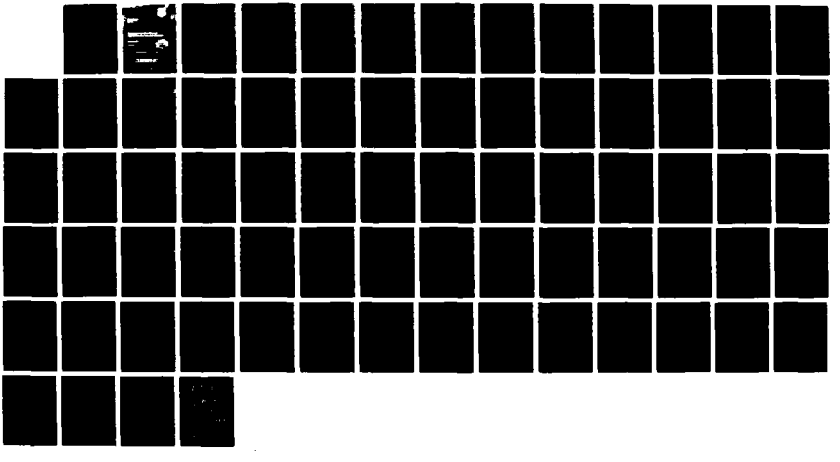
AD-A192 799 AN EVALUATION METHODOLOGY FOR DEPENDABLE MULTIPROCESSORS(U) SRI INTERNATIONAL MENLO PARK CA J GOLDBERG MAR 88 SRI-ESU-2918 RADC-TR-88-23 1/1

UNCLASSIFIED F19628-86-D-0003 F/G 12/6 NL



[redacted]

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

A192 799

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) ESU-2918	5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-88-23
6a. NAME OF PERFORMING ORGANIZATION SRI International 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (COTC)
6c. ADDRESS (City, State, and ZIP Code) 333 Ravenswood Avenue Menlo Park CA 94025-3493	7b. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Rome Air Development Center 8b. OFFICE SYMBOL (If applicable) COTC	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F19628-86-D-0003
8c. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. 63223C; PROJECT NO. B413; TASK NO 03; WORK UNIT ACCESSION NO. 40

11. TITLE (Include Security Classification)
AN EVALUATION METHODOLOGY FOR DEPENDABLE MULTIPROCESSORS

12. PERSONAL AUTHOR(S)
Jack Goldberg

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM Jul 87 TO Sep 87	March 1988	76

16. SUPPLEMENTARY NOTATION
N/A

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Computer Evaluation Methodology High Performance Fault Tolerance
09	02		

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report outlines an approach to a methodology for evaluating high performance, reliable computers. The purpose of the methodology is to provide a framework and a basis for tool development that will make it possible to conduct such evaluations systematically and efficiently. The increasing complexities of high performance computer systems and the stringent requirements for high reliability in harsh environments (e.g., space) make such an evaluation methodology an absolute necessity.

The report discusses sources of difficulty in evaluation, such as the many complexities of multiprocessing, the difficulty of distinguishing various factors (algorithms, software), operating systems, fault diagnostics, etc.) that affect performance and fault tolerance, the use of formal and experimental analyses, and the special problems of computer security. Criteria and suggestions are given for the design of unified working environments and specific classes of tools that support the methodology.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL David F. Trad	22b. TELEPHONE (Include Area Code) (315) 330-2925 22c. OFFICE SYMBOL RADC (COTC)

DD Form 1473, JUN 86 *Previous editions are obsolete.*

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Abstract

This report presents an approach to a methodology for evaluating high-performance, high-dependability computers for use in Battle Management/C3 (BM/C3) applications. The purpose of the methodology is to provide a conceptual framework and a basis for tool development that will make it possible to conduct such evaluations systematically and efficiently. There is an urgent need for such a methodology, because of the complexity of BM/C3 applications and the complexity of the computer systems that will be required for BM/C3 service.

A special characteristic of BM/C3 computation is that a great many requirements must be satisfied simultaneously: high performance, error tolerance and fault elimination, overload resistance, resource degradation, ease of program development, extendability, etc. In response to this characteristic, the methodology presented in this report is strongly based on the concept of *multiple evaluation domains* and on the extensive use of *models* to define all aspects of a system and its use.

The methodology calls for the construction of a set of evaluation domains that define all the aspects of the candidate system, its application, its operating environment, and perhaps its development environment, that are significant to the evaluator. Each domain is defined by a set of formal or near-formal models to a level of detail needed for the desired precision of evaluation, and the set of domain models, taken together, is employed in testing and analysis to provide systematic coverage of the evaluation space. Another feature of the methodology that addresses the need for combining evaluation criteria is the notion of using a single domain, called *Stress*, to include all the factors that may cause errors and failures. Standard modeling approaches are reviewed. The need is emphasized for new models of system behavior under stress.

