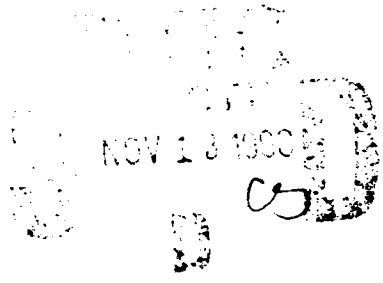
Technical Document 1834
June 1990

Performance Engineering for Mission Critical Embedded Computer Systems



L&S Computer Technology, Inc.

NAVAL OCEAN SYSTEMS CENTER
San Diego, California 92152-5000

J. D. FONTANA, CAPT, USN
Commander

R. M. HILLYER
Technical Director

ADMINISTRATIVE INFORMATION

Contract N00039-86-C-0247 was carried out by L&S Computer Technology, Inc., P.O. Box 9802, Austin, TX 78766, for the Office of Naval Technology, Arlington, VA 22217, under the technical coordination of T. Sterrett, Computer Systems Software and Technology Branch, Code 411, Naval Ocean Systems Center, San Diego, CA 92152-5000.

Released by
R. A. Wasilausky, Head
Computer Systems Software
and Technology Branch

Under authority of
A. G. Justice, Head
Information Processing and
Displaying Division

NOTE

FS

Executive Summary

This report provides background information on performance engineering and the POD performance modeling tool, and gives an overview of the project activities. Finally, the project summary section reviews the results, lessons learned, and suggests future directions. A detailed review of the project activities is in Appendix A.

Naval mission critical, embedded computer systems (MC-ECS) must respond to external events within their allotted time, otherwise they fail. Failures may have life or death consequences. Lifecycle performance management, or *performance engineering* (PE), calls for building performance into systems beginning in the requirements definition phase, and continuing the performance management through the design, implementation, testing, and post-deployment phases. Experience with PE shows that it can detect and avoid project-threatening performance failures in sufficient time to correct them and enable timely delivery of a quality product. Furthermore, performance is *orders of magnitude better* with this approach than with a "fix it later" approach in which performance considerations are deferred to the testing phase and, when necessary, "tuning" attempts to correct performance failures. Better performance means both people and computer resources can be used to enhance the functionality of the system rather than to correct performance deficiencies.

Performance of MC-ECS has always been important. Developers traditionally had highly-skilled software engineers who were experts in building efficient software. They were successful in building high-performance systems; however, the result of this so-called "guru approach" is an over-reliance on these same gurus to maintain the systems. (One naval organization actually imposes travel restrictions to ensure that gurus do not travel on the same airplanes -- just in case). To assess performance, developers traditionally built hand-crafted simulation programs. The simulations were labor-intensive, inflexible, and often required as much development effort as the software system itself. Recent trends seek to use newer software development methods to improve productivity and software quality and decrease the over-dependence on gurus. Unfortunately, few of the new software methods address PE.[1]

[1]A prominent example is in a recent article on "Designing Large Real-time Systems with Ada," published in *Communications of the ACM* [NIE87]. The authors begin with a definition of real-time systems which emphasizes the importance of performance. Then the article proposes a design methodology and provides an example that completely ignore performance.

Embedded systems of the future need PE more, not less. Ada offers new capabilities for designers, but it also *dramatically increases the risk of performance failures.* Performance implications of Ada code are not obvious; the number of Ada tasks, their allocation to distributed processors, and their relative priorities have dramatic consequences on performance that are impossible to evaluate without performance models; and extensive error control and other run-time features make the compiled code inherently less efficient than the hand-crafted code in older systems. Furthermore, few of today's software engineers have the performance intuition of their predecessors and they have not yet gained the first-hand experience with Ada (they tend to create systems with far too many tasks and thus rendezvous) -- so performance failures are more likely.

We cannot rely on hardware to resolve all performance problems. Size, weight, and cost restrictions preclude adding extra hardware late in the system lifecycle to achieve performance goals. Hardware requirements must be determined early -- models are essential to accurate hardware sizing. Even though hardware technology is rapidly evolving, more powerful processors result in a dramatic increase in software size and complexity. The extra capacity is quickly consumed with more sophisticated functions and algorithms. New VLSI technology makes custom chip design viable, but software must exploit the new technology to realize dramatic improvements.

Previous Navy-sponsored research produced the performance modeling tool, POD. It enables PE by enabling performance analysts to quickly construct and evaluate models of predicted performance. It uses quick, analytic model solutions that match the solution technique to the amount and precision of information available in the lifecycle. The simple models identify areas that warrant the time and effort required for detailed simulation studies. POD is the most powerful PE tool currently available and is far more cost-effective than the hand-crafted simulation models of the past. Nevertheless, POD does not enjoy widespread use for Naval MC-ECS development. Both PE and POD must be used if they are to be effective for building MC-ECS that meet performance objectives. POD users need to know how to model systems under development and how to use the tool. They also need to understand PE: the necessary steps and why, when, and how to perform them. It is vital to specifically relate these topics to MC-ECS systems so the users have a clear understanding of the applicability of PE and POD to their unique problems.

To remedy this situation the Navy has taken initial steps towards technology transfer of PE for MC-ES. Together, we have applied POD to an MC-ECS case study that evaluated a new algorithm for computing target data in the F/A-18 software. We modeled the algorithms and used POD to study the performance of numerous alternatives, such as: applying the algorithm to multiple targets, varying the number of integration steps, examining start-up versus steady-state processing, and varying hardware speed. The case study had no particular performance problems; the tool confirmed that there should be no surprises.

We also gathered historical MC-ECS specific examples of the use of PE and POD and used them to construct a set of examples that demonstrates how to construct and evaluate models, how to collect data, how to identify and analyze alternatives, how to present results, validate models, and other PE steps. The examples, combined with PE course materials adapted to MC-ECS concerns, were combined and the prototype course was presented to selected attendees. We evaluated the results of the course to determine whether we successfully achieved the proof of concept that we sought.

Project summary

The project had three main thrusts: (1) to apply PE and POD to an actual case study, learn from the experience, identify PE and POD requirements for MC-ECS, and integrate the results into the technology transfer materials; (2) to customize PE and POD technology transfer materials to specifically address issues of concern to MC-ECS developers; (3) to deliver the prototype technology transfer course.

1. **Case study success?**

The selected case study demonstrated the applicability of PE and POD to actual MC-ECS development problems. It successfully provided materials for the technology transfer course. We identified tool requirements for: ease of use, new reports to support typical analyses, and the need to apply the tool to many, actual case studies. Tool use for actual studies ensures that it is tailored to typical problems and detects errors (in the tool) triggered by operational data that may not be detected during regular testing.

2. **Customize PE and POD to MC-ECS systems.**
The materials in the prototype course demonstrated the range of possibilities and addressed several important classes of MC-ECS applications. More actual case studies are needed to drive the technology transfer as well as ensuring that the methods and the tools are suited to the problems.

3. **Technology transfer delivery.** The class was small, the attendees were knowledgeable and familiar with problems in developing MC-ECS. They were unfamiliar with PE and POD and gained a great deal of knowledge from the class. We succeeded with the proof of concept of PE and POD. The pilot course encountered several problems with the use of the tool; they were subsequently corrected and should not affect future technology transfer efforts. Because the first delivery was a prototype version of the course, we expected to learn such lessons from it -- and we did.

The project led to several technology transfer discoveries that need remedies. (1) Experienced designers, familiar with the hand-crafted simulation studies of the past, incorrectly perceive that analytic models do not apply to their systems. (2) There is no established process and procedure that prescribe the role of PE and modeling in system development. (3) While experienced modelers have no difficulty, the prototype version of POD may be too difficult for some designers to use. It should be better integrated with CASE development tools.

Future directions

Although the technology transfer course is not yet mature, it has evolved to a point that it provides useful information to developers of MC-ECS systems. It should be offered to more attendees while it evolves with more case studies.

The use of PE and POD on MC-ECS should continue to evolve. More case studies are needed, and both the tools and the PE methods need enhancements. The following needs surfaced during the course of the project and are likely to be important to the performance analysis of future systems:

- Better analysis tools to determine processing frequency and relative task priorities. They should be visual tools with an easy method of specification.

- Extensions to models to evaluate federated systems of processors and their inherent bus or network contention.

- Models that explicitly represent data structures and evaluate their effect on processing requirements.

- Additional solution methods to evaluate: periodic jobs, Ada rendezvous impact, data latency limits, locking, task structures and processor allocations.

- Improved POD usability: fine tune the reports, add graphics-based specification and evaluation, and automate typical PE analyses. An important future consideration is to determine the proper POD platform -- while UNIX-based systems are prevalent in the research communities, they are not currently in widespread use among MC-ECS developers. The adoption and widespread use of the Desk-TopII (Sun 4) as a Navy standard computer would make UNIX a viable choice for the future.

- Integrate the performance modeling tools with CASE tools to reuse design information, automatically generate and update models, and provide performance predictions directly to developers.

- Integrate the PE methods into the new Navy system development methodology so the steps become part of the standard development lifecycle -- rather than an add-on activity.

APPENDIX A
Project activities

1. **Review background information:** June 9 - July 12, 1988.
Studied reports documenting previous F/A-18 software modeling work. Reviewed POD User Manual with emphasis on features that support real time systems evaluation. Prepared simple test models.

2. **Identify and collect preliminary case study information:** China Lake, July 12-14, 1988.
This visit is documented in the July 1988 trip report (Appendix B). Learned about F/A-18 software evaluation problems in general, gathered background information on AYK-14 hardware and software execution environment, and preliminary information on the new algorithm to be evaluated.

3. **Research Review Briefing for NOSC Sponsors:** Monterey, CA, August 9, 1988.
Presented overview of POD and its importance for Software Performance Engineering of Navy Systems, and in particular for Mission Critical Embedded Computer Systems. Discussed technology transfer issues and how they are addressed by this project. Proposed two future research directions: evaluating system *effectiveness*, and features to enhance POD usability. The presentation slides are in Appendix C.

4. **Refine case study data:** China Lake, August 10-11, 1988.
Met with Charles Bechtel, Ken Trieu, and Roy Crosbie. Discussed details of the case study and the performance data required. Charles Bechtel subsequently prepared an *excellent* report with processing details and performance specifications (see Appendix D). Discussed modeling considerations and tool capabilities with Roy Crosbie.

5. **Prepare preliminary outline for technology transfer:** August 31, 1988.
Progress on installing POD was slow. To expedite the project, the preliminary outline was prepared earlier than originally planned. It adopted the L&S standard Software Performance Engineering seminar framework, and enhanced it to focus on MC-ECS system issues, include POD laboratory exercises, and F/A-18 and other case studies.

6. **Project review meeting:** Santa Fe, September 1-2, 1988.
Tony Sterrett represented NOSC and Bob Westbrook, represented China Lake. We reviewed the case study status - all were impressed with the thoroughness of Bechtel's report. We discussed the preliminary technology transfer outline and formulated initial answers to its questions. We agreed that the technology transfer should evolve as experience is gained with its presentation. We also collaborated on the POD installation to resolve problems.

7. **Install POD and become familiar with its features:** completed September 10, 1988.
The MacII A/UX version successfully executed the 3 simple test models, and the more complex distributed processing model contributed by Sterrett. Commands and reporting features were examined.

8. **F/A-18 case study - model formulation, testing and documentation**: October 1988.
The original processing and the adaptations for the two algorithm alternatives were modeled, and we assessed POD capabilities and limitations for the case study. It is feasible to use POD for this application - it can detect performance problems due to processing that exceeds the 50 ms. threshold, and evaluate the performance of: the number of integration steps, the adaptation to multiple targets, varying hardware speed, examining steady-state versus startup processing, and other similar studies. Several minor POD problems were detected, documented and submitted to NOSC and BGS. Although the code problems were relatively minor, we were unfortunately unable to derive model results in sufficient time to provide performance feedback to developers. NOSC personnel subsequently elected to create an improved version of POD to support the technology transfer project.

9. **Research Review Briefing for NOSC Sponsors:** San Diego, March 1989.
Reviewed the project status, some insights into transfer of PE and POD technology for MC-ECS, and some suggestions for future directions to enhance the transfer. The slides are in Appendix E, and the insights and directions were reviewed in this report's summary.

10. **Review POD-related technical reports and prepare case studies for technology transfer:** April 1989.
Created case study materials based on the FAA Air traffic control studies, a combat system design, and a signal processing application.

11. **Prepare final outline and technology transfer materials:** July 1989.

Adapted the PE course materials, and the POD usage materials to specifically address MC-ECS. Created case study materials and laboratory exercises. Appendix F contains the course materials.

12. **Conduct technology transfer:** August 1989.

The pilot version of the technology transfer course was offered at China Lake for attendees selected by NOSC. The feedback from the course indicated that they all gained an understanding of the PE process, the role of models during the entire development lifecycle, and the potential for tools such as POD to support development efforts.

APPENDIX B

July 1988 Trip Report

Trip Report
China Lake
July 12-14, 1988
Connie U. Smith

Meetings with:
Sponsor: Dr. Robert McWilliams
Consultant: Dr. Roy Crosbie

Tuesday, July 12

1. Introduction to Naval Weapons Center and the Aircraft Weapons Integration Dept., Embedded Computing Technology Office (Code 31C) by Jay Crawford.

2. Software Performance Engineering briefing by Connie Smith.

3. Overview of "Computing Problems in Tactical Aircraft" Project conducted summer, 1987 by Dr. Robert McWilliams, Dr. Roy Crosbie, and Linda Roush. Discussion of characteristics of F-18 software and typical performance concerns.

4. Discussion of applicability of operational analysis and Software Performance Engineering techniques to embedded computer systems with Dr. Ed Kutchma. Discussed typical performance concerns and candidate case studies.

5. Discussion of F-18 Software Development Branch activities and typical performance concerns with Mike Spencer, Dr. Ken Trieu, Charles Bechtel, and Dick Nuckles.

6. Identified candidate list of case studies (attached).

Wednesday, July 13

1. Reviewed F-18 documentation with Dr. Roy Crosbie. Most of the day was devoted to this information gathering task. Extracted background information on the AYK14 and the software from documents. Searched (unsuccessfully) for data on the Bus Architecture and details of how the Mission Control computers and software use the bus to send and receive data with other attached devices.

2. Discussion of future Ada plans and possible future SPE / Ada performance concerns with Dr. Lee Lucas.

3. Discussion with Dale Christenson on the extent of the use of software design methods and CASE tools at China Lake. Discussed current analysis and design activities. No recent or current high-level design activities suitable for a technology transfer case study were identified. Most of this type of work is done by McDonnell Douglas. Most Navy activities focus on modifications and extensions to existing software.

4. Demonstration and briefing on Simulation Lab activities by John Hessler. Investigated the potential of using the simulation software design as a case study. It is a very interesting application, but there are currently no performance issues and it is not representative of the majority of the work at NWC.

Thursday, July 14

1. Continued the documentation search and retrieval.

2. Met with Charles Bectel to discuss the "passive ranging algorithm" case study. Clarified some F-18 and AYK14 operational issues. Discussed the data necessary for the case study. Developed a list of items needed. They will gather information and send it within 2 weeks.

3. Met with Linda Roush to further discuss "Computing Problems in Tactical Aircraft." Some of the mission control functions have been off-loaded to the SMS (stores management) computer. Details are unfortunately not included in the Mission Computer Operational Flight Program Design Specification document. It will be difficult to precisely predict end to end responses without the characteristics of this additional work. It may be possible to obtain measurement data; otherwise, we will use a "microanalysis" performance goal: to complete each frame's processing within 50ms. She also suggested investigating the status of the Canadian efforts to create PSL/PSA structure charts and reports of the F-18 software.

4. Reviewed the demo of a subset of the F-18 software design in the Statemate CASE tool (under consideration by McDonnell Douglas). It provides useful high level design information. In the future it should be straightforward to evaluate performance using much of its design information. It is unclear when the complete system documentation might be available, but it is unlikely that it will help with this project.

Conclusions

The NWC personnel were extremely helpful. I learned a great deal about the F-18 software and the performance-related problems that are important to their software developers. We discussed the role of analytic models to support their trade-off studies. Coordination with Dr. Crosbie's project should leverage this modeling effort.

For technology transfer, the most appropriate case study should focus on typical Navy concerns such as estimating the impact of adding new functions to existing OFP's. The passive ranging algorithm is representative of these typical problems. Three alternative algorithms are under consideration, and the developers wish to know the performance impact of each alternative. It is an actual problem they now face and they are interested in the results. Ideally, we want an adaptive modeling study. We can first concentrate on the CPU time of each algorithm. To illustrate more complex studies, we can examine the performance impact of dividing the processing between frames. If possible, we can also analyze the effect of bus contention. This depends on data availability and the ability of POD to accurately represent the scheduling and contention. POD should give reasonable approximations.

We are tentatively pursuing option number 3 on the attached case study options list: the passive ranging algorithm. The technology transfer can suggest ways that SPE and POD can be used to study all of the problems. So in addition to getting actual performance data for one, all will be worked into the technology transfer materials.

Case Study Options

1. Compare growth of software from the 83X, 85X, and 87X releases, running on the AN/AYK-14 Model XN-5.

2. Evaluate exchange of algorithm for ballistic trajectory.

3. Evaluate addition of passive ranging algorithm.

4. Evaluate relative difference between XN-5 and XN-6 using same software.

5. Determine extent of night attack retrofit possible on F/A-18A/B using the XN5. (This proved to be inappropriate - night attack will use XN6).

