "Software Performance Engineering: A Case Study with Design Comparisons," *IEEE Transactions on Software Engineering*, to appear 1993.

[STAN80] T.A. Standish, *Data Structure Techniques*, Reading, MA, Addison-Wesley, 1980.

[STAN91] John A. Stankovic and Krithi Ramamritham, "The Spring Kernel: A New Paradigm for Real-Time Systems," *IEEE Software*, vol. 8, no. 3, Mar. 1991, pp. 62-72.

[TRIV82] Kisor S. Trivedi, *Probability and Statistics With Reliability, Queueing, and Computer Science Applications*, Englewood Cliffs, NJ, Prentice-Hall, 1982.

[VALD92] Alberto Valderruten, et al., "Deriving Queueing Networks Performance Models from Annotated LOTOS Specifications," *Proceedings Modelling Techniques and Tools for Computer Performance Evaluation*, Rob Pooley and Jane Hillston, ed., Edinburgh, September 1992, pp. 167-178.

[VANT78] D. Van Tassel, *Program Style, Design, Efficiency, Debugging, and Testing*, Englewood Cliffs, NJ, Prentice-Hall, 1978.

[WINC82] J.W. Winchester and G. Estrin, "Methodology for Computer-based Systems," *Proceedings National Computer Conference*, vol. 51, 1982, pp. 369-79.

[WOOD86] C.M. Woodside, "Throughput Calculation for Basic Stochastic Rendezvous Networks," Technical Report, Carleton University, Ottawa, Canada, April, 1986.

[WOOD89] C.M. Woodside, "Throughput Calculation for Basic Stochastic Rendezvous Networks," *Performance Evaluation*, vol. 9, 1989.

[WOOD91] C.M. Woodside, et al., "The CAEDE Performance Analysis Tool," *Ada Letters*, vol. XI, no. 3, Spring 1991.

[XU87] J. Xu and D.L. Parnas, "On Satisfying Timing Constraints in Hard Real-Time Systems," *Proceedings ACM SIGSOFT 91 Conference on Software for Critical Systems*, New Orleans, LA, vol. 16, 1991, pp. 132-145.