A selected set of domains thus provides a comprehensive definition of an evaluation space, which can be used systematically to guide specific evaluation efforts. Because of the key role of models, the methodology is strongly concerned with model construction and validation.

The report discusses sources of difficulty in evaluation, such as the many complexities of multiprocessing, the difficulty of distinguishing various factors (algorithms, software, operating system, fault diagnostics, etc.) that affect performance and fault tolerance, the use of formal and experimental analysis, and the special problems of security. Criteria and suggestions are given for the design of unified working environments and specific classes of tools that support the methodology.

Contents

Chapter 1

Introduction

This report presents an approach to a methodology for evaluating high-performance, high-dependability computers for use in Battle Management/C3 (BM/C3) applications. The purpose of the methodology is to provide a conceptual framework and a basis for tool development that will make it possible to conduct such evaluations systematically and efficiently. There is an urgent need for such a methodology, because of the complexity of BM/C3 applications and the complexity of the computer systems that will be required for BM/C3 service. Many new computer architectures, incorporating high-order multiprocessing and fault tolerance, have been proposed and many new design proposals may be expected in the future. These designs are so different from conventional designs and from each other, the application is so complex, and the environment of operation is so harsh, that it becomes very difficult to evaluate the suitability of a candidate computer system for a specified service or to compare the merits of different designs.

Unfortunately, the state of the art in performance and dependability evaluation is not adequate to the task.[1] The use of standard benchmark programs, the most widely used evaluation method, has been discredited

[1] "There is, at this time, no commonly accepted methodology for the evaluation of the performance of supercomputer systems and no standard set of metrics for the representation of performance. More powerful evaluation methods are essential for comparing competing systems, allocating available machine resources, matching specific tasks to specific hardware, and designing new systems." *An Agenda for Improved Evaluation of Supercomputer Performance*, Committee on Supercomputer Performance and Development, National Research Council, National Academy Press, 1986, p. 1. Note that these remarks only address performance evaluation.

as an accurate indicator of performance because of the great sensitivity of performance to the match between algorithm characteristics and machine characteristics. Some evaluation tools exist, e.g., analytic, queueing and simulation models for performance, and marked-graph and Markov-based models for fault tolerance, but the existing ones have been developed independently (hence they are difficult to use in combination), and they cover only a few of the significant qualities of the new machines and applications. It is clear that many new tools are needed, but there is a danger that new tools will not be mutually supportive or consistent. A primary goal for a methodology of evaluation is thus to provide a framework for developing and integrating new evaluation tools.

A special characteristic of BM/C3 computation is that a great many requirements must be satisfied simultaneously: high-performance, error tolerance and fault elimination, overload resistance, resource degradation, ease of program development, extendability, etc. In response to this characteristic, the methodology presented in this report is strongly based on the concept of ***multiple evaluation domains*** and on the extensive use of ***domain models*** to define all aspects of a system and its use.

The methodology calls for the construction of a set of distinct evaluation domains that define the various aspects of the candidate system, its application, its operating environment, and perhaps its development environment, that are significant to the evaluator. Each domain is defined by a set of formal or near-formal models to a level of detail needed for the desired precision of evaluation, and the set of domain models, taken together, is employed in testing and analysis to provide systematic coverage of the evaluation space. Ideally, all domains of interest would be applied simultaneously in an evaluation; in practice, the evaluator may choose to apply many domains simultaneously but with coarse detail, or only a subset of domains at a time, with fine detail. In some cases, a domain may be constructed that unifies several evaluation criteria. One example suggested is the domain of Stress, which includes all the factors that lead to system error and failure.

A major objective of system evaluation work is to develop a ***model of the candidate system*** that allows the evaluator to estimate the system's performance and reliability properties over a wide range of design versions and operating environments. Such a model should be much less costly to exercise than elaborate simulations. Because of the key role of models, the methodology is strongly concerned with model construction and validation.

The report discusses various difficulties in evaluation, such as the com-

plexity of multiprocessing performance evaluation, the difficulty of distinguishing various factors (algorithms, software, operating system, fault diagnostics, etc.) that affect performance and fault tolerance, the use of formal and experimental analysis, and the special problems of security evaluation. Criteria and suggestions are given for the design of unified working environments and specific classes of tools that support this methodology.

The report addresses the following topics:

- Requirements for an evaluation methodology
- A model of evaluation based on domains
- Approaches to evaluation
- Specific evaluation objectives
- A systems evaluation laboratory.

Chapter 2

Requirements for an Evaluation Methodology

This chapter discusses general and specific requirements for a BM/C3 computer evaluation methodology. The general objectives listed are similar to those usually declared for system development methodologies. BM/C3-specific objectives address issues of requirements, architectural potential, and trade-offs.