6. Compare performance of single executive XN-6 with dual executive XN-6 for same software. (There is currently insufficient data for this study, but it would be an excellent case study to pursue later. A baseline model with concurrent processing would be beneficial for future algorithm trade-off studies.)

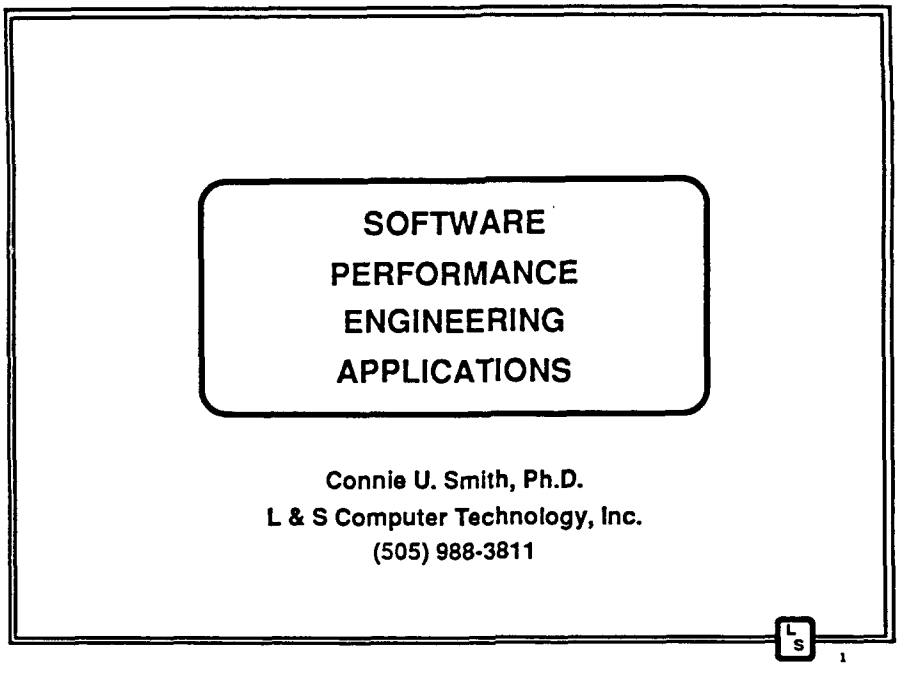
7. Evaluate 85X longest path problem (This is based on the timing problem studied in the "Computing Problems....." project.)

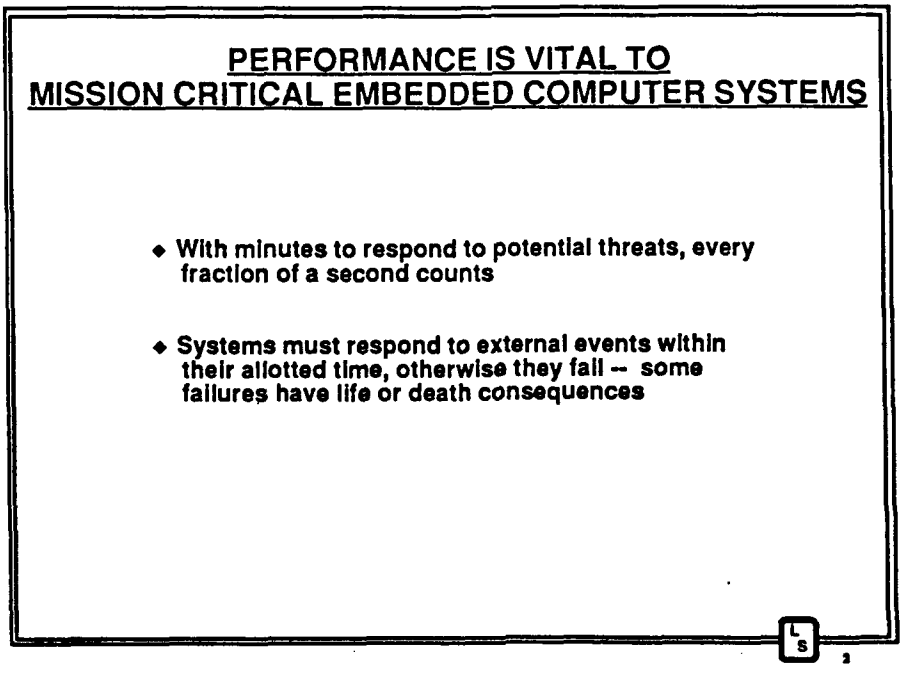
8. Simulation software for the "hardware in the loop simulation lab."

APPENDIX C

August 1988 Briefing

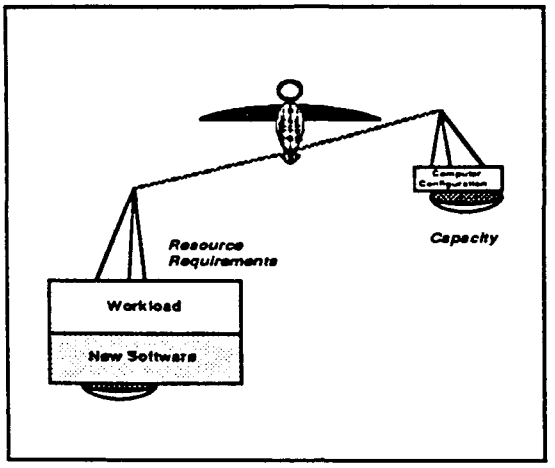
Software Performance Engineering Applications





Software Performance Engineering Applications

PERFORMANCE BALANCE



3

SOFTWARE PERFORMANCE ENGINEERING

"Systematic planning for and evaluation of software performance throughout its life cycle."

GOALS:

- **Enhance**
  - responsiveness
  - usability
  - staff productivity
- **Preserve**
  - maintainability
  - understandability
  - timely completion
- **Control**
  - computer capacity requirements

- **Build performance into systems rather than try to add it later**

4

Software Performance Engineering Applications

METHODOLOGY OVERVIEW

FOR each development stage DO
BEGIN

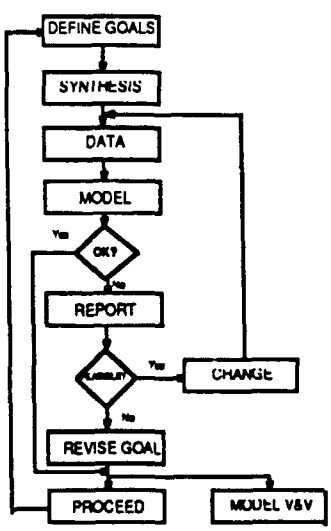
REPEAT
Develop design concept
Model it
Evaluate model solution

UNTIL predicted behavior is satisfactory

Proceed with design synthesis

END

SPE METHODOLOGY



- *Performance Walkthroughs*
- Data Gathering Strategies
- Compensation for Uncertainties
- Data Presentation & Tracking
- Model Verification & Validation
- General Principles for Performance-oriented Design

APPENDIX D

Case Study Report

F/A-18 Algorithm Analysis

This data is intended to answer at least some of the questions posed by Dr. C Smith at our last meeting.

Estimating the word count for each algorithm could be very difficult so I decided to simplify it a bit. I planned to count "high-level" instructions and then use an expansion factor (high-level to assembly) to determine the total number of assembly language instructions.

Determining a valid expansion factor is the hard part: some high-level instructions can be represented as single assembly instruction, but most require two or more instructions. Eventually I came up with the following method:

Assignment	Load (e.g. LD), Store (e.g. SD)	
Add/Subtract	Load, Add (e.g. AD), Rescale (e.g. LALD), Store	(1)
Multiply	Load, Multiply (e.g. MDR), Rescale, Store	
Divide	Load, Rescale, Divide (e.g. D), Store	

Note that the example instructions are for double precision (32-bit) integer arithmetic. The divide is not double precision: Most programmers will sacrifice some precision and utilize a single precision divide (it's 4x as fast). The AYK-14 XN-5 has no floating point unit (i.e. fixed point arithmetic is used throughout).

To "validate" the expansion factor I used data from a previous project. This implemented an algorithm which was first modelled in Fortran. The Fortran statements were classified and counted:

	High-Level	Est Asm	
Assignment	59	118	
Add/Subtract	57	228	
Multiply	28	112	
Divide	9	36	
Total	153	494	==> 3.22 Est Expansion

The project actually used 422 assembly language instructions (sorry - no break down into catagories) which results in an 2.76 expansion factor. Because the project concentrated on optimizing memory usage, I think that 2.75 is a little low for the average project. I believe the original assumption (1) will work.

The expansion factor mentioned from high-level statements to assembly level statements does not account for instructions which require two machine words. These words don't degrade execution speed (any more than the rates listed below) but they do take up more memory. In general, there is a 20%-50% increase from instruction count to memory requirements (e.g. 10 instructions may take 12 to 15 machine words).

Estimated execution performance for the assembly language statements is listed below (in microseconds):

INSTR	DESCRIPTION	XN-5	XN-6	
LD	Load Double	2.49	0.95	
SD	Store Double	2.68	2.10	
AD	Add Double	2.73	1.19	
LALD	Left Shift Dbl	1.89	1.11	
MD	Multiply Double	8.27	4.07	
D	Divide	9.87	4.38	(note single precision)
L	Load Single	2.24	0.80	
S	Store Single	1.86	1.15	
A	Add Single	2.24	0.94	
LALS	Left Shift Sngl	1.47	0.90	
M	Multiply Single	5.40	2.19	

The Algorithms -

Both algorithms will require an interface to the current program. This interface (setting data up etc.) has been estimated to require 1000 assembly words. No mix of statements has been given so I had to guess. My guess is the result of looking at an algorithm intended to function similarly to the two candidate algorithms. I counted the mix of high-level statements; this should be used to determine the overall makeup of this interface. I would assume that all of these instructions run during each pass.

	HOL	%total
Assignment	37	37%
Add/Subtract	20	20%

Software Performance Engineering Applications

MODELING REQUIREMENTS

- Low overhead
- Accommodate:
  - Incomplete definitions
  - Imprecise performance specs
  - Incremental development
  - Changes during development

GOAL:

Initially

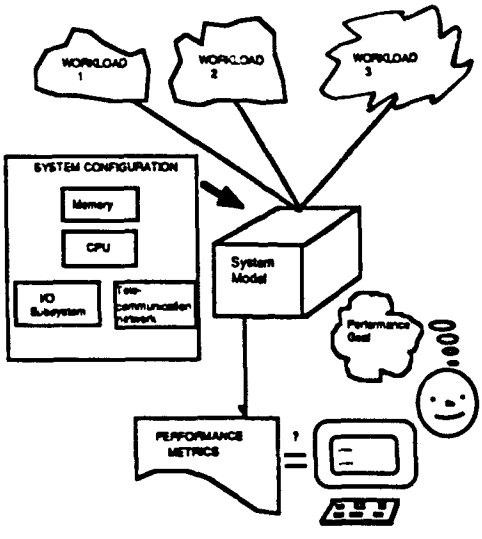
- ➤ Distinguish between "good" and "bad"

Later

- ➤ Increase precision
- ➤ Provide implementation decision support

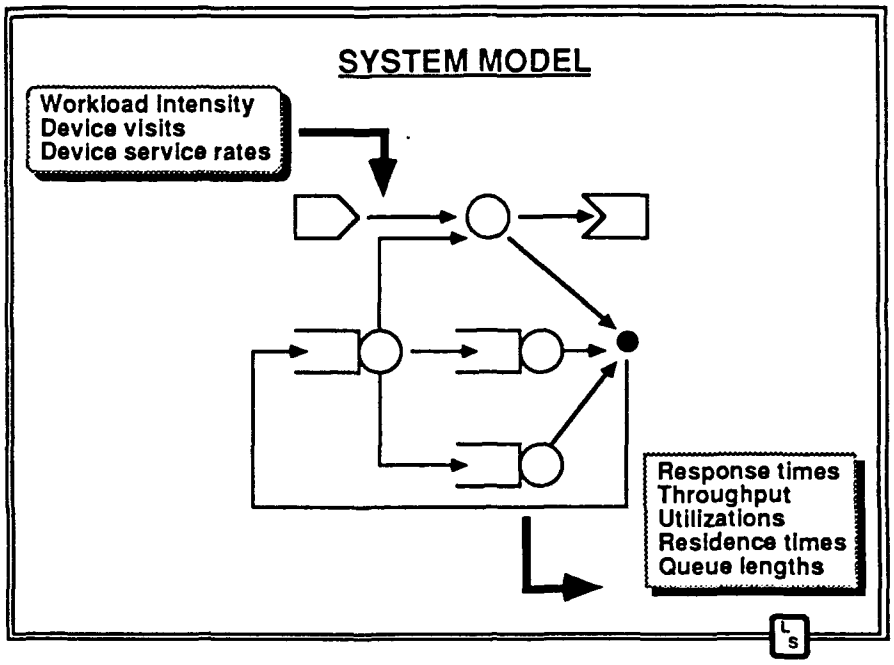
L S 7

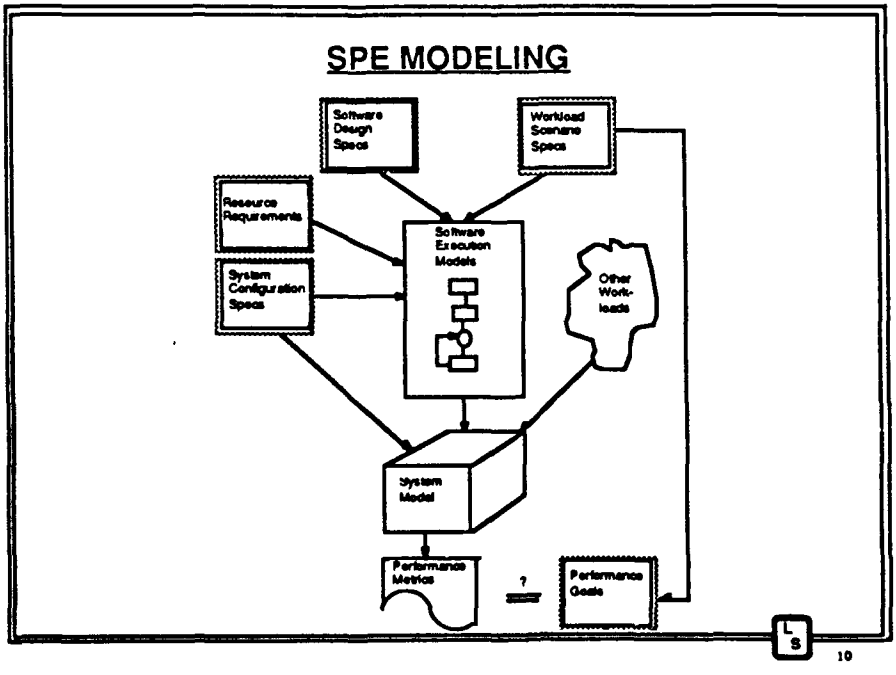
CONVENTIONAL PERFORMANCE MODELS



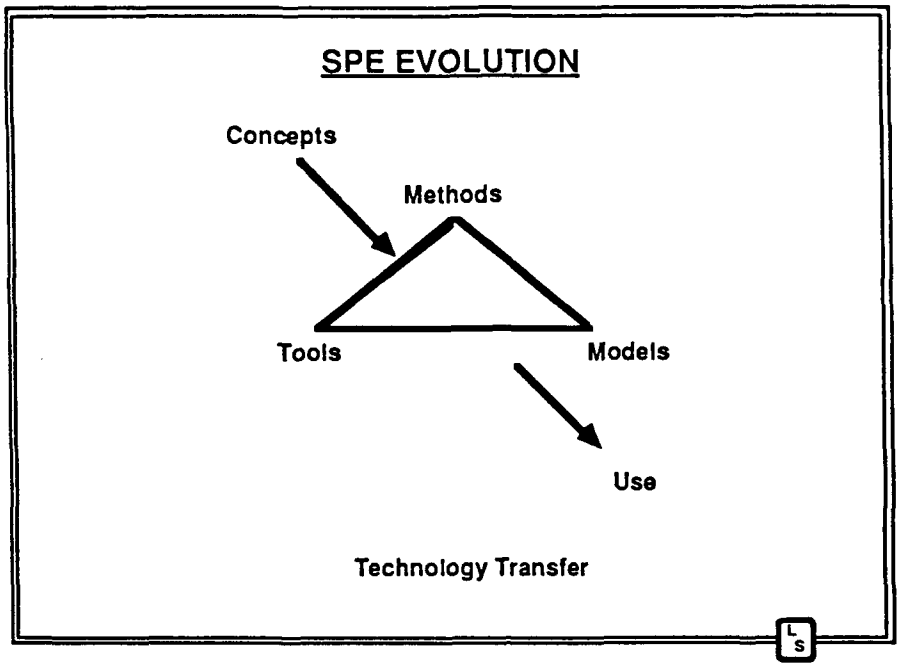
L S 8

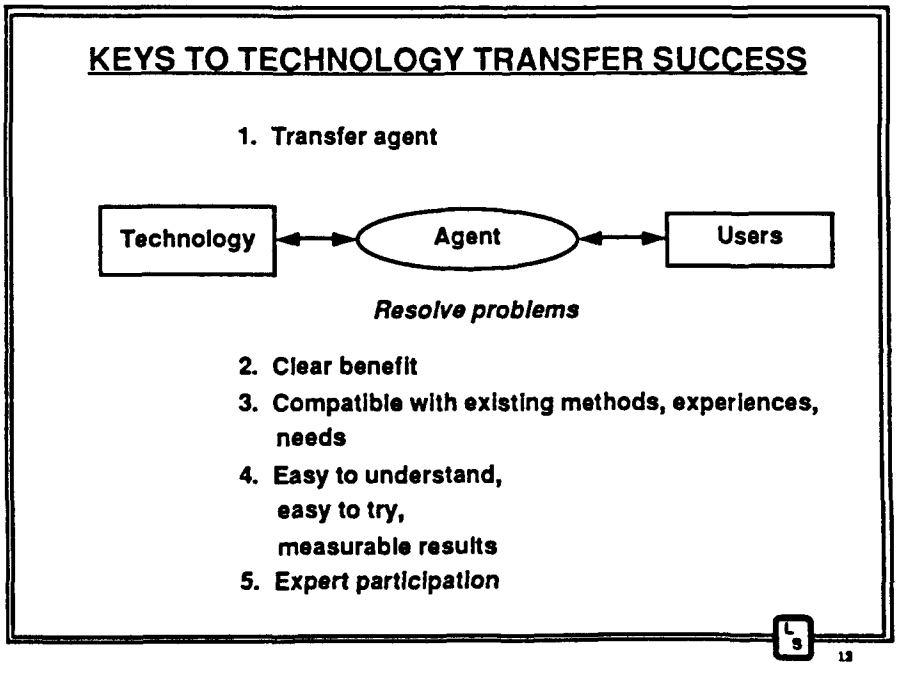
Software Performance Engineering Applications