2.1 General Requirements for a Computer Evaluation Methodology

A general methodology for evaluating multiprocessors will require (1) a theoretical framework, (2) a systematic approach to designing evaluations, and (3) a set of requirements and guidelines for developing evaluation tools. These objectives are similar to those of system development methodologies.

A Theoretical Framework for Computer Evaluation. Users, developers, and evaluators of systems need a consistent vocabulary and a way of reasoning about computers and computer applications so that they can determine the completeness, consistency, and effectiveness of practical evaluation efforts. A good theoretical framework should support the building of models (or sub-theories) of how computers are built and used and should suggest practical approaches to their evaluation. The models should allow prediction of behavior over a wide class of environments and designs. System

evaluation, which may require construction and execution of many complex processes over time, has very similar needs for a theoretical methodology as the system development function.

A Systematic Approach to the Design of Evaluations. An evaluation methodology should help evaluators define clear objectives and criteria, plan well-related and effective analyses and experiments, design tests that will give realistic and thorough coverage of service requirements, and derive meaningful conclusions.

A Consistent Set of Specifications for Evaluation Tools. A practical methodology should be supported by tools that are individually powerful, mutually consistent, and extendable. The tools should enable the evaluators to express evaluation objectives clearly and efficiently.

2.2 Specific Evaluation Issues for BM/C3 Computers

Following are examples of the major issues specific to Battle Management/C3 that must be addressed by an evaluation methodology.

Satisfaction of Requirements. For a candidate computer, are there configurations that will satisfy specific BM/C3 performance and dependability requirements, given appropriate algorithms and software implementation? If not, what are the most significant shortcomings?

A practical BM/C3 requirement set may specify several complex, multistage scenarios with different stress conditions in each stage and different priorities on availability, performance, recovery speed, fault coverage, etc. Computer systems designed for such requirements will have to be evaluated with respect to entire scenarios. A major complication in computer evaluation is the uncertain power of application software. Computer system performance will depend on both the computer and the application software, and it is not easy to decide which of these parts of a design might be responsible for a failure to meet performance specifications. Ideally, one wishes to specify requirements in terms of mission service objectives, such as the proper tracking of some number of targets. In practice, it may be necessary to translate these objectives to computationally oriented requirements that are based on an accepted set of BM/C3 algorithms.

Requirements for BM/C3 applications, including performance and reliability, security, lifetime, availability, maintainability, programmability, etc., are at present only approximately defined. Effective evaluations will require clear definitions of these properties for formulating requirements. Notions that integrate performance and reliability properties, such as performability (the performance delivered by an unreliable system over a given time interval), are important contributions.

A desirable side benefit of an evaluation methodology would be a clear set of criteria for expressing requirements for BM/C3 computers.

Architectural Potential. What are the potential performance and dependability of a subject computer architecture? The question might assume various projections of (1) technology, e.g., device and subsystem speed, capacity, size, weight and power, (2) fault class (the types of faults and possible fault combinations that are expected to occur), and (3) the BM/C3 application.

Design Trade-offs. What are the trade-offs within a candidate architecture among factors such as performance, lifetime, reliability, survivability, and programmability? This knowledge is of great interest to the BM/C3 system architect, who may wish to apply trade-offs, e.g., performance vs. expected lifetime of an individual computer at the global level. An interesting issue in evaluating trade-offs is: over what range of performance and reliability may trade-offs be usefully made?

The notion of trade-offs between performance and reliability factors also provides a useful framework for comparing architectures and evaluating the state of the art, analogous to power-speed tradeoff comparisons of device technologies.

Application Potential. What is the comparative potential of a set of different architectures with respect to different computations within the BM/C3 application? This question is of natural interest to sponsors of architectural developments.

Intrinsic limits. What are the limiting factors that determine the potential of multiprocessor architectures in general, and of particular multiprocessors, for BM/C3 computation? Issues include degree of diagnosability, real-time recovery, achievement of consistency in distributed-fault-tolerant

computing, irreducible performance costs of fault isolation, irreducible costs in synchronization of parallel processes, etc. This information is of crucial interest in the configuration of systems to meet large threats.

Critical Issues. What data elements are critical to the accuracy of evaluations? Examples might include data about fault modes, e.g., radiation effects, transient-error rates, software error rates, and application characteristics. What improvements to existing tools and theories could significantly reduce evaluation costs?

Chapter 3

A Domain-Based View of Evaluation

This chapter introduces the notion of domains as a means of structuring complex evaluations. Examples of relevant domains are given.

3.1 Evaluation Domains

3.1.1 The need for structure in complex evaluations

Computer evaluation is a highly fragmented practice, in which performance, reliability and security are usually evaluated independently, without reference to joint concerns. Furthermore, computer performance is typically evaluated using benchmarks that are only remotely related to the data and operating conditions of the intended application. Complex interactions among dependability factors such as reliability, maintainability, and security are seldom examined.