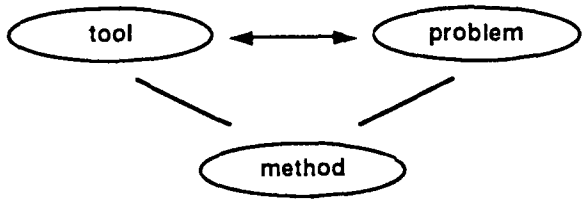
Software Performance Engineering Applications





Software Performance Engineering Applications

DISCOVERIES



Key problems:

perceived applicability of analytical models

lack of method

ease of use

You cannot address these independently

13

COORDINATED DEVELOPMENT

1. Tool must match the problem
2. Ease of use is vital to acceptance & use
3. Mission critical embedded computer system effectiveness is vital:

   Build the *right* system

   and

   Build the system right

14

Software Performance Engineering Applications

FUTURE RESEARCH ISSUES

1. Application-driven approach
   - methods + tools + applications
   - Orient to software engineers & their typical concerns
   - Define the method & specify POD's role
2. Topics:
   - Ease of use - examples
   - Extend performance models to address system effectiveness
   - Identify typical design & evaluation problems
   - Streamline model building and model verification & validation
   - Integrate with CASE environments
   - Create adaptive modeling capability

15

IMPROVE USABILITY

Sample model specification:

```
CONFIGURATION.SPECIFICATION
DEVICE CPUX TYPE = CPU
   RATE = 1 &MIPS
END
Z
MODULE .SPECIFICATION
MODULE FRAME-20HZ
   EST CPUX USAGE = 20
   CALL TIME CODE READ
   CALL GET 20 HZ INPUTS
   EST BUS USAGE = 5
Z
```

What is possible.....

16

Software Performance Engineering Applications

MODEL SYSTEM EFFECTIVENESS

- Streamline MC-ECS use:
  - Include user activities in the models
  - Comprehensive system models
  - Evaluate time to accomplish key activities

Example: F-18 screen displays

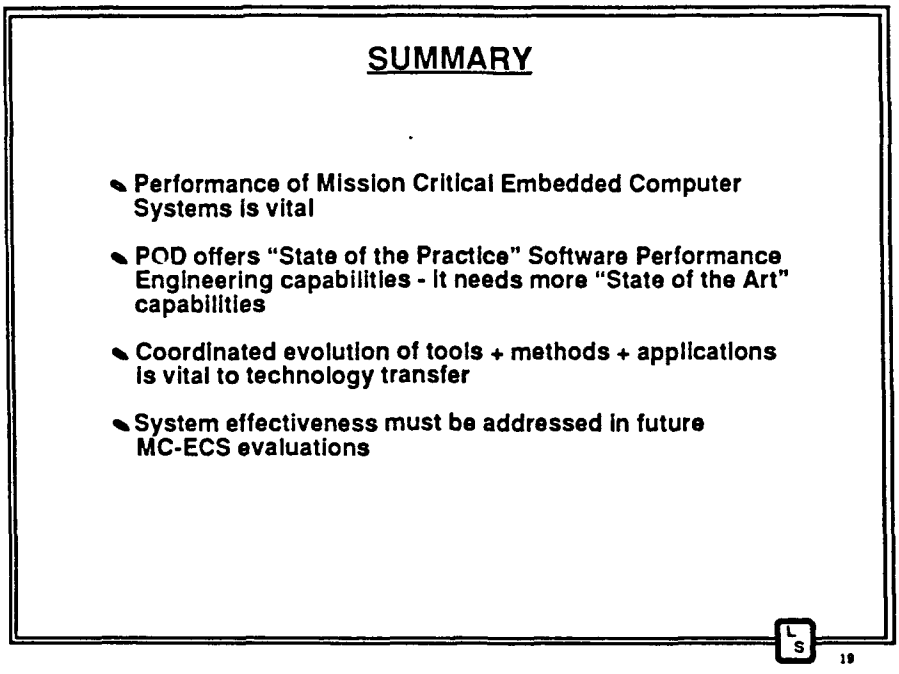
17

COORDINATE TOOLS, METHODS, APPLICATIONS

- Automate definition & evaluation of typical concerns
  - Example - algorithm choices
- Streamline model building and model verification & validation
  - Example - Match code - models - measurements
- Integrate with CASE environments
  - Example - Reuse design specifications for models
- Create adaptive model capability
  - Example - Multiple solution methods with unified model representation

18

Software Performance Engineering Applications



Software Performance Engineering Applications

COORDINATED DEVELOPMENT

1. Tool must match the problem
2. Ease of use is vital to acceptance & use
3. Mission critical embedded computer system effectiveness is vital:

   Build the *right* system

   and

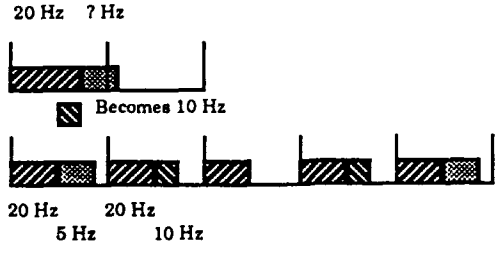
   Build the system right

PROPOSAL

- Continue case-study-driven applied research
  - Select another MC-ECS case study
  - Apply SPE and POD
  - Document applicability
  - Identify and *resolve* limitations in methods / tools
  - Transfer technology
- Anticipated enhancements
  - The following were typical F18 software performance questions and concerns. They are likely to occur in similar systems.

Software Performance Engineering Applications

DETERMINE PROCESSING FREQUENCY



NEED: Easier specifications
Visual results

15

EVALUATE BUS CONTENTION AND PROCESSOR INTERFERENCE

- Poll devices at beginning of cycle
- Channel 3 dedicated to handoff protocol between processors
- Data requirements and algorithms are interrelated

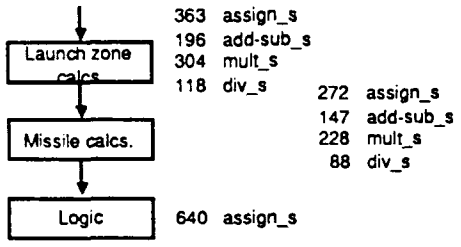
NEED: Case study execution details to construct models
Possible POD extensions

16

Software Performance Engineering Applications

EXPLICIT DATA REPRESENTATION

Current POD specifications: processing requirements



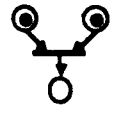
- Data structure alternatives affect # instructions executed
- Need to know data accesses to determine I/O requirements

NEED: Define data representation and analysis requriements
Tool enhancements

17

INTEGRATE ADDITIONAL PERFORMANCE SOLVERS

- Periodic job contention
- Ada rendevous
  Synchronization of concurrent processes
- Data latency
  Must respond to critical data values within window
  Data that is only valid for brief window
- Augment graph analysis and queueing theory with petri nets:

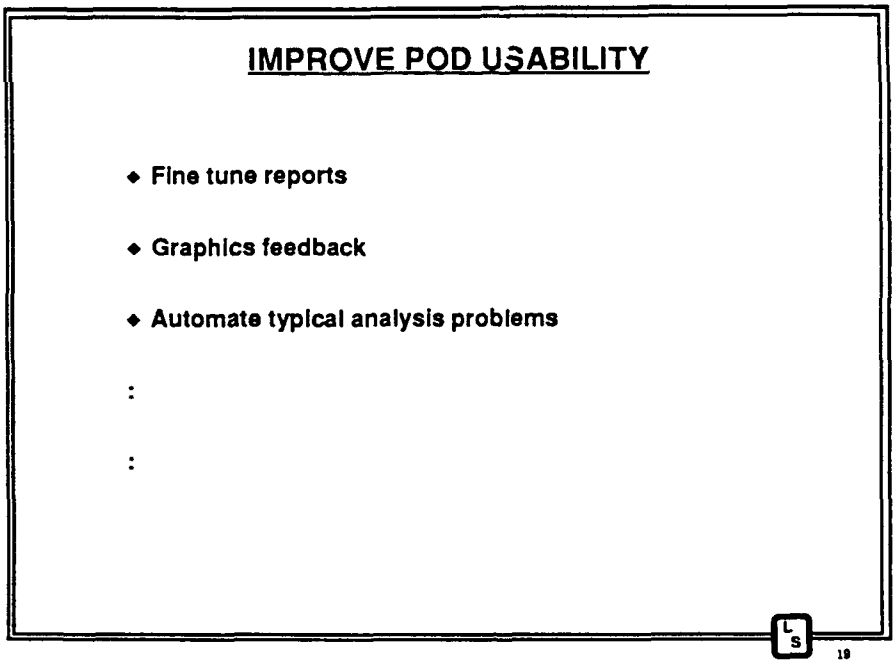


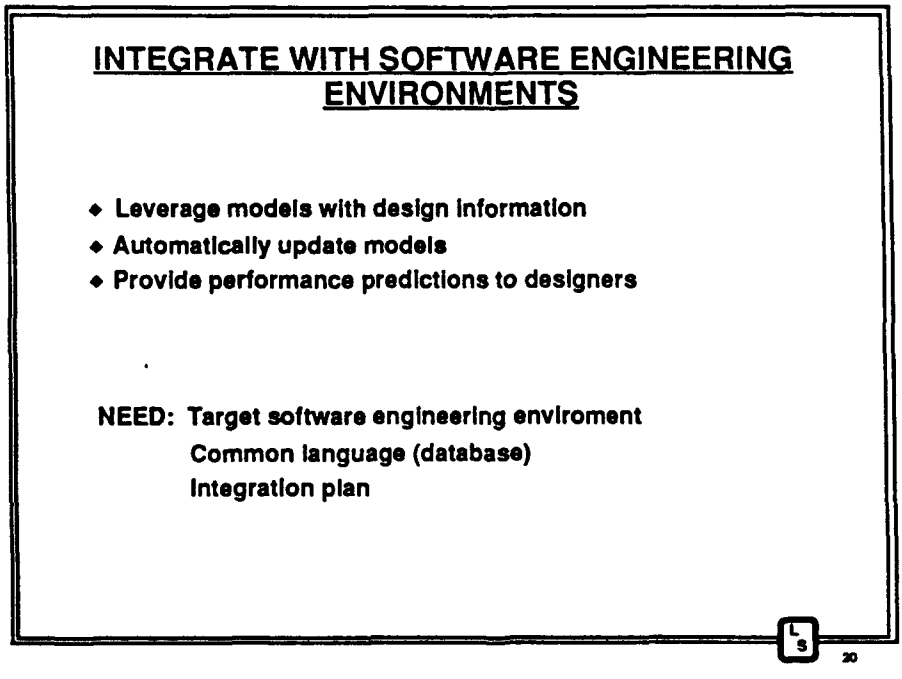
Places
Transitions
Tokens
Enabled transitions
Transition firing

NEED: Case study details
Prototype petri net interface to POD model definitions

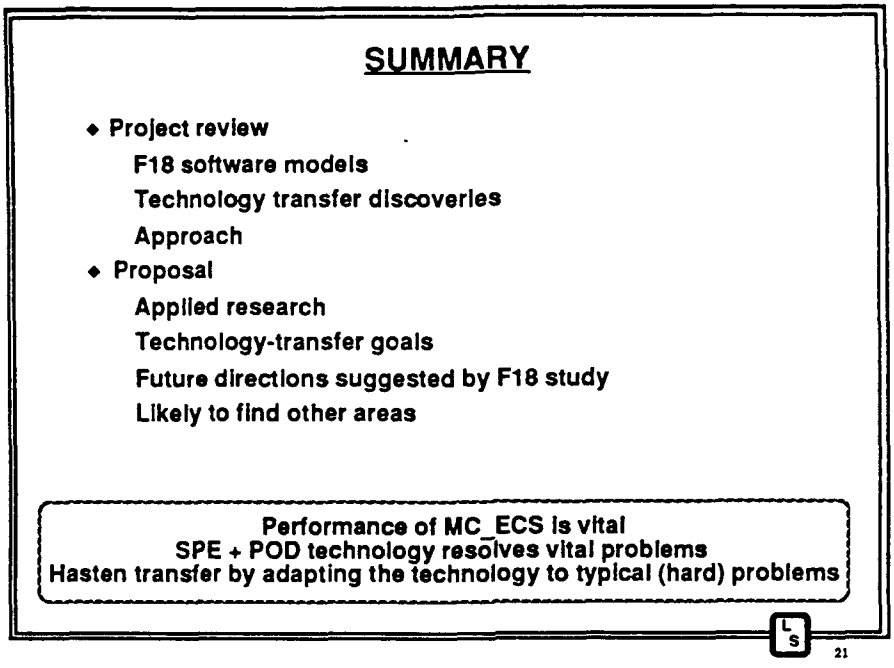
16

Software Performance Engineering Applications





Software Performance Engineering Applications



APPENDIX F

Technology Transfer Course Materials

APPENDIX G

Related Papers

41

Applying Synthesis Principles to Create Responsive Software Systems

CONNIE U. SMITH, SENIOR MEMBER, IEEE

Abstract—Performance engineering literature shows that it is important to build performance into systems beginning in early development stages when requirements and designs are formulated. This is accomplished, without adverse effects on implementation time or software maintainability, using the software performance engineering methodology, thus combining performance design and assessment. There is extensive literature about software performance prediction; this paper focuses on performance design. First, the general principles for formulating software requirements and designs that meet response time goals are reviewed. The principles are related to the system performance parameters that they improve, and thus their application may not be obvious to those whose speciality is system architecture and design. The purpose of this paper is to address the designer's perspective and illustrate how these principles apply to typical design problems. The examples illustrate requirements and design of: communication, user interfaces, information storage, retrieval and update, information hiding, and data availability. Strategies for effective use of the principles are described.

Index Terms—Design principles, software design optimization, software development method, software performance engineering, software performance models, software performance principles, software responsiveness.

I. Introduction

ENGINEERING new software systems is a process of iterative refinement. As illustrated in Fig. 1, each refinement step involves understanding the problem, creating the proposed solution, describing or representing it, and assessing its viability. The assessment includes evaluating its correctness, its feasibility, and its preferability (when there are alternatives). Many factors affect preferability, such as maintainability, responsiveness, reliability, usability, etc. This discussion focuses on only one, the *responsiveness* of the software; that is, the response time or throughput as seen by the users.[1] The understanding, creation, representation, and assessment steps are repeated until the proposed product of the refinement "passes" the assessment.

Responsiveness should be designed into the software, at the requirements and design levels of abstraction, when the number of alternatives is greatest and global optimizations are easily made. Then the leverage is greater: performance can be as much as one or two orders of magnitude better than for software that is first constructed then "tuned" to improve performance [4], [30]. Furthermore, since improvements are made at a high level, prior to coding, responsiveness can be achieved without sacrificing understandability or maintainability. With software tools to support the assessment, responsiveness can be achieved with little or no additional development time and cost.

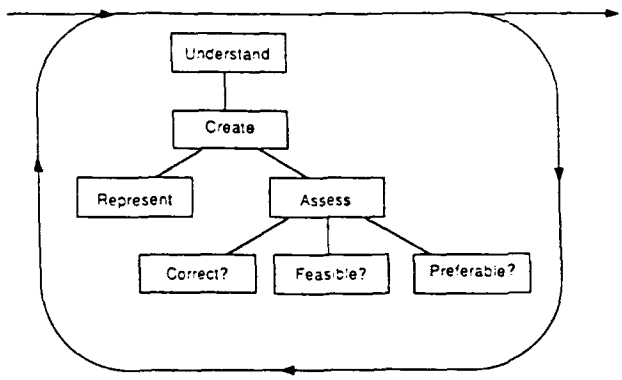


Fig. 1. Engineering design process.