Although BM/C3 computational and dependability requirements are not yet well defined, candidate computer systems must, at some point, be evaluated with respect to a full set of very specific requirements. A satisfactory methodology must therefore provide a framework and techniques for conducting such a comprehensive evaluation. The proposed approach allows the evaluator to define requirements and capabilities in a set of domains that address all possible evaluation interests.

The following hypothetical (and extremely simple) system is an example of the need for multiple evaluation domains:

- Candidate system: Hypertree Inc., model II, 100 processor configuration
- Application problem: Track up to N1 type A objects and up to N2 type B objects
- Sensor data type: Radar
- Solution approach: Smith-Jones algorithm
- Performance criterion: Precision of tracking
- Environmental stress: Radiation, level 3
- Intended lifetime: 10 years.
- Maintenance availability: Remote, 0.95
- Fault tolerance approach: Multilevel, distributed.

Full description of each entry may be extremely detailed, involving complex functions and scenarios, with many variables. Full evaluation of the candidate system will require that all the domains of interest be explored systematically, but the number of distinct combinations of all the variables may make complete evaluation totally impractical. The situation clearly calls for some structuring of the evaluation process that will allow rational exploration of the range of possibilities.

The problem may be characterized abstractly by considering an evaluation as a measurement in a multidimensional evaluation space, where each dimension is derived from a domain that defines some aspect of the system or its operation. In this view, the required and actual behaviors are a pair of points in the evaluation space. In practice it will not be possible to compute the value of the *point* from a single analytic expression or experiment, and it will therefore be necessary to compose the evaluation as a set of values obtained in disjoint evaluation spaces. Special care must be taken to perform unified evaluations for pairs of domains that have significant interaction, e.g., security and fault tolerance. While practical compromises may have to be made in the goal of fully united, multiple-domain evaluation, defining an evaluation within the framework of a single multidimensional evaluation space provides a useful conceptual framework for describing and comparing evaluations. In practice, describing an evaluation in terms of a set of orthogonal domains can help the evaluator to design a set of experiments that

will efficiently address all relevant evaluation concerns. It also can help in comparing the results of different evaluation experiments.

3.1.2 Criteria for Domain Definition

The evaluator has the freedom to define the applicable domains, but success in the overall evaluation may rest on the correction section of domains. The general goal of a set of domains is that they simplify exploration of the evaluation space and that they correspond to the interests of users and designers. Important criteria for defining a domain are:

- The definition of the domain allows a simple and comprehensive expression of a system goal, property or operating condition.
- The subject is sufficiently complex to justify an individual modeling effort.
- The concerns of the domain are reasonably orthogonal to concerns of other domains.

These criteria are not absolute. One can see this in an example to follow. There the operating environment is described by a pair of domains, one emphasizing the physical problem to be solved and the other the data exchanged between the computer and its physical environment. These domains are not fully mutually orthogonal, and in simpler applications might well be merged into a single domain. Such arbitrary separation into distinct domains can simplify the description of the environment and allow for convenient isolation of design decisions about how evaluation data should be structured. Another violation of orthogonality is the notion of a domain of "stress", to be discussed.

Within each domain, we envision the creation of models, probably with hierarchical structure, that express the essential ideas of the domain. Many modeling techniques are available and are widely used, including those based on equations, graphs (static and dynamic), queues, deterministic and stochastic state machines, algorithms, and programs.

3.2 Example Domains

The following examples are a selection from the many possibly significant domains. In practice, some domains may be sufficiently complex to merit further decomposition into subdomains.

Life-cycle Viewpoint. The many participants in a system's life-cycle–Users, architects, algorithm designers, language and software engineers, maintenance persons, and operators–all have unique viewpoints on computer system behavior. These viewpoints may be expressed in separate models and may require different experimental approaches. Within the framework of a given view, the evaluator may wish to construct and use limited views in order to explore specific properties of a system. These viewpoints may serve as bases for distinct domains or as restrictions or guides in the exploration of other domains.

Application Problem. The computations employed in an evaluation must closely reflect the real-world problem to be solved in the BM/C3 application. This is one of the most poorly handled issues in computer evaluation, and so deserves special attention. Programs used to generate tests for computer systems must bear a clear and direct relation to the physical world of BM/C3.

Ultimately, a computer system must be evaluated not on the basis of processing speed, but on the basis of its effectiveness in solving the application problem. Models are therefore needed for describing the physical, informational, and control aspects of the problem to be solved. BM/C3 examples include (1) physical: missiles and weapon dynamics, (2) informational: knowledge to be shared with other computers, and (3) control: strategic and tactical BM/C3 responses. The application domain interacts with the computer domain through the medium of input and output data.