The representation and assessment steps have been addressed elsewhere (references are cited later in this section). This paper addresses how to create systems likely to have acceptable performance, and how to revise them if assessment indicates that performance objectives will not be met. Some synthesis principles are described, and their application to software requirements and design creation is illustrated with many examples. The principles do not replace performance models (described in previous publications), but supplement them for engineering systems that meet responsiveness goals.

The remainder of this section reviews related work and contrasts it with this paper. Section II describes the principles and Section III illustrates applying them to communication issues; user interfaces; data organization for long term information storage, retrieval, and update; information hiding; and data availability (*when* data is created, sorted, retrieved, or converted). Section IV presents a strategy for using the principles, and Section V offers some conclusions. An Appendix summarizes the performance model basis for the principles, and the rationale for the set of principles.

Several software engineering methodologies advocate a

Manuscript received April 30, 1986; revised April 8, 1987.

The author is with the Performance Engineering Services Division, L&S Computer Technology, Inc., P.O. Box 9802, Mail Stop 120, Austin, TX 78766.

IEEE Log Number 8823080.

[1] For high performance systems, responsiveness can be a correctness requirement. If two alternatives both achieve the correctness goal (e.g., a response is produced within the specified time), the quantitative difference between the expected response times can be used to assess their preferability.

software design process similar to that in Fig. 1 [1], [2], [23], [37]. Booth [6], Sanguinetti [24], [25], Smith and Browne [28], [29], and others [10], [19], [33], [38] present performance prediction models applicable during early developmental stages. A software performance engineering methodology prescribes how performance assessment is integrated with traditional software engineering methodologies [30]. An extensive bibliography of performance engineering work is in [33]. The performance modeling and assessment are necessary for constructing responsive systems, but they are not sufficient. The synthesis principles are also needed to guide the creation step.

These synthesis principles apply to large *systems* of programs in *early* life cycle stages, when one is concerned with formulating requirements and design specifications that will lead to *systems* with acceptable responsiveness. Bentley [3], Ferrari [8], [9], and Knuth [11]-[13] and others [18], [37] have addressed program efficiency: creating efficient programs and "tuning" programs to improve efficiency. While many of the fundamental performance concepts are similar, large system design is different.

Lampson presents an excellent collection of hints for computer system design that address effectiveness, efficiency, and correctness [15]. His efficiency hints are the type of folklore that has until recently been only informally shared. The principles presented later formalize and extend the efficiency hints. Kopetz presented principles for constructing real-time process control systems [14]; some address performance. An earlier version of the general principles is in [34] and the performance analysis of three independent principles is in [35]. To experienced performance analysts, those hints and 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. They are also an effective way of communicating this knowledge to software architects and designers. This paper, therefore, explains the "expert knowledge" with an updated version of the principles, and illustrates applying them to software system requirements and design. Note that they *supplement* performance assessment rather than replace it.

Creating software requirements is included in this discussion even though the requirements are often prescribed and thus "set in concrete." In practice, however, the requirements may be negotiable, particularly when there are good performance reasons for doing so (and they can be presented quantitatively). Therefore, the examples presented in the following sections illustrate requirements as well as design alternatives.

II. Synthesis Principles

These principles have been developed through practical experience with large software systems. On one particular project, severe performance problems were detected during the system integration phase. A thorough performance "tuning" study was conducted that resulted in numerous proposals for improvements. Many were deemed infeasible because of the magnitude of the change at the late development stage. Others were implemented as proposed. Each of the proposed improvements was cataloged and classified by the type of change. The classifications were related to a performance model such as that in the Appendix, and the generalized set of principles was formulated on the basis of the relationship between the changes and the impact on the parameters of the performance models.

The principles apply to software systems, large or small, executing on a spectrum of computer systems: microcomputers, large mainframe computers, distributed systems, and MIMD computers.[2] They are most effective for very large software systems with high processing demands.

The following sections present the six general principles for the synthesis of responsive software systems: fixing, locality design, processing versus frequency tradeoff, shared resources, parallel processing, and centering. Each principle is defined and explained using simple examples. A hypothetical automated teller machine (ATM) example illustrates many of the principles. Examples are provided for both system requirements and design tradeoffs. Two of the principles, processing versus frequency tradeoff and centering, are described in detail in [35], so their description here is abbreviated. An earlier version of the fixing point principle was also previously defined; however, due to its revised statement and its importance in Section III, it is described in more detail.

A. Fixing Point Principle

Fixing is the mapping of an action or function desired to the instructions that accomplish the action. It is also the mapping of information required 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 action or information is required. Fixing could be done at several earlier points: earlier in the execution, at program initiation time, at compilation time, or external to the software system.

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

It is cost-effective to retain the fixing information when the cost of retaining it is less than the cost of fixing multiplied by the probability that refixing is unnecessary. Jobs or transactions will be more responsive when customized interfaces are designed with early fixing for common predictable actions and information, and when baselined versions of information are changed infrequently and are fixed early. For flexibility, special interfaces can be pro-

[2]When the processors have conventional von Neuman architectures. This excludes pipelined processors, systolic arrays, and other special purpose architectures. Such computer systems may have additional performance-determining factors not addressed here.

vided and used only when needed for uncommon actions, for information infrequently accessed, and for refixing to more recent versions as required.

An example of fixing requirements is choosing the interface for DBMS information selection. Runtime fixing is done when general ad hoc queries against the information are allowed, and the query is parsed and satisfied at runtime. Alternatively, the information can be fixed earlier, at compile time, with managed queries: predefining the information that is to be quickly accessible and building a menu with only those items. The data is still selected at run time, but the code to retrieve it is fixed at compile time.

Note that binding is a subset of fixing. An example of fixing that is not binding is determining when data in files or internal tables should be sorted. Files or tables kept in the desired order, with all additions preserving the order, are fixed early. Late fixing applies to ordering them when needed. (Some experts would consider this an example of "extended binding." Unfortunately, many subconsciously view binding in a limited context, so the term *fixing* is used to encourage broader interpretation.)

B. Locality Design Principle

Locality refers to the closeness of the mapping of logical tasks and resources to physical resources. 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 mapping then leads to four types of locality design for performance engineering. They are illustrated in the following example. Consider the logical task to sort a list of integers. *Temporal locality* of the mapping of this logical task to the physical resources is better if the integers are all mapped at the same time to the physical processor that sorts rather than one at a time (with a large interval of time between each). Similarly, *spatial locality* is better if the task is in a location that is near the physical resource, such as in memory that is directly accessible by (local to) the processor on which it will execute, rather than located on a disk drive attached to a different machine. The task could be mapped to different types of physical processors; consider the choice of mapping to a general purpose CPU or mapping to a special purpose chip designed specifically to sort lists of integers. *Effectual locality* is better for the special purpose processor since its purpose matches the task more closely than the general purpose CPU. *Degree locality* refers to the extent of the task, as in the length of the list of integers as compared to the size of the processors (e.g., speed, memory, size, etc.).

The locality design principle can thus be stated as follows:

> *Locality Design Principle:* Systems should be designed to have a close mapping of logical tasks and resources to physical resources.

Spatial locality is in menu networks that cluster related activities into a single menu and change menus when user activity changes. For temporal locality, those activities that are requested most frequently should be in the first menu seen by the user; infrequent activities should be seen much later. (Since they are clustered into the same menu, they also have good spatial locality; temporal locality refers to *when* they appear in the sequence of menus.) Effectual locality is found in most ATM's: they use a special purpose automated teller machine that has a user interface customized to the ATM application, rather than a general purpose terminal console. Degree locality matches the size of the ATM application to the machine. A microprocessor is sufficient, a supercomputer is not needed.

C. Processing Versus Frequency Tradeoff Principle

This principle addresses the amount of work done for each processing request, and its impact on the number of requests made. The "tradeoff" concerns the cases when more work per request reduces the number of requests made, and vice versa. The principle is as follows:

> *Processing Versus Frequency Tradeoff Principle:* Minimize the processing times frequency product.

A requirement tradeoff for the ATM example is determining whether multiple transactions per ATM session are to be allowed. The prompt for continuing a session requires some additional processing time, but the total number of sessions may be less than when a separate session is required for each transaction.

Another example is in displaying results of a database query when multiple data items are selected. Either all results can be displayed with a GETALL, or the first item can be displayed with a GETFIRST and each additional one with a GETNEXT. If users frequently wish to display all results, then the GETALL command may be desired for convenience. The principle can then be applied to the design. A design with a direct GETALL has a high bandwidth interface to the database; a design with an intermediate level of abstraction intercepts the user's GETALL request, issues multiple calls to the database (the GETFIRST and multiple GETNEXT's), accumulates the results, and transmits all back to the user at once.

There is not always a tradeoff in the two factors. For example, for file I/O one can retrieve one byte of information at the same relative cost as 20 bytes of information, because the processing time is dominated by the I/O operation. Therefore, if most of the time the additional information is eventually required, there will be essentially no change in processing time *and* the number of requests will be reduced. Thus there is no processing time penalty for reducing the number of requests.

D. Shared Resource Principle

Computer system resources are limited and must be shared. To share, either multiple processes can use the resource at the same time, or they can take turns, each process using the resource one at a time (multiplexing). The management of shared resources affects the software

in two ways: the additional processing overhead for scheduling the resource, and the possible additional time for waiting to gain access to the resource. The general principle is as follows:

> *Shared Resource Principle:* Resources should be shared when possible. When exclusive access is needed, the sum of the holding time and the scheduling time should be minimized.

By sharing resources (rather than taking turns), the overhead for scheduling is minimized and there is minimal wait to gain access (there may be a wait if another process already has exclusive access even though the requestor is willing to share).

Decreasing the sum of the holding time and the scheduling time of multiplexed resources decreases the *average* wait time to gain access to the resource.[3] There are four ways to minimize the holding time:

1) Minimize the processing time (using the other principles).
2) Hold only while needed.
3) Request smaller resource units.
4) Fragment the resource requests.

The first decreases the holding time by doing less work while the resource is held. The second says that a multiplexed resource should be requested just before it is used and released immediately afterwards. The third, requesting smaller resource units, means that less of the resource is held. The fourth, fragmenting requests, means partitioning one request that requires a long holding time into shorter requests, each of which requires a shorter holding time.

Minimizing the holding time of a multiplexed resource may increase scheduling time. The scheduling of service requests requires more processing for smaller resource units; and fragmenting requests into multiple shorter requests introduces more requests for scheduling service. *There will be a net improvement only when the additional processing is less than the expected wait time using the larger units or longer requests.*

E. Parallel Processing Principle

Processing time can sometimes be reduced by partitioning a process into multiple concurrent processes. The concurrency can either be real concurrency where the processes execute at the same time on different processors, or it can be apparent concurrency where 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 (e.g., the CPU, memory, or files), additional wait time may be introduced. Additional overhead processing may also be required to manage the communication and coordination between concurrent processes.

[3]The shared resource principle is a synergistic principle; it improves the average waiting time for all processes, rather than improving a process's own responsiveness. See the Appendix for further information.

The principle is as follows:

> *Parallel Processing Principle:* Parallel processing should be exploited (only) when the processing speedup offsets the communication overhead and the resource contention delays.

In general, the benefits derived through apparent concurrency are not significant compared to those achievable using the other principles. It has the further disadvantage of adding complexity to the software system. Real concurrency will be effective if the processing time reduction is much greater than the additional overhead for communication and coordination of the concurrent processes. The performance improvement must also be weighed against the cost of the additional processing power and the cost of more complex software, to determine whether it will be effective.

F. Centering Principle

The five previous principles provide guidance for the synthesis of software requirements and designs. Their application improves the performance of the part of the system to which they are applied. This principle is different in that it addresses leveraging the application of the principles by focusing attention on those portions of a large software system that have the greatest impact on its responsiveness.

The principle is as follows:

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

That is, create special execution paths for the dominant workload functions that are customized and trimmed to include only processing that *must* be done while the user waits. The principles in Sections II-A–II-E are applied to the special paths to minimize their processing. Separate transactions should be constructed for the workload functions that are requested less frequently.

Most automated teller machines apply the centering principle to system requirements by including a "quick withdrawal" transaction that reduces processing by eliminating prompts and processing for account type, amount, and additional transactions; and (on some ATM's) by omitting the new account balance from the receipt. They also apply it to the system design by customizing the access methods for the small percentage of customers who are likely to use ATM's.

The centering principle applies to all systems, but *what* one centers on may depend on the performance goal or the type of system. Thus far, the discussion has implicitly addressed application software systems that support online interactive users where the performance goal is responsiveness to users. When system throughput (number of responses per unit time) is a performance goal, or general purpose software systems do not have a dominant workload, centering is on those components with the largest cumulative space-time product across the specified usage scenarios. Note that *most* general purpose systems (e.g., database systems, operating systems, or other com-

mercial products) have dominant workloads. The developers may not know what they are, but they can be identified and it is important to do so. If one instead focuses only on the components with a large cumulative space-time product, improvements can be made by reducing their time, but other opportunities to shorten path lengths by eliminating general processing steps not applicable to the dominant workload may not be found.

Early in the life cycle, centering focuses on the functions frequently requested by users. Later, during implementation, the centering principle also addresses software components with large resource demands, the "major consumers," even though they may not be executed frequently. Because the major consumers hold resources, they may cause excessive delays to the dominant functions. While they can be identified earlier, they are typically less important than the dominant functions. They are addressed later, but do not drive the design earlier as do the frequent functions.

There is an additional difference in the early and late life cycle centering considerations: the effect of the improvements. Early life cycle centering focuses on the functions frequently requested by the users. We assume that reducing the processing time for these requests does not affect the number of times they are requested. Later in the life cycle we also address the major consumers of resources. When their processing time is reduced, the components that fall in the set of major consumers of resources may change. Thus, reducing the resource requirements of the major consumers could be an endless process, since there will always be major consumers of each resource. It is not endless because we focus on achieving the performance goal, not on minimizing resource usage.

G. Summary

In this section, the principles for synthesis of responsive software systems were introduced and illustrated with simple examples. Most of them involve tradeoffs. A quantitative analysis of the performance effect of the fixing point, processing versus frequency tradeoff, and centering principles is in [35]. While they can be evaluated with simple back-of-the-envelope calculations, the others require more sophisticated performance models. Since the performance engineering methodology in [30] incorporates the modeling activity into the software design process, it is easy to use the models to quantify the tradeoffs. Many performance analysts have the necessary modeling skills, and numerous support tools are available.

III. Applying the Principles

Since there is a close correspondence between the performance principles and the computer performance factors that they affect (see Appendix), they are likely to be intuitive to an experienced performance specialist, but less familiar to one who specializes in software system design. This section illustrates applying the performance principles to software system requirements and design. It illustrates that performance-oriented design does not preclude the use of good software engineering practices.

A. Communication in Software Systems

External communication is the sending and receiving of information between processes. The processes may be executing on the same processor or on different processors (e.g., multiprocessors, distributed systems, or MIMD machines). External communication also includes "system calls" for operating system services, such as a call to an I/O service routine. Internal communication is the sending and receiving of information within a process; passing parameters in procedure calls is a common form.

The overhead for communication is often ignored during the design and implementation of software systems. This is because it is a transparent operation (the actual communication is usually handled by the operating system), the communication is not part of the "real work"—it is merely a support activity, and the bandwidth of external communication lines is known to be fast. Therefore the time to transmit messages is perceived to be insignificant. Nevertheless, the overhead is *substantial* and must be considered during the design of software systems.

The parallel processing principle directly addresses external communication. It specifies that processes should execute in parallel only when the communication overhead (and resource competition) are offset by the speedup in processing. Models are usually needed to quantify the effect on performance metrics of greater and lesser degrees of parallelism [32]. The other principles address reducing the communication overhead (which may make parallel processing viable). They are discussed in the remainder of this section.

First, consider using the fixing point principle to determine communication requirements. For external communication between processors, the sending and receiving processors must be fixed. A dedicated communication line connecting the two processors is the earliest fixing. The latest fixing is a shared communication line with processors examining each message to determine if it is theirs.

Consider applying the fixing point principle to communication design. For external communication the sending and receiving processes must be fixed. With late fixing, messages go to a central "mailbox"[4] and receiving processes periodically check to see if any waiting messages are theirs. Early fixing sends messages directly to the receiving process as, for example, with remote procedure calls or system calls. Intermediate fixing sends messages to the private mailbox of the receiving process. For fixing the location of messages in a private mailbox to the receiving process, late fixing requires the receiver to call a system routine to get the message (from a location hidden to the receiver). Earlier fixing allows the receiver to read the message directly.

The fixing point principle can also determine how procedures or sections of code within a program receive messages. The latest fixing is a central driver routine exam-

[4]This refers to a generic mailbox: some unspecified location (such as primary memory, disk, etc.) serves as a holding area for messages that have been sent but not yet examined by the receiver.

ining each message and invoking the appropriate routine. Earlier fixing is setting a "switch" that routes subsequent messages directly to the routine expecting a series of input messages. The "switch," of course, must be closed when the routine completes the message series. An example is in text editors: a user enters an "input mode" and subsequent input is automatically added rather than checked to see if it is a command.

Locality design also applies to communication. Temporal locality is best when there is a minimal time lag between when a message is sent, and when the receiver gets the information. So temporal locality is better for communication via a remote procedure call than via a mailbox when the receiver must keep checking for message arrivals.

Spatial locality applies to the nearness of the communicating processes. The locality is better when they are on the same processor than when they are on geographically separated processors. One can also view spatial locality as the access time to the message. With this interpretation, messages in shared memory have better spatial locality than messages in a mailbox that resides on a disk.

Effectual locality applies to the mapping between the logical communication mechanism chosen and its physical environmental support. For example, some computer architectures support rapid context switching. On these systems, communication via procedure calls has better effectual locality than communication via message passing. Some operating systems are message-based, so the effectual locality of message passing on them is better than for other mechanisms.

Degree locality is a close mapping between the amount of information sent, the amount that is essential to the receiving process, and the bandwidth of the communication channel. In a packet switching network, for example, degree locality is best if the amount of information transmitted (and needed) is equal to the packet size.

Processing versus frequency tradeoff also applies to communication. Communication time is often a large part of the total time, so it should be included in the processing time when evaluating the frequency times processing tradeoff. Communication requires overhead processing. The principle also suggests combining messages into fewer, longer messages rather than transmitting many shorter ones to reduce the frequency that overhead processing is needed.[5]

The previous examples illustrate the independent aspects of the communication problem: reducing communication overhead to improve one's own performance. The shared resource principle addresses the synergistic aspects of communication. When communication channels are shared, the potential wait-time (to gain access) and the scheduling time (the communication overhead) determine the best holding time (the message size and the transmission frequency). A performance model will likely be required to solve for holding time, especially if a large amount of external communication is inherent in the software system being developed.

Centering implies that the communication overhead for the dominant workload functions should be minimized. Thus, the fixing point, locality design, and frequency times processing tradeoff principles should be applied to the dominant workload functions to minimize their communication overhead. For large software systems with strict performance goals, the other principles should also be applied to workload elements that are major consumers of communication resources, because they affect the responsiveness of the overall system.

B. User Interface Design

The structure and strategy for *acquiring* information and for *viewing* it is the user interface of a system. Thus, for an interactive computer system, it includes the screen layouts and the interaction scenarios. For batch systems it includes the report formats and the input media formats. Batch systems are not specifically addressed here, but the general principles apply to them as well.

The user interface design affects both the amount of data transferred and the number of interactions between the computer system and the user's terminal (or device). The time required for the interactions can dominate the total time required to process the user's transactions; careful design of the interface can substantially reduce this time.

First consider applying the fixing point principle. A user interface with menu selection screens and data entry panels employs earlier fixing than one with free format commands and keywords that must be interpreted at runtime. Earlier fixing uses function keys on a terminal, or buttons on a device such as an ATM, to select frequent activities. Buttons on a mouse or a puck can be fixed early either to frequent commands or to picture elements that are frequently used. In the graphics example in Fig. 2, one mouse button is dedicated to the frequent function "change the current picture element to the next in sequence." In this example, one of the elements is typically used more than 90 percent of the time, so it automatically becomes the current picture element every time a placement is made. Thus 90 percent of the time only the inner loop in Fig. 2(b) is executed.

Another application of fixing is the design of the menu hierarchy or network. In a hypothetical menu hierarchy, a user who wishes to create a new picture first sees the main menu, selects "edit model," then sees the corresponding Level 2 menu, selects "add," then sees a third level menu and selects the first picture element to be added. An alternative, in Fig. 3, places the most frequent actions from the leaves of the previous menu tree (such as the picture elements to be added) in dedicated areas on the screen. Because they are selected directly with a pointing device, such as a mouse, fixing instructions to actions is earlier. On devices with more limited screen area, small icons can represent the most frequent actions, while actions desired less frequently can be grouped to-

[5]This example is not quite so simple. The communication overhead is reduced with this strategy; however, the contention for communication lines must be considered as well as the effective bandwidth of the communication channel to determine the net effect on response time. That is, the shared resource principle must also be considered.

```
Draw_topology;                    Draw_topology;
.                                 .
.                                 .
.                                 .
REPEAT                            REPEAT
  Point_to_pic_element;             Point_to_pic_element;
  Point_to_location;                REPEAT
UNTIL finished;                       Point_to_location;
.                                   UNTIL new_element;
.                                 UNTIL finished;
.
```

(a) (b)

Fig. 2. Fixing frequent actions and picture elements. (a) Late fixing: picture element selected for each placement. (b) Earlier fixing: current picture element fixed.

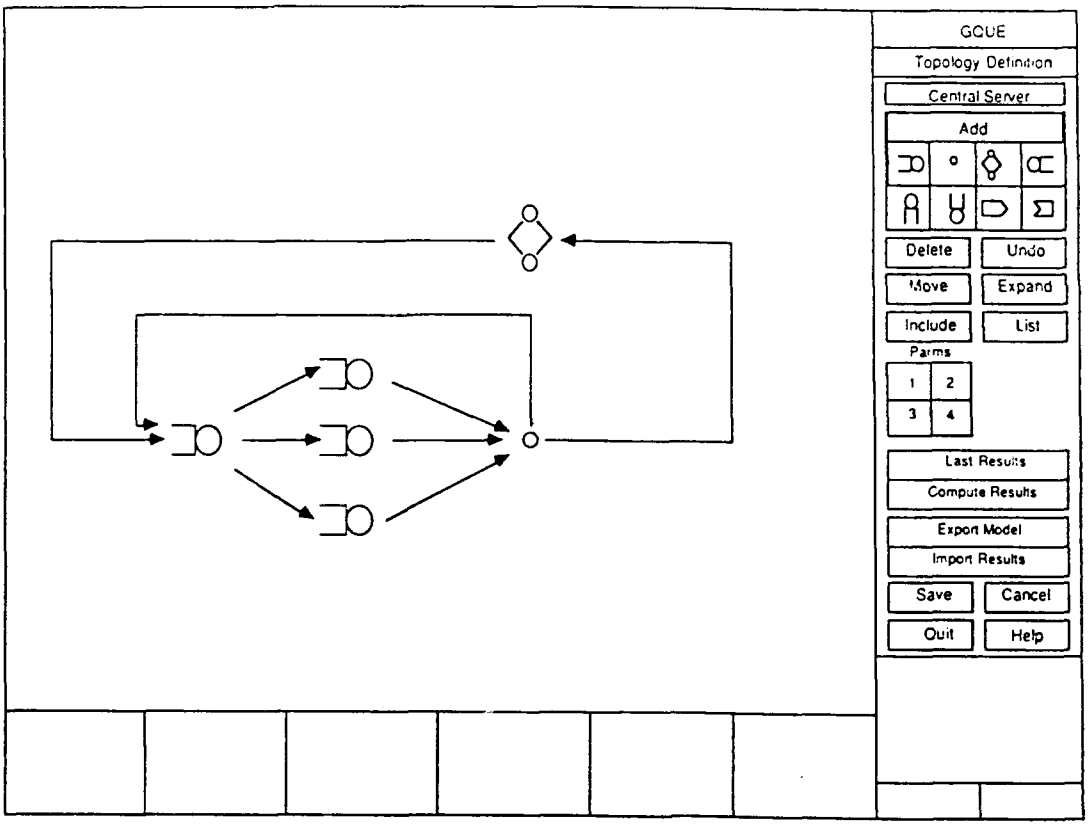


Fig. 3. Menu locality.

gether and selected via pop-up menus since later fixing is acceptable. For systems without a pointing device, the choices shown on the screen can be selected with function keys.

A related example is fixing the coordinates returned when the mouse button is pushed to the desired menu function or picture element. For example, any pair of coordinates within the "Add" box boundary in Fig. 3 should be fixed to the instructions that do the addition. Late fixing checks the coordinates of the cursor location against the coordinates of the box boundaries after the button is pushed. Earlier fixing uses a "gravity" feature. Centers of gravity are predefined and when the mouse is moved the cursor virtually "jumps" from the current center of gravity to the nearest one in the direction of the cursor movement (it may or may not actually "jump" on the screen). Gravity is often used with a grid on the screen to snap the cursor to grid intersection points. With gravity, the coordinates of the center of gravity are returned when the button is pushed making the comparison of coordinates (to fix the action to the instructions) much faster. Note that earlier fixing will not always be cost-effective because of the cost of retaining the information. For early fixing, it applies when one must frequently fix a region on the screen to an action (or to an object in a picture), and when the cost of retaining the fixing information is low, as when the cursor tracking and gravity calculation can be processed in parallel on the graphics device processor rather than on a central processor.

The fixing point principle also provides guidance for the placement of information on the screen and the length of time it should be retained. Information that is stable should be placed in a location where later retransmission can be avoided. Other types of reference information that are occasionally needed, and then only for a short period of time, should be placed in an area on the screen such

that it does not displace more stable information (that would later need to be redisplayed and thus retransmitted). On an intelligent display device, a pop-up window can be used and the displaced information maintained in a device buffer so that it can be redisplayed without retransmission.[6]

For screen organization, temporal and spatial locality design are interrelated: data that is needed within a short period of time should be closely located on the screen, and data not needed at the same time should not be mixed When the amount of data to be viewed is much greater than the screen capacity, and it is likely that all the data is needed, locality is better if all the data can be transmitted to the interactive device, and all the viewing manipulation commands (paging through data, locating specific information, printing hardcopies, etc.) can be processed on it without intervention from the central processor. This applies when most data is needed. If it is more likely that only a small amount is needed, the software should be structured so that only the needed information is transferred. Later, more could be separately requested. The information display is thus structured hierarchically based on frequency of use.

Effectual locality addresses the match between display device capabilities and the software requirements and design. Several capabilities already mentioned are supported by bit-mapped graphics devices (so that only changed data need be retransmitted) and device intelligence (for buffering data and manipulating it without intervention from the central processor). Windowing capabilities also offer opportunities for viewing and manipulating information concurrently. High resolution screens offer opportunities for matching the size of the information displayed to its value at that stage in the processing. For example, if information is being used only as a frame of reference, it need not be as large as the primary information being viewed or manipulated. Color displays offer opportunities for providing perceptual feedback to the user that may aid in problem solving. Thus, effectual locality design calls for using device intelligence, high resolution, color, and windowing when they can decrease the number of interactions with the user.

Degree locality is the nearness of the amount of data needed and the amount displayed or entered. Degree locality is better if default values are used and only nondefault data need be specified. The graphics example in Fig. 3 illustrates. The default picture in the figure is created with one menu selection. It is frequently created then modified with a few screen interactions, thus reducing the total processing to create a new picture.

[6]Note that this example (and some later) and the communication issues in the previous section are similar. This is because we are viewing screen layouts and interaction scenarios as requirements and design issues, while the I/O's to and from the interactive devices are really communication with the device. Nevertheless, the purpose is to present software development issues and how the principles aid in the synthesis process. The principles apply to user interface issues irrespective of whether one views them as a communication or a user interface problem. It is because of the commonality in the applications that the set of six principles is a better formulation of the synthesis concepts.

The processing versus frequency tradeoff principle also applies to the user interface. An example of decreasing processing is a hierarchical "help" command: a response that presents only the information the user is most likely to need reduces processing. There may be an increase in frequency; occasionally the user needs more information, so the longer version of the help is also requested. Decreasing the number of inputs by increasing processing is illustrated with an "Include" command (to incorporate a predefined model). One can either remind the user of the names that can be included, or assume that she or he remembers the name. The appropriate choice depends on the application. Automatically displaying the list may reduce the number of user inputs (due to errors and to separately selecting the "List" command *and* the "Include").

An application of the shared resource principle is designing screens such that information derived from shared files is segregated from information that requires exclusive file access. It is appropriate when the shared information alone is useful, especially when there is a long delay to get the nonshared data. The amount of data requiring exclusive access is an information storage issue and is discussed in the next section. The shared resource principle also applies to concurrent interfaces via multiple windows on a screen. It determines the best screen organization to maximize sharing of the limited screen area.

The parallel processing principle also applies. Several examples have been mentioned that call for asynchronous processing on the interactive device (e.g., paging through large amounts of data). Allowing multiple processes to communicate with a single interactive device is another example that may become more important in the future.

As usual, the centering principle focuses attention on the frequent screens and interactions. It is important to minimize the number of interactions for them.

C. Information Storage, Retrieval, and Update

There are three primary considerations for information storage, retrieval, and update:

- The structure of the information: its aggregation into files, records, and data items, and the relationship between aggregates such as ordering, hierarchy, etc.
- The information content: its representation and format
- The location of the data: the physical location of the files, records within files, and the data items within the records

Thus, the data organization decisions are *what* the structure and contents should be, and *where* they should be located.

The fixing point principle applies to when and how often the decisions are made. In database management systems, fixing information requested to the actual data format and location at runtime is more expensive than fixing at compile time. Data items accessed by the dominant workload functions should be fixed at compile time. Therefore, the database should be structured so that those items are sta-

ble and not affected by other database changes. Then compile time fixing is viable, since information structure for the dominant workload functions changes infrequently.

A second example addresses the creation and fixing of temporary information. Consider creating a matrix that shows the fraction of transactions in each ATM region made by customers residing in each of the other regions. Each account address must be fixed to an ATM region. The earliest fixing is to include the ATM region in every account record. Alternatively, a mapping file could correlate account addresses to ATM regions. Assume that a mapping file is used because the interval between uses for the mapping information does not justify retaining it with the account information. The analysis program processing for each ATM transaction is to access the account information to get its geographic location, then access the mapping data to get the corresponding region. The latest fixing point stores mapping data in an external file that must be accessed for each transaction. Earlier fixing "preprocesses" the mapping data in the first phase of the program to create a data structure in virtual memory for fixing the transaction region. Temporary fixing also maps complex database structures into simpler files for more efficient processing.

Spatial locality applies to the location of data in a distributed database. The data should be closest to the location where it is most likely to be needed. If distributed data resides in a remote location, late fixing gets data from the remote location when it is referenced, whereas earlier fixing recognizes earlier (e.g., at run initiation time, or even the beginning of the day) that remote data is needed, and transports it before execution.

Locality design also applies: data items used together (temporal locality) should be clustered together (have good spatial locality). Often external files have an abstract or logical structure; that is, related information is clustered together, but the relationship is based on logical content rather than temporal references. One example is the hypothetical personnel database structure in Fig. 4. The abstract relationship appears to be reasonable: personal data, job history, payment history, time log data, and payroll data are conceptually distinct and are clustered accordingly. The locality is better than if it were all stored in one record. For printing checks, though, multiple clusters must be accessed (personal data, time log data, and payroll data). There are few scenarios that access only one cluster with this structure.

Effectual locality design addresses the closeness of the mapping of the logical to the physical database design. It also applies to the design of internal data structures. For example, a binary search into an ordered data structure is usually best for random retrievals from a large table. However, if the table does not fit entirely in real memory, the binary search probe into the table may result in page faults making the average access time per probe greater. Thus, effectual locality addresses the total time to locate and retrieve the desired data item: the number of probes

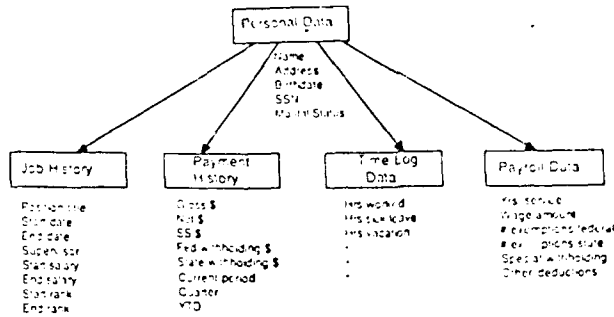


Fig. 4. Database structure with little temporal locality.

into the table and the time per probe (the instructions to make the probe plus page fault processing).

Degree locality matches the size of the data structures and the storage medium. While small amounts of data are often manipulated by programs, storage devices process large amounts more efficiently. Buffering, blocking, and using direct access storage devices with cache memories improve the closeness of the mapping.

The parallel processing principle applies primarily to distributed computing systems. A proper distributed data organization reduces the overhead for communication and synchronization of processes executing on separate processors, thus making parallel processing viable. An improper organization increases this overhead to the point where parallel processing is no longer effective.

Centering calls for selecting a data organization that minimizes the physical I/O operations for frequent requests (dominant workload functions). For external files and databases there many be conflicting workloads; random access may dominate during the day, but sequential access may dominate for overnight workloads. When the conflicting workloads run concurrently, models must be used to determine the best overall organization.

D. Information Hiding

Information hiding is the concept of hiding implementation details [20], [21]. It applies both to hiding data organizations and to the implementation of operations on the data. Parnas recommends applying information hiding to "design systems for change." With his method, aspects of systems that are likely to change become "secrets" that are hidden from the rest of the system. Aspects of systems that are unlikely to change are operations or "interfaces" known by the rest of the system. By hiding the secrets, the effect of changes to them is localized. Abstract data types (ADT's) and object oriented programming are methods of implementing information hiding [5], [17].