Problem Data. Models are needed to describe the data that arise from the problem domain. These include sensor, communication, and weapon-command data and their types, rates, and distributions, both in time and in location. These models serve as a bridge between the application domain and the computer architecture; thus (1) the models should be closely relatable to models in the problem domain in order to assure their validity, and (2) the models should be capable of expressing significant architectural issues, such as parallelism, granularity, and data dependency.

The Subject Computer. Models are needed to describe the elements of the subject computer, including algorithms, programs, operating system, processing functions, input-output functions, possible interactions with other computers, fault-diagnostic and recovery functions, and maintenance

services. The models should allow parametric representation of (1) significant performance and fault-tolerance issues, including critical determiners of performance, such as communication, synchronization, and control, and (2) of fault tolerance, such as observability and reconfigurability. The models must be easily relatable to the models of the data domain.

The subject computer is itself a complex of systems that may be defined with different degrees of precision and permanence. For example, algorithms, application programs, and system software are always subject to change in order to increase functionality and improve performance.

Stress. We define stress as any condition in a system or its environment that tends to compromise the achievement of full performance potential. It includes faults in hardware and software implementations, faults in hardware or software design, reductions in specified maintenance service, overloads of input data beyond expected or specified values, and possible malicious actions to interfere with operations. A particular stress even may contain several coincident types of stress. The notion of computing under stress is implicit in the concept of performability, which addresses the variation in computing performance with loss of resources.

A unified model of stress is needed, even though individual stress types may also be manifested in other domains. Examples include (1) design faults in hardware and software, (2) implementation faults in hardware and software (these are the so-called conventional faults) including both undetected manufacturing faults and faults that occur during operation, (3) extreme conditions of data beyond specified values, such as overloads or degradations in the information provided about the environment, and (4) interrupted or improper maintenance and operator service. These models will often be constructed with several levels; for example, there may be one model of a physical or cognitive process that gives rise to faults, such as radiation, metal growth, or incorrect problem analysis, and a second model that describes how these processes are manifested as faults in particular computer elements. Of course, since computer elements are themselves organized hierarchically, faults at the lowest levels will have various manifestations at higher levels (up to the point where they are masked or removed). Stress effects may be arbitrarily complex in extent, form, and time behavior.

Operating Modes. A BM/C3 computer will operate within the context of a multiple platform system. Among its responsibilities may be (1) local

BM/C3 computations, (2) cooperative computations involving other platforms, (3) backup computations to accommodate failures in other platforms, and (4) communication relay service. It will participate in diagnostic and remote repair operations as either the subject of ground maintenance operations or as an intermediary in the maintenance of other systems. It may be required to operate independently of ground maintenance operations for some period of time. Its modes of operation may vary over its lifetime, e.g., within an orbit, in different system readiness conditions, during an engagement, and in maturity and senescence. For each mode, different types and levels of dependability may be required. Dependability types include reliability, availability, maintainability, autonomy, security, and performability.

Descriptions of this domain should describe the service, durations, and dependability factors required for each mode, and the required speed of transition between service modes.

Programmability. Current multiprocessor architectures employ several different approaches to computational parallelism, e.g., shared memory or disjoint memory multiprocessing, single or multiple instruction multiprocessing, and systolic processing. Each approach provides high performance for some but not all kinds of parallelism. A practical set of BM/C3 algorithms will contain a variety of forms of parallelism, and for those forms that are not well supported by a particular computer, special effort in programming will be needed to overcome the poor match of algorithmic and machine forms of parallelism. A similar difficulty may pertain in matching a standard programming language, e.g., Ada, to a particular architecture. The difficulty of this programming effort may be a significant evaluation issue.

Chapter 4

Approaches to Domain Modeling

This chapter discusses some fundamental issues in the use of the domain modeling technique to guide evaluations. The first issue discussed is model construction and validation. The second issue is the design of evaluation experiments. Several basic approaches are presented and compared.

4.1 Model Validation

4.1.1 General Issues in Model Validation

Model Construction

Construction of accurate models of the subject system and its environment is a central activity in the proposed evaluation methodology. The literature of computer engineering (see the Bibliography for general references) is abundant with examples of useful models for all aspects of computer design. However, there is a need for new models that describe system behavior under stress, e.g., transient performance degradation during error recovery, and behavior of load-balancing schemes under severe overload. A good set of system models (covering interests such as throughput, overload response, fault recovery, and degradation) will provide an economical means for predicting system behavior for a wide range of possible values in:

- The application and operating environment

- The architecture, e.g., changes in the number of processors and the speed of particular subsystems
- The rates of the several fault types
- The behavior of larger systems in which the subject computer is a component.

Model Validation

Exercising the models will be far less expensive than exercising the full system for all the conditions of interest[1], but the validity of those exercises will depend on how well the models represent the real objects they are supposed to describe. Computing processes are very difficult to model accurately because they are determined by complex logical relations and are not subject to the conservation relations that simplify the analysis of physical systems; furthermore, modern multiprocessors and distributed computers are not deterministic in their sequential behavior. A central problem in computer system evaluation is therefore the ***validation of models***.

In other contexts, the term ***model validation*** may refer to the determination of the logical consistency of a design with a specification model, or of a specification model with a requirements model (sometimes called ***design validation***). In the present context, it means the determination of whether a system model provides an acceptably accurate representation of a system design from a chosen viewpoint. As an example, fault free performance of a system may be represented by a queueing model, with queues identified with processors, memory levels and operating system service. The behavior of the model is only an approximation of the behavior of the real system, but it must be a reasonable and useful approximation to be valid.

Typical questions that arise in validating models are:

- Is the model of the computer consistent with the design?
- What are realistic values for the parameters of the model?
- Is the behavior predicted by the model valid for the specified application and for the specified stress condition?

[1] Although in some cases, it may be more expedient to use a real subsystem in a test than to develop a simulated version

Model validation is a process of reasoning from evidence, where the evidence may be axiomatic or experimental. We review these two types in the following discussions. In practice, validation will require a combination of these types.

We note again that there are many different possible viewpoints and levels of detail in modeling, thus the *model* in the following discussion may range from a very limited facet of a system to a very comprehensive view. An appropriate strategy for arriving at an understanding of a system would be to conduct a sequence of model evaluations employing a variety of models of different sizes and viewpoints. As a general rule, validation of models should be given the same care as validation of designs.

4.1.2 Model Validation Based on Axiomatic Evidence

One way to construct a model is to form an abstraction from a set of facts or rules whose truth is assumed as axiomatic. For example, the design of a floating-point multiplication unit may be modeled by a deterministic algorithm. Such a model is considered valid if it can be proven to be logically consistent with the assumed facts, in this case, the basic laws of digital arithmetic and the rules of behavior of the proposed logic elements.

Ideally, validation of a model based on axiomatically assumed facts should be done by a mathematical proof. This is feasible for small objects, and has been accomplished on nontrivial systems by using special machine aids.[2] Tools for proving practical systems are under development, but they are not yet ready for industrial use.

For large design objects, a structured review by experts is a more practical way to verify that a model is consistent with assumed facts.

Ideally, design validation should confirm the correctness of a model for all possible data inputs. This may be impractical for large systems. If only a subset of input conditions is considered, there is an additional responsibility to verify that the subset properly represents the intended application.

4.1.3 Model Validation Based on Experimental Evidence

Many of the factors that govern system behavior are not known with logical precision and must be estimated experimentally. In this case, validating the

[2]P.M. Melliar-Smith and R.L. Schwartz, "Formal Specification and Mechanical Verification of SIFT: A Fault-Tolerant Flight Control System," *IEEE Tr. on Computers* vol. C-31 (July 1982) pp. 616-630.

model will require statistically sound inference from the observed behavior to the properties of the model. One such example is the modeling of fault detection and recovery. In this case, faults occur randomly, and the detection and recovery process may be too complex to be modeled mathematically. For fault occurrence, some statistical distribution may be available in published reports or from local experiments. For fault detection and recovery, a set of experiments may be performed to determine the distribution of processing times and probabilities of success. These results are then entered as parameters in the proposed model.

Experimental verification requires that the forces applied to the subject system by the test environment, i.e., the test data and the physical stress conditions, adequately represent the intended operating environment. This is a very serious problem in evaluation, where a system design may be available well before the environment has been fully specified.

4.1.4 Axiomatic and Experimental Validation: Combinations and Trade-offs

Different combinations of reasoning and experimentation may be employed to validate a model. At one extreme, only a minimal amount of experimentation may be conducted to measure the non-logical processes, and the remainder of the validation may be strictly deductive. At the other extreme, a model may be validated purely by experimentation, even though some parts of the system behavior could be derived from the design. If the behavior of interest has a low frequency of occurrence, as in the case of faults and fault tolerance, the cost of testing may be prohibitive. (For example, achieving 90% confidence in a design that aims for 10^{-9}per year probability of failure of a fault-tolerant system will require on the order of 10^{10} system years of testing!)

In principle, deductive reasoning, prudently applied, can greatly reduce the costs of experimentation. For example, every N-fold symmetry that can be proven about a system's behavior may be used to reduce testing by a factor of N. As another example, if a design depends upon the properties of a right triangle (or any other mathematically well-known entity), it would be very wasteful to test those properties experimentally!

A second justification for combined experimental and axiomatic validation arises in model calibration. For example, when a Markov model is used to describe degraded operation, it may be more practical to estimate the transition rate experimentally than by detailed analysis of the recovery

processes.