Information hiding has the advantage that software is less dependent on the format, location, and current operations than might otherwise be the case. A disadvantage is that, without careful implementation, it can be inefficient: the overhead of procedure calls for the interfaces may be excessive; the granularity of the data items may be too small; or the locality may be suboptimal. If these inefficiencies adversely affect the dominant workload

functions, the responsiveness of the system may be unacceptable. The inefficiencies are not inherent defects of information hiding, but may exist if a straightforward implementation is used.

It is vital to consider the performance of key interfaces. A fundamental assumption is that they are unlikely to change; they may be used throughout the rest of the system, so changes to them can propagate extensively. The internal data representation is easier to change later; thus, it is not a key consideration, but it is just as easy to do it right the first time.

Sections III-C and III-E address internal data representation issues; the remainder of this section addresses the interfaces. The important issues are identifying the interfaces that are key to responsiveness, early fixing of data to the interfaces, properly retaining the fixing information, and appropriately decomposing information.

The centering principle identifies the interfaces most frequently requested. Using the other principles, their efficiency is then optimized. This may lead to identifying new interfaces that are needed. For example, an "account" may hide the name, address, social security number, and balance, and have an interface for each (e.g., "get name," "get address," etc.). If a customer wishes to open another account, and the software is to use the information from the first account, three procedure calls are made to the interfaces. This function is not likely to be a dominant workload function; however, if it were frequent, an additional operation for "get all account information" or even "create new account" should be provided. Other special interfaces may be needed to get combinations of data items that are frequently needed together.

The fixing point principle specifies when the information is fixed, and how long it is retained. Consider the example in Fig. 5(a). Each interface performs an I/O to retrieve the desired data element. This is fine for random requests for data elements. But, if dominant workload functions are likely to request multiple data items for the same account,[7] earlier fixing reads all data items when the first is requested. Each interface then checks to see of the desired data item is in memory, or if an I/O is needed, as in Fig. 5(b).

The cost of retaining the fixing information is important when multiple processes call the interfaces. In the example, information for multiple accounts must be retained. One implementation is to retain data items for each user; each interface then checks to see if the desired data item is among those in memory, or if an I/O is needed, as in Fig. 5(c). Later fixing uses the same interface as in Fig. 5(a), but modifies the I/O routine as in Fig. 5(d). It first checks to see if the desired data is in one of its buffers in memory before it starts the physical I/O operation.

The fixing point principle also applies to fixing code to the interface desired. If every interface requires a procedure call, the overhead may be excessive. However, it may be possible to fix earlier, at compile time, by using a preprocessor that inserts the code in-line.

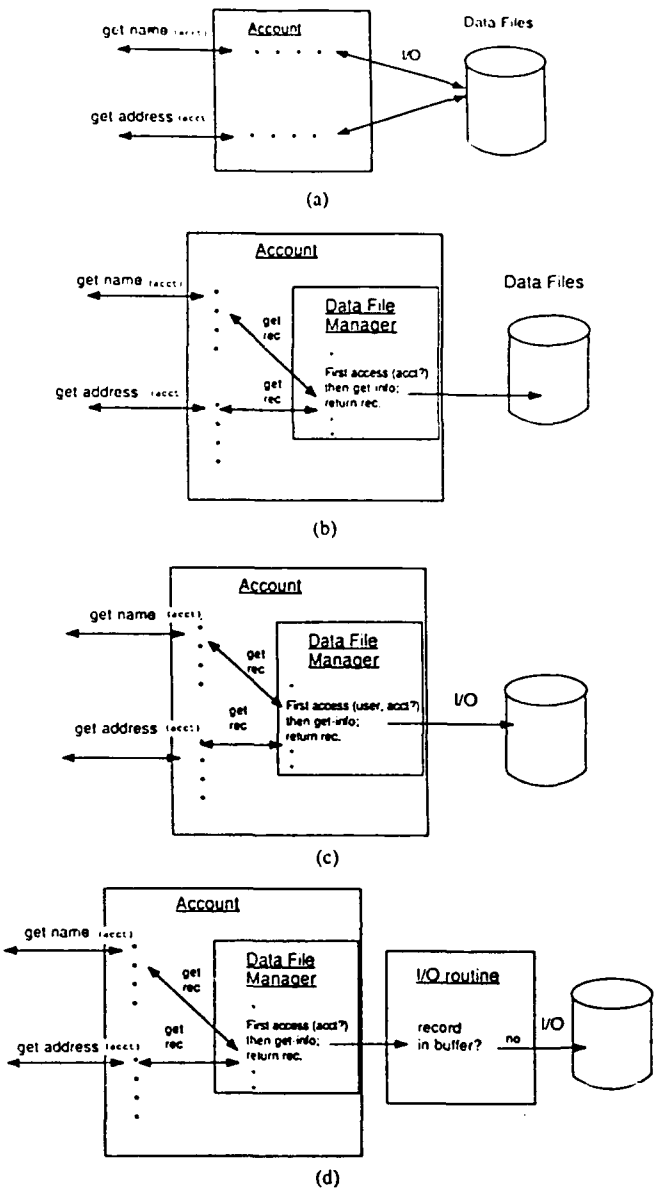


Fig. 5. Fixing abstract data type information. (a) Late fixing of data items: each operation performs an I/O. (b) Earlier fixing of data items: each operation calls the data file manager. It may buffer multiple records for efficiency. (c) Data items are fixed and explicitly retained for each user. (d) Later fixing: information is not explicitly retained for each user, but may still be present in I/O buffers.

Locality design also applies. There are two mappings to be considered: mapping the information (the logical resource) to the physical processor as before; and mapping the information to the external (user) domain through the interfaces. Effectual locality calls for both mappings to be close. Thus, there should be an abstract data type or object for each significant element in the user's environment. Ideally the mapping to the physical resources is also close, such as mapping account data items needed together to the same physical record.

Fig. 6(a) shows a hypothetical database scenario for retrieving logical records that are composed from multiple

[7]And if the account data elements are stored together—if not, the locality principle applies, as described later.

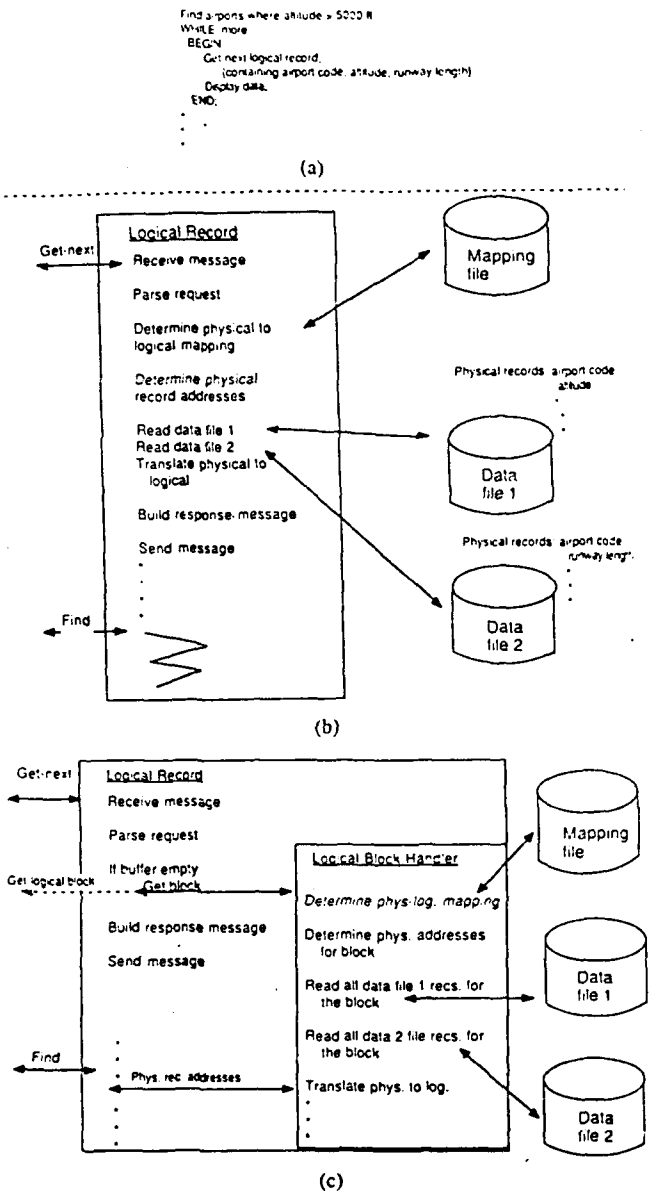


Fig. 6. Locality design for abstract data types. (a) Hypothetical scenario. (b) Logical records are composed for each access. (c) With nested logical block handler.

physical records. Fig. 6(b) shows a design: for every "get-next" request the physical to logical mapping is determined, the physical records are read, and the translation from physical to logical performed. Fig. 6(c) shows a hierarchical abstract data type with a logical block handler nested within the logical record abstract data type. The logical block handler determines the mapping once, reads all the physical records necessary to compose the block, then performs the translation once. The mapping information is only read and processed once per block rather than once per record. In the worst case, the number of physical data file I/O's is the same as in the previous case, but it may be possible to reduce the number of physical I/O's (depending on the data organization) by reading in larger physical blocks of data. The performance is even better if the mapping is established once when logical data processing begins. This example focuses on the mapping overhead, but other types of DBMS overhead may also be reduced (e.g., binding physical record addresses, error checking, buffer management, etc.).

Note that this is a special interface for sequentially retrieving logical records. Random retrievals would use a different strategy. If sequential retrievals are used by the dominant workload functions, it is important to include this customized interface; it substantially reduces processing time. If not originally included, it would be difficult to later add a high-performance, sequential retrieval interface.

The temporal and spatial locality between the user domain and the ADT is improved in this example by adding the "get logical block" interface to the logical record data type, as shown with the dashed line. It additionally saves on DBMS overhead for message passing and request parsing since it is only needed once per block instead of once per record.

The processing versus frequency tradeoff principle applies to the interfaces to be called and the amount of processing for each. The example in Fig. 6 can also be viewed as a processing versus frequency tradeoff. The logical block handler does more processing per call, but it is called less frequently.

The shared resources principle applies when objects can be accessed by multiple users. The holding time is minimized by applying the principles to both the data representation and to the interfaces as in this section. The "hold while needed" can be enforced through the interfaces. The fragmentation of requests can be similarly controlled by either combining requests into one interface, or fragmenting each into a separate interface. Locks can be a "secret," so the granularity of the locks can be varied from the entire database to an account as appropriate, without propagating the change throughout the system.

If the abstract data types may be mapped to different processors, the parallel processing principle applies. The locality design principle provides guidance for mapping them to physical processors. For example, ADT's should not be divided between processors unless the effectual locality is improved by mapping an interface to a processor designed specifically to handle that operation. ADT's are well suited to parallel processing since the interfaces are explicit and the processing is encapsulated. Thus it is easy to model the communication overhead and the processing time to evaluate the cost-effectiveness of various parallel processing strategies.

This section illustrates applying the principles to the synthesis of abstract data types and objects. Interested readers should also refer to related work by Booth and Wiecek on performance abstract data types [7]. They advocate extending the abstract data type definition to include performance specifications that facilitate the performance assessment.

E. Data Availability

Data availability addresses *when* data is available, that is, when it is created, stored, retrieved, or converted.

Choices are upon demand (i.e., when it is needed), or anticipatory (i.e., before it is needed). Data availability concepts are similar to the data organization and abstract data type interface concepts discussed in the previous two sections; they decomposed the data issues into organization and access strategy. This section integrates the two, using the principles, to ensure that *items used most frequently have minimal access delays*.

The centering principle is key to identifying and focusing on the data items that are used most frequently. Early fixing is the primary technique for achieving minimal access delay. With early fixing the data access is anticipatory. An example is the logical block handler in Fig. 6(c). It anticipates that many logical records will be requested, and composes (fixes) an entire block from the physical records before they are requested. The earlier example concerns fixing the mapping information once per block rather than once per record. Here the concern is reducing the average time to access each logical record. In the logical block handler the same strategy accomplishes both, but that is not always true (an example of a conflict is given at the end of this section).

Data availability address both *when* information is fixed and *how long* it is retained. The expected interval between requests to data items indicates the best strategy for retaining fixing information. For example, data items that have long periods of inactivity interspersed with occasional periods of high activity (e.g., 50 requests in a 5-minute interval, twice per day) should be retained in a location that has minimal access delay during the periods of high activity.

Another example is in computer-aided design (CAD) systems. There are two common strategies for organizing data used for CAD:

- To *integrate* the database and the analysis programs by having the programs read data directly from the database as it is needed, and to insert results directly into the database.
- To *interface* the database and analysis programs by first extracting the data, feeding all into the analysis program, and later taking all results and inserting them into the database in bulk.

The later is typical, since most of the analysis programs were written before the databases were created. Many CAD experts advocate the former, due to its flexibility and the overhead involved for the data extraction (preprocessor) and insertion (postprocessor).

Locality design determines the best of the two strategies for responsiveness. Effectual locality is best if the mapping of the data organization is close to the needs of the analysis program. The physical data organization within the database may not correspond closely. Even if it does, it does not have the best temporal and spatial locality since the analysis program must interface through the data management routines. If the database is used by many other CAD tools and designers, there is likely much more data than any single program needs, so the degree locality may not be close. Thus, data availability is best for the interface strategy. Furthermore, the performance of the analysis program can be optimized by using data structures that have a close mapping to the solution algorithms. Future analysis programs offer opportunities for improving the locality of integration since both database organization and algorithms can be better matched.

Shared resources also impact the access delay. It is minimized if processes share the resource. If exclusive access is needed, there is a conflict: the delay for each access is lower if the data is locked once and held until the data is no longer needed, because the code for locking and unlocking is executed fewer times. This strategy, though, may increase the time that competing jobs must wait to obtain access to the item. Performance models are necessary to resolve the conflict.

IV. Strategy for Effective Optimization

The general principles are necessary for developing systems with good performance characteristics, that is, for "doing it right the first time," but they are not sufficient. There are four factors that determine their effectiveness; the principles must be:

1) applied to appropriate software components
2) necessary
3) lead to global improvement
4) cost-effective.

Each of these is explained in the following paragraphs.

The principles must be applied to the components that *are critical to performance. Identifying them at design* time can be difficult; and intuition can be misleading. Some designers mistake the components that are most difficult to design and implement for the critical components; whereas, the critical components are generally those most frequently executed.

For example, network security management is difficult to design because many complex situations must be handled. Security may require a fair amount of execution time; however, other network software components, such as the communication protocol routines, are generally more frequent. Security management *may be* critical to performance, modeling determines its impact.

The second effectiveness factor is applying the optimization efforts to software components only when *necessary*. Usually, it is unnecessary to overachieve a performance goal; therefore, if the performance goal can be easily attained, it is not effective to devote valuable development time to extensive performance enhancements. Similarly, it is not effective to expend much effort optimizing components of the software system that have little impact on overall performance.

Another aspect of necessary optimizations is distinguishing software requirements that are necessary from those that are both unnecessary and adversely affect responsiveness. Sometimes these "artificial" requirements are introduced at design time with the intent to improve performance. An example is (an extra) requirement for a component that produces a list of items: that the list be in sorted order. The order requirement may be artificial, the

intent being to reduce subsequent search time. Performance models show whether the sort improves overall responsiveness. A straightforward approach (eliminating the sort) may not degrade responsiveness, and will result in less code to maintain.

The third effectiveness factor says that optimization techniques should result in *global improvements* to the software performance. This is most important in detailed design and coding stages, when many people are involved in development. Optimizations made in one part of the software system must be consistent with those made in other areas. Performance models quantify the overall effects.

The last of the four factors is that the performance improvement must be *cost-effective*. The time to implement the optimizations must be weighed against expected savings. It may be extremely difficult to achieve a specified performance goal for a large software system. Achieving the goal may be possible only at great costs in personnel time and in elapsed time to implement the system. It is important to estimate the cost of achieving the performance goal and insure that it is justified. It is often possible to negotiate for more reasonable performance goals before expending excessive efforts to achieve unrealistic ones.

V. Summary and Conclusions

Six principles for the synthesis of responsive software systems were presented, and applications to software system requirements and design synthesis were illustrated.

Section III presented several examples that could be viewed as applications of more than one of the principles. The principles correspond directly to the system performance parameter that they affect (see Appendix for further discussion). The different views usually arise because one design improvement favorably affects multiple system performance parameters. Since the goal is to create responsive systems, it does not matter which of the views leads the designer to the desired product. Since more than one principle may apply, the probability increases that a responsive system can be created. There would be a problem if they were contradictory, but they are not because the principles explicitly address the tradeoffs, and models identify the best alternative.

Consider the relationship between the principles and the responsiveness and maintainability of the software system. When improvements are made early in the life cycle, they only affect the requirements or the design, thus no changes to code are required. Performance tuning projects conducted late may require time for numerous changes to program code, additional time for retesting, and the resulting code becomes more difficult to maintain. The performance is improved, but not as much as possible because many important improvements that can easily be incorporated early in the life cycle are infeasible later. Note that with the performance engineering methodology one can identify such improvements and evaluate their effect early in the life cycle, before code is produced [31].

It is also interesting to note the correlation between the response time of the system and its responsiveness to users in the more general sense. For example, the special ATM function for "quick withdrawals" mentioned in Section II-F not only produces a better average response time, but also provides a more "friendly" interface for the many customers who use it. Similarly, the recommendations in Section III-B improve both the responsiveness and the usability of the system. Thus, the principles can be applied to improve performance of systems without adversely affecting software usability, readability, maintainability, or other quality factors. They are compatible with good software engineering practices.

Appendix
Basis for the Principles

The computer system on which the software executes can be viewed as an abstract model as in Fig. 7. Fig. 7(a) shows several types of jobs (*A*, *B*, C, · · ·) arriving for service, possibly waiting in a queue for their turn for service, then leaving upon completion. Fig. 7(b) shows an expanded model that identifies computer system resources that each of the jobs may use while being served. It is well known that the performance of such a system depends on the following parameters of the model [16], [26]:

- the arrival rate of each type of job
- the computer system resource requirements of each type of job
- the contention delays that result from the interaction with other jobs in the system
- the scheduling policies used to determine which waiting job next obtains the needed computer system resource.

The six principles improve performance by favorably affecting the corresponding system performance parameter. The table below summarizes the correspondence:

System Performance Parameter	*Performance Principle*	*Type*
Type of jobs	Centering	Independent
Resource requirements	Fixing Point	Independent
	Locality Design	Independent
	Processing vs. Frequency Tradeoff	Independent
Job Interactions	Locality Design	Synergistic
	Shared Resources	Synergistic
	Parallel Processing	Synergistic
Scheduling	—	—

Centering focuses on the jobs that are key to the responsiveness of the system. The fixing point principle concentrates on when and how often processing occurs. Locality design pertains to effective use of resources. Processing versus frequency tradeoff affects the number of requests for resources and the amount requested. Shared resources influence the number of jobs available to use a resource in a time interval. The extent of parallelism and the competition for resources among the parallel tasks is affected by the parallel processing principle. The scheduling of jobs is not addressed because it is generally a

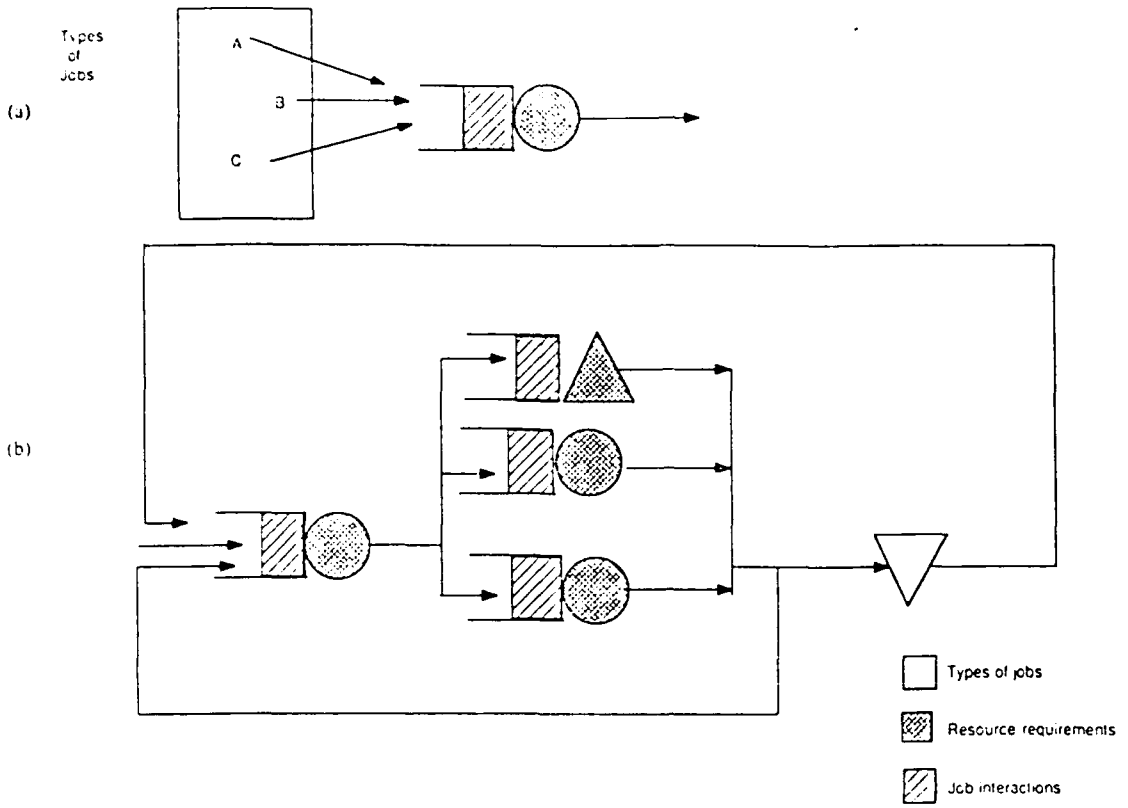


Fig. 7. Computer system model. (a) Abstract system model. (b) Expanded model.

service provided by the computer system, and not frequently a decision that must be made in the early development stages of new software systems. If scheduling is addressed, it is usually only after implementation when job priorities are manipulated.

There are two types of principles shown in the table, independent and synergistic. Independent principles improve the responsiveness of the "job" to which they are applied by improving its own performance parameters (e.g., reducing its resource requirements). Thus the improvement is *independent* of the characteristics of other types of jobs. The synergistic principles, on the other hand, improve the overall responsiveness through *cooperation:* they can reduce the average time waiting for resources if the competing jobs abide by the recommended principle. The locality design principle is both independent and synergistic, because it can improve a job's own responsiveness as well as benefit competing jobs.

Most of the principles require a tradeoff decision. The performance of the various alternatives may not be obvious, particularly for the synergistic principles, since it is affected by many interrelated factors. A previous paper provided quantitative improvement formulas for the independent principles [35]. Performance models similar to Fig. 7 can aid in the decision making for the synergistic principles [28], [29], [36]. Because of the interaction, though, the definition of some of the principles may seem ambiguous. In practice, the performance engineering models (similar to that in Fig. 7) resolve the ambiguities.

References

[1] M. Alford, "SREM at the age of eight: The distributed computing design system," *Computer*, vol. 18, no. 4, Apr. 1985.
[2] T. E. Bell, D. X. Bixler, and M. E. Dyer, "An extendable approach to computer-aided software requirements engineering," *IEEE Trans. Software Eng.*, vol. SE-3, no. 1, pp. 49-59, Jan. 1977.
[3] J. L. Bentley, *Writing Efficient Programs*. Englewood Cliffs, NJ: Prentice-Hall, 1982.
[4] —, "Programming pearls," *Commun. ACM*, vol. 27, no. 11, pp. 1087-1092, Nov. 1984.
[5] G. Booch, "Object-oriented design," in *Software Engineering with Ada*. Menlo Park, CA: Benjamin/Cummings, 1983.
[6] T. L. Booth, "Use of computation structure models to measure computation performance," in *Proc. Conf. Simulation Measurement and Modeling of Computer Systems*, Boulder, CO, Aug. 1979, pp. 183-188.
[7] T. L. Booth and C. A. Wiecek, "Performance abstract data types and a tool in software performance analysis and design," *IEEE Trans. Software Eng.*, vol. SE-6, no. 2, pp. 138-151, Mar. 1980.
[8] D. Ferrari, *Computer Systems Performance Evaluation*. Englewood Cliffs, NJ: Prentice-Hall, 1978.
[9] D. Ferrari, G. Serazzi, and A. Zeigner, *Measurement and Tuning of Computer Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1983.
[10] R. M. Graham, G. J. Clancy, and D. B. DeVaney, "A software design and evaluation system," *Commun. ACM*, vol. 16, no. 2, pp. 110-116, Feb. 1973.
[11] D. E. Knuth, *The Art of Computer Programming, Vol 1: Fundamental Algorithms*. Reading, MA: Addison-Wesley, 1968.
[12] —, "An empirical study of FORTRAN programs," *Software Practice and Experience*, vol. 1, no. 2, pp. 105-133, Apr. 1971.
[13] —, *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Reading, MA: Addison-Wesley, 1973.
[14] H. Kopetz, "Design principles for fault tolerant real time systems," in *Proc. Hawaii Int. Conf. System Sciences*, vol. 19, pp. 53-62, Jan. 1986.
[15] B. W. Lampson, "Hints for computer system design," *IEEE Software*, pp. 11-28, Feb. 1984.

[16] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative System Performance*. Englewood Cliffs, NJ: Prentice-Hall, 1984.

[17] L. A. Linden, "The use of abstract data types to simplify program modifications," in *Proc. Conf. Data: Abstraction, Definition and Structure* (*ACM SIGPLAN Notices*), vol. 11, 1976.

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

[19] L. J. Mekly and S. S. Yau, "Software design representation using abstract process networks," *IEEE Trans. Software Eng.*, vol. SE-6, no. 5, pp. 420-434, Sept. 1980.

[20] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Comman. ACM*, Dec. 1972.

[21] D. L. Parnas, P. C. Clements, and D. M. Weiss, "Enhancing reusability with information hiding," in *Proc. Workshop Reusability in Programming*, Sept. 1983, pp. 240-247.

[22] J. L. Peterson and A. Silberschatz, *Operating System Concepts*. Reading, MA: Addison-Wesley, 1983, pp. 91-129.

[23] W. E. Riddle, J. C. Wileden, J. H. Sayler, A. R. Segal, and A. M. Stavely, "Behavior modeling during software design," in *Proc. 3rd Int. Conf. Software Engineering*, IEEE Catalog No. 78CH13177C, May 1978.

[24] J. W. Sanguinetti, "A formal technique for analyzing the performance of complex systems," in *Proc. Computer Performance Evaluation Users Group 14*, Boston, MA, Oct. 1978, pp. 67-82.

[25] ——, "A technique for integrating simulation and system design," in *Proc. Conf. Simulation Measurement and Modeling of Computer Systems*, Boulder, CO, Aug. 1979, pp. 163-172.

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

[27] H. A. Sholl and T. L. Booth, "Software performance modeling using computation structures," *IEEE Trans. Software Eng.*, vol. 1, no. 4, Dec. 1975.

[28] C. U. Smith and J. C. Browne, "Aspects of software design analysis: Concurrency and blocking," in *Proc. Performance 80*, Toronto, Ont. Canada, May 1980, pp. 245-254.

[29] C. U. Smith, "The prediction and evaluation of the performance of software from extended design specifications," Ph.D. dissertation, Univ. Texas at Austin, University Microfilms Pub. KRA81-00963, 1980.

[30] ——, "Software performance engineering," in *Proc. Computer Measurement Group Int. Conf. XIII*, New Orleans, LA, Dec. 1981, pp. 5-14.

[31] C. U. Smith and J. C. Browne, "Performance engineering of software systems: A case study, in *Proc. AFIPS Nat. Computer Conf.*, Houston, TX, June 1982, pp. 217-224.

[32] C. U. Smith and D. D. Loendorf, "Performance analysis of software for an MIMD computer," in *Proc. Conf. Measurement and Modeling of Computer Systems*, Seattle, WA, Aug. 1982, pp. 151-162.

[33] C. U. Smith, "Performance engineering: A bibliography," (*Special Issue on Software Performance Engineering*), *Computer Measurement Group Trans.*, vol. 49, pp. 63-68, Sept. 1985.

[34] ——, "Synthesis principles for high performance software," in *Proc. Hawaii Int. Conf. System Science*, vol. 19, pp. 17-27, Jan. 1986.

[35] ——, "Independent general principles for constructing responsive software systems," *ACM Trans. Comput. Syst.*, vol. 4, no. 1, pp. 1-31, Feb. 1986.

[36] ——, *Performance Engineering of Software Systems*. Reading, MA: Addison-Wesley, to appear, 1989.

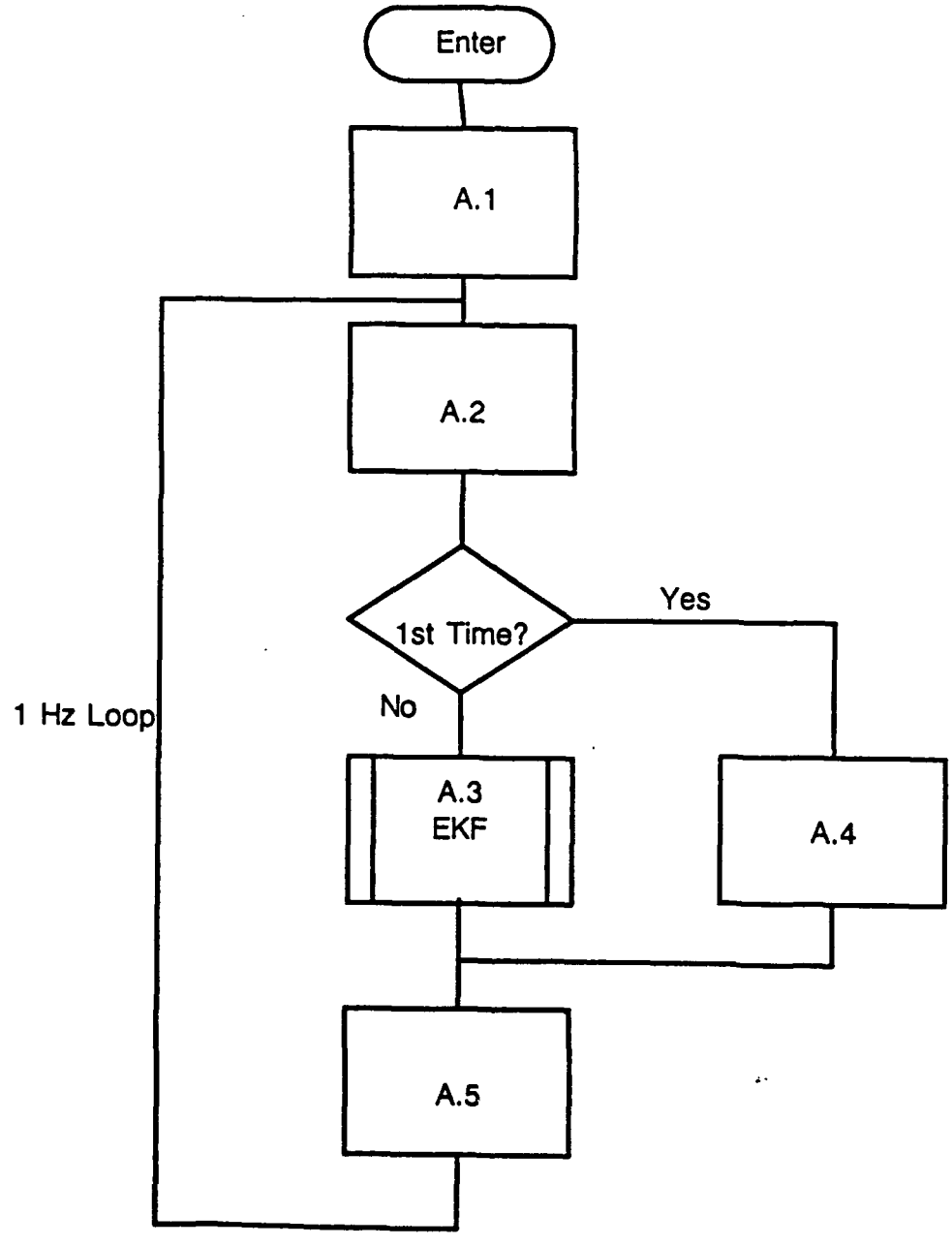
[37] D. Van Tassel, *Program Style, Design, Efficiency, Debugging, and Testing*. Englewood Cliffs, NJ: Prentice-Hall, 1978.

[38] J. W. Winchester and G. Estrin, "Methodology for computer based systems," *Proc. NCC*, vol. 51, pp. 369-379, 1982.

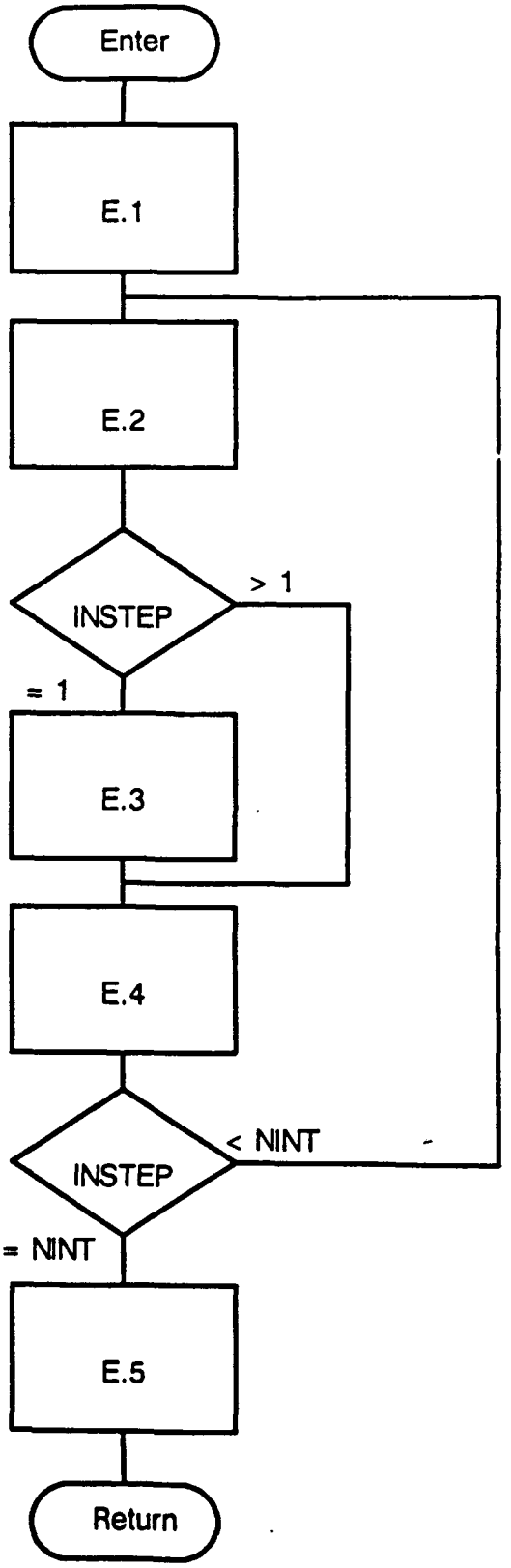
Connie U. Smith (S'79-M'80-SM'87) received the B.A. degree from the University of Colorado, Boulder, and the M.A. and Ph.D. degrees in computer science from the University of Texas at Austin.