It is clear, therefore, that validating a model should be a combination of deductive and statistical reasoning, with the maximum amount of deductive reasoning applied consistent with the amount of effort required and the resulting degree of confidence. A special merit of experimentation is that some crude results are usually obtainable, while an overloaded deductive process, if poorly structured, may fail to produce any useful results at all.

Simulation analysis also enters into this equation. As mentioned above, a simulation model approximates a full logical representation of a design. Extrapolations from experimental tests on the model may help to demonstrate that a proposed model is consistent with the design of the system; aspects of behavior that depend on random processes must still be determined by real-world experiments.

4.2 Approaches to the Design of Validation Experiments

This section discusses some of the problems of designing validation experiments, including problems of predicting computer performance, and approaches to effective test design, and evaluation of the potential of non-fault-tolerant designs.

4.2.1 Problems in Predicting Computer Performance

Experience has shown that choice of test data for validation experiments is a very difficult problem, even for simple, von Neumann-type computer configurations. Standard benchmark programs are notoriously unreliable predictors of performance, and even sample programs extracted from working application programs fail to eliminate surprises when computers are placed in actual service. Some sources of difficulty in such computers are as follows:

- The existence of memory hierarchies, in which access times differ greatly among the levels, may produce great variations in program speed due to the mismatch of the scope of program references to the storage capacities at individual levels of the hierarchy. Memory hierarchies may include registers, caches, main memories, and secondary memories. Uncertainty in program execution speed may result even in two-level hierarchies, due to the unpredictability of the scope of memory references for different input data, among different programs, and

even within programs. This effect is significant for individual program references and it becomes severe for changes in program context, where a large amount of data may have to be relocated within a hierarchy.

- Uncertainties due to competition for resources (data, channels, processors, etc.) may lead to load-dependent thrashing or deadlocks.
- Differences in instruction execution speeds may cause greatly differing program speeds, due to the unpredictable branching behavior of programs.

In parallel-processing computers, these effects are compounded by many additional sources of uncertainty, such as:

- How frequently one process may demand access to data located in the memory hierarchy owned by another processor
- Delays in synchronizing interdependent processes
- Delays in resolving contentions among processes for resources, e.g., buses and memories; this may occur between processes within a coordinated set and, at a higher level of abstraction, between unrelated processing sets that share resources
- Default rates in pipeline processing.

From the point of view of system evaluation, these factors illustrate the great difficulty of selecting test programs and data that will properly represent a computer's intended application. From the point of view of system performance, the factors illustrate the importance of selecting and matching architectures to fit the application.

4.2.2 Approaches to Effective Test Design

This section reviews basic problems in the design of tests and some possible solutions. Issues include machine-based vs. application-based test design, problems in the separation of architectural and software effects, problems of instrumentation, and some special problems in reliability testing.

Machine-Based Test Design

At present, test design is an art in which several viewpoints are undertaken, some machine-centered and others program-centered, as in the following examples:

Tests based on a machine-centered view:

- *Critical-point testing.* Tests that explore the extreme values or critical points of a given computer design, such as communication bottlenecks, discontinuities within a memory hierarchy, and context changes.

- *Piecewise-continuous testing.* Tests that explore continuous regions of a given computer design, in which performance is a smooth function of the level of activity, e.g., flow through interconnection networks and program execution without change in context.

Tests based on a program-centered view:

- *Critical-process testing.* Tests that emphasize critical program points, e.g., synchronization and function calls.

- *Representative-mix testing.* Tests that reflect the expected mix of programs of different types for data values that span the intended application.

Much work remains to be done on the problem of test design.

Application-Based Test Design

The foregoing techniques are usually applied with minimal concern for the application domain, in order to focus on intrinsic hardware and operating system limitations. Ultimately, the importance of these limitations will depend on how they are exercised by application programs. It is therefore essential to take an application-based view of the system.

An example of application-based testing is the use of a parameterized test generator. The generator models real-world processes, e.g., a set of moving targets, so that real-world conditions are defined by a set of parameters that may be set by the experimenter. For each setting, data are generated that would be produced by a sensor that observes the real environment.

In more general terms, a model of the problem itself is created and is used to generate test data. For large problems, a hierarchy of models may

be necessary in order to limit individual model complexity. Two significant benefits of this approach are that: (1) well-structured models will aid in the systematic exploration of a problem space, and (2) the evaluator's assumptions about the problem domain are made explicit for further review and evolution.

Separation of Architecture and Software Effects

One of the most difficult problems in computer system evaluation is how to distinguish the intrinsic limitations of the underlying architecture from the limitations of the application and operating system software—limitations in algorithms, languages, compilers, schedulers, resource managers, and diagnosers. The several kinds of limitation are not independent, and their interrelations are not linearly related. Thus, one algorithm for an application may be very poorly matched to an architecture, while another may employ its features with great efficiency; a language or operating system construct for interprocess communication may be unnecessarily clumsy; or a fault diagnosis program may make poor use of machine-level diagnostic information. Even functions that are close to the basic machine design, such as cache management disciplines, might be subject to modifications that would give great improvements in performance.