She is currently a principal consultant with the Performance Engineering Services Division, L&S Computer Technology, Inc. Of her 19 years' experience in industry, government, academia, and consulting, 11 have been in the practice, research, and development of software performance prediction techniques. They were developed experimentally and applied to numerous large systems under development. Based on this experience, she proposed the "Software Performance Engineering" (SPE) methodology in 1981 that systematically assesses performance throughout software development. The synthesis principles are proposed to further advance SPE to a method for *preventing* problems as well as *detecting* them early. She has published numerous papers and articles on the subject and is currently preparing a book. Several state of the art graphical tools to support SPE have been developed under her direction. Her other research interests are performance modeling, software/hardware codesign, software engineering, design methods, graphical user interface design, and operating systems.

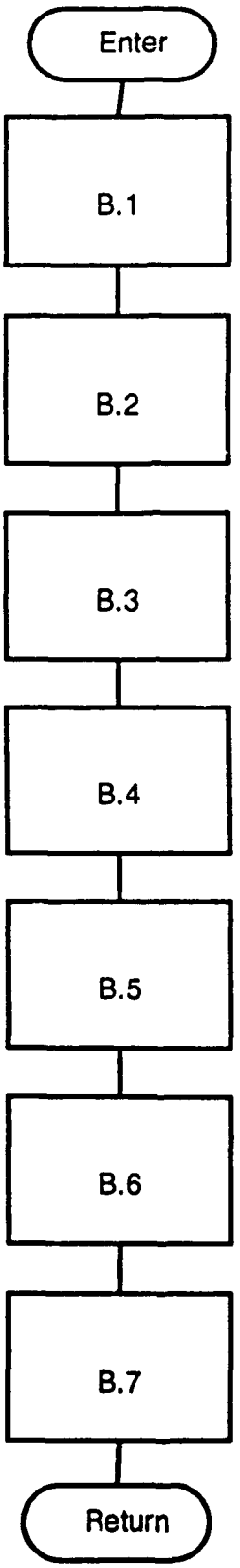
Dr. Smith is a member of the Association for Computing Machinery and the Computer Measurement Group (CMG). In 1986 she received the A. A. Michelson award for outstanding contributions to computer metrics for her work in Software Performance Engineering. She is a past ACM National Lecturer, Vice Chair of ACM Sigmetrics (1983-1987), and a director of CMG (1982-1986). She is the General Chair for the 1988 Sigmetrics Conference in Santa Fe, NM, and has served on many other conference and program committees.



Top Level – Algorithm #1



EKF
Subroutine
Algorithm #1



Block B.3 is where the 20Hz interface will reside.

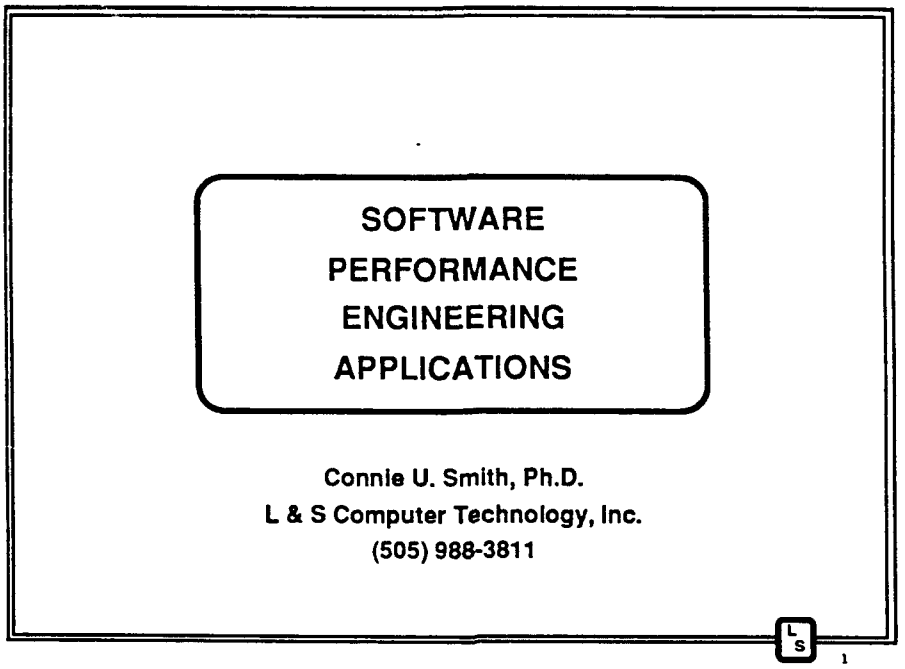
NOTE: There is no looping structure which I thought was in TWS.

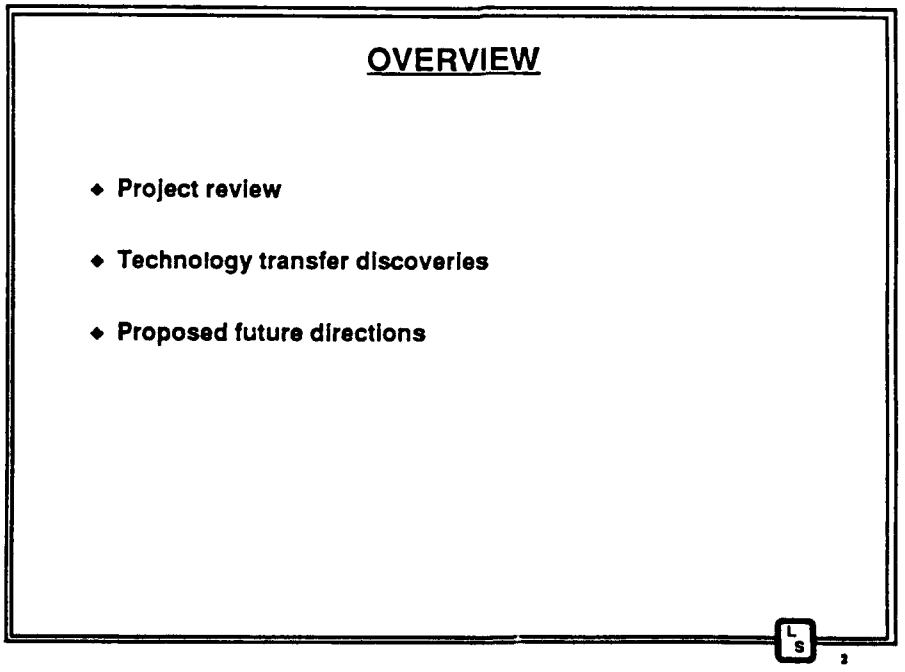
A/A Processing Flowchart

APPENDIX E

March 1989 Briefing

Software Performance Engineering Applications





Software Performance Engineering Applications

BACKGROUND

- Software Performance Engineering (SPE):

  Build performance into systems

- POD: *A Navy tool for predicting performance*
- Technology transfer project:

  Apply POD & SPE to MC-ECS -- F18 Software

  - Demonstrate results
  - Use results to promote technology transfer

3

MODELING REQUIREMENTS

- Low overhead
- Accommodate:
  - Incomplete definitions
  - Imprecise performance specs
  - Incremental development
  - Changes during development

GOAL:

Initially

- ➢ Distinguish between "good" and "bad"

Later

- ➢ Increase precision
- ➢ Provide implementation decision support

4

Software Performance Engineering Applications

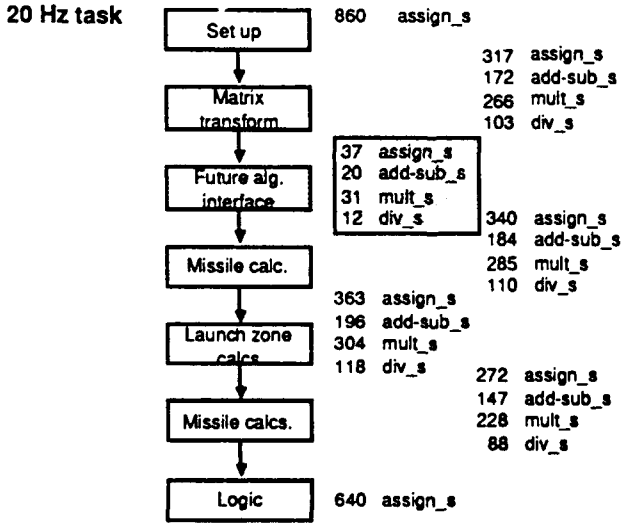
CASE STUDIES CONSIDERED

- Compare growth of software for 83X, 85X, and 87X releases on the AN/AYK-14 model XN-5
- Evaluate exchange of algorithm for ballistic trajectory
- Addition of "new feature" algorithm for computing target data
- Evaluate hardware upgrade: XN-5 to XN-6
- Concurrent processing: single vs. dual executive on XN-6

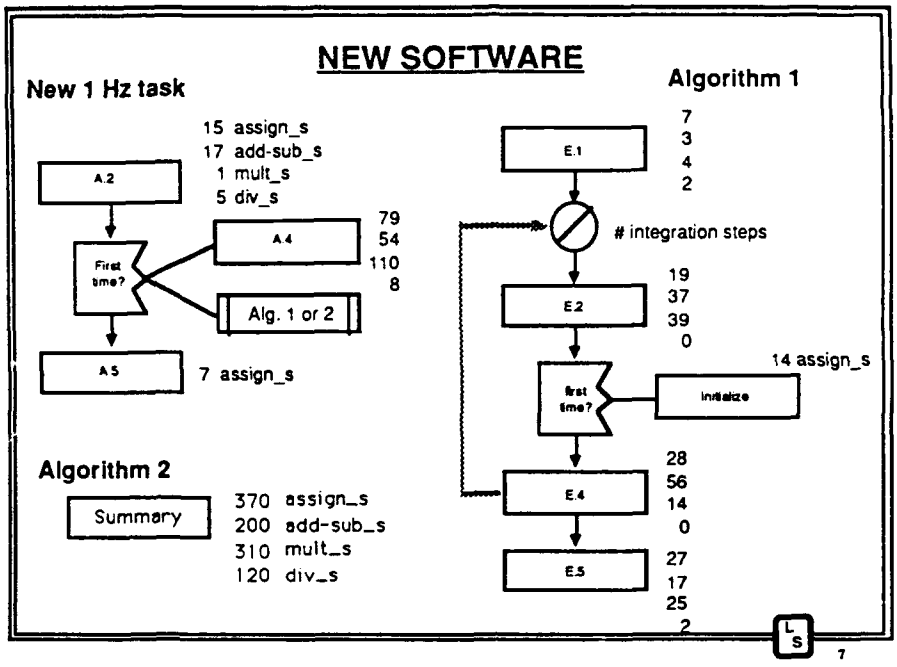
Others identified, but insufficient data for near-term case study

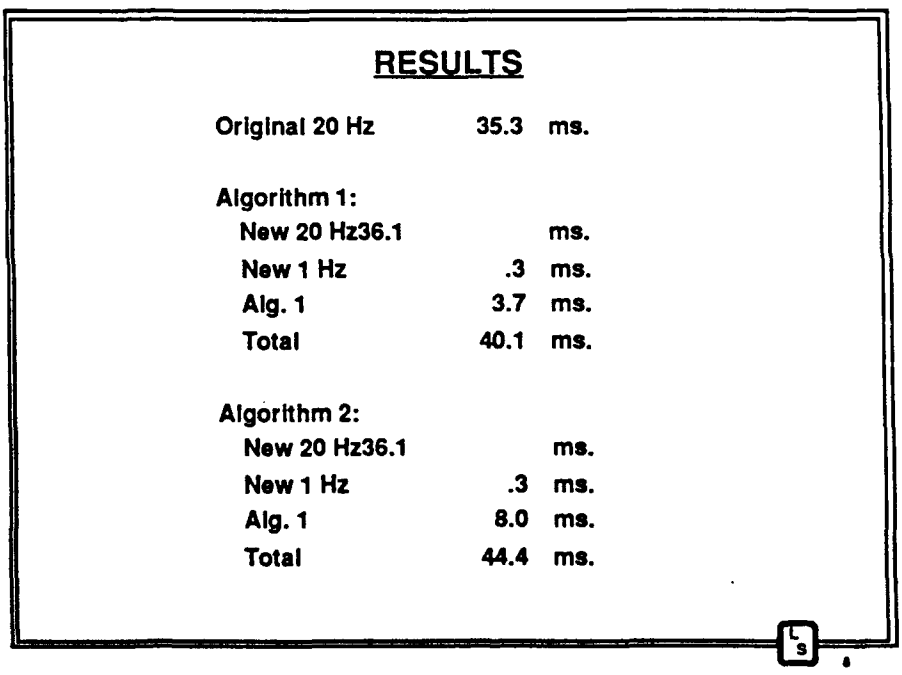
Selected algorithm addition because representative, of interest to designers, and sufficient data available

SOFTWARE MODELS



Software Performance Engineering Applications





Software Performance Engineering Applications

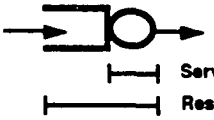
APPLICABILITY OF POD & SPE

- Found many software evaluations that could benefit from the technology
- Case study had no particular performance problems; the tool confirmed that there should be no surprises
- POD enables the rapid evaluation of many tradeoffs
  - Applying the algorithm to multiple targets
  - Varying the number of integration steps
  - Examining startup processing as well as steady state
  - Varying the hardware speed

9

TECHNOLOGY TRANSFER DISCOVERIES

- Usability limitations:
  - NOSC approach is *vital* to technology transfer:
    - Develop prototype into demonstration version
  - Make tool more appealing with visual interface
- Theory limitatic : Periodic arrivals in queueing theory?



Standard queueing models compute average congestion: R= (N+1) S

If periodic arrivals do not collide: R = S

Determining if collisions occur is nontrivial

If your only tool is a hammer, every problem looks like a nail

Effective technology transfer → Augment tools and SPE methods

10

Software Performance Engineering Applications

KEYS TO TECHNOLOGY TRANSFER SUCCESS

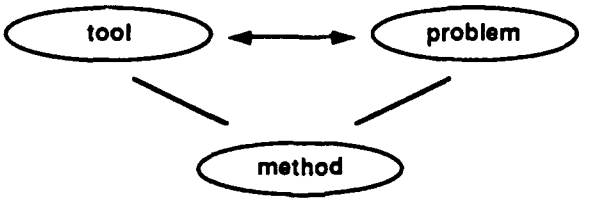
1. Transfer agent

Resolve problems

2. Clear benefit
3. Compatible with existing methods, experiences, needs
4. Easy to understand,
   easy to try,
   measurable results
5. Expert participation

11

DISCOVERIES



Key problems:

perceived applicability of analytical models

lack of method

ease of use

You cannot address these independently

13

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1 AGENCY USE ONLY *(Leave blank)*	2 REPORT DATE	3 REPORT TYPE AND DATES COVERED
	June 1990	Final

4 TITLE AND SUBTITLE
PERFORMANCE ENGINEERING FOR MISSION CRITICAL EMBEDDED COMPUTER SYSTEMS

5 FUNDING NUMBERS
C: N00039-86-C-0247

6 AUTHOR(S)

7 PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
L&S Computer Technology, Inc.
P.O. Box 9802, Dept. 120
Austin, TX 78766

8 PERFORMING ORGANIZATION REPORT NUMBER

9 SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Office of Naval Technology
Arlington, VA 22217

Naval Ocean Systems Center
San Diego, CA 92152-5000

10 SPONSORING/MONITORING AGENCY REPORT NUMBER
NOSC TD 1834

11 SUPPLEMENTARY NOTES

12a DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b DISTRIBUTION CODE

13 ABSTRACT *(Maximum 200 words)*

This document provides background information on performance engineering and the POD performance modeling tool, and gives an overview of the project activities. Finally, the project summary section reviews the results, lessons learned, and suggests future directions. A detailed review of the project activities is in Appendix A.

14 SUBJECT TERMS
mission critical, embedded computer system (MC-ECS)

15 NUMBER OF PAGES
62

16 PRICE CODE

17 SECURITY CLASSIFICATION OF REPORT	18 SECURITY CLASSIFICATION OF THIS PAGE	19 SECURITY CLASSIFICATION OF ABSTRACT	20 LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAME AS REPORT