Separation of hardware and software effects will require considerable ingenuity in the design of experiments.

Instrumentation Problems

Disturbance of an observed process by the process that observes it is a fundamental problem in physical experimentation (known as the Heisenberg effect), and is a very significant problem in computer measurements. The instrumentation provided in recent generations of computers is much improved over that in previous generations, but inevitably there will not be enough for the deep evaluation experiments that will be required. The two possible strategies will be to add special hardware and software instrumentation in the subject system, thus introducing uncertainties in observed performance, and to design powerful analysis programs that will extract the maximum information from the available observables.

Special Problems in Reliability Evaluation

Several primary questions about reliability in a computer system are:

- What is the distribution of the occurrence in time and location of all the expected fault types, and what is the probability of occurrence of unexpected faults?
- How are the faults manifested within the proposed design?
- How effectively are faults and errors processed by proposed fault-tolerance mechanisms?

This section discusses some of the issues that influence the design of experiments for answering these questions.

Fault Distribution. Knowledge about how faults may be distributed in a proposed computer can be obtained only by experiments or field observations. Unfortunately, there may be a practical limit to the accuracy of such data, because new systems tend to employ very recent, high-performance components for which little reliability data is available. Accelerated component testing may help. For faults arising in the packaging, e.g., transient faults due to internal signal interference or to external radiation that has penetrated the enclosure, testing of physical prototypes may provide useful information, although the packaging will probably be affected by the chosen architecture.

Fault multiplicity is a significant issue in harsh, long-term missions. There are several kinds of multiplicity,including multiple independent faults in a single logic element, clusters of faults within a functional module, and sets of faults that are located in several modules. In long-term missions, there may be a substantial accumulation of faults that originated at different times. A second type of multiplicity is the co-occurrence of faults of different types: permanent, transient, hardware, software, and design. Another dimension is extent, i.e., faults may range from single faulty elements to entire computer subsystems.

Experimental designs should thus start with a clear definition of the fault types that are expected and their multiplicities.

Fault Manifestation. The way in which a fault condition is manifested in erroneous computing behavior may be very technology-dependent. For example, an open circuit may have entirely different consequences for logical behavior in different logic technologies (e.g., it may appear as stuck output, added delay, or data-dependent error). Knowledge about how a fault is

manifested may be obtained by analysis, by simulation, or by physical experiments in which actual faults are injected in a prototype machine. The latter approach is becoming less attractive as the density of components increases.

In the standard approach to error analysis, a fault is assumed at the lowest computing level, e.g., a circuit failure, and its effects are traced outward from the initial module to other modules, and upward through levels of the system. A more general approach would be to allow any level of the system to be defined as faulty, without reference to the *primary* fault.

The effect of a fault on a computation will depend on the location of the fault and the point in the computation at which the fault is manifested as an error. The impact on computed results may vary widely.

Fault Tolerance. The goal of fault-tolerant design is to achieve acceptable computation despite the occurrence of faults. The three basic objectives of fault tolerance are (1) fault containment, (2) error recovery, and (3) fault elimination. The purpose of fault containment is to limit the extent of propagation of errors within the computing system. This should be an intrinsic property of the system design. The purpose of error recovery is to substitute the correct value of a computation (or a best approximation) for an erroneous value. The substitution may occur in real time (in the literature of fault-tolerant computing this is called *Forward Recovery*, or with some delay, *Backward Recovery*). The purpose of fault elimination is to modify the computer, by hardware or software means, in a way that prevents the fault from further affecting its operation. Fault elimination requires *Fault Detection and Location, Reconfiguration, and Reinitialization* (to resume processing with minimal loss of productivity).

Fault-tolerant designs may employ several strategies, depending on circumstances of time and processing type. For example, in some cases there may not be time for fault elimination, so the only feasible protection is through error recovery. In some cases, it is crucial to preserve the integrity of certain stored data, while in others, the data changes so rapidly that erroneous values may be allowed to remain in the system until they are replaced by normal processing.

These many processes require quite complex designs, and the testing of these designs will involve very substantial effort. Experimental design for testing fault tolerance must address the problems of fault distribution and fault manifestation discussed above, in this section. Clearly, the value of

a fault-tolerant design will depend upon whether the design addresses the fault types and distributions that will occur in the real application, and its effectiveness will be a function of (1) how well it recognizes and diagnoses the errors that are manifested in the machine by those faults (i.e., its coverage), and (2) how well it survives unanticipated faults.

4.2.3 Fault-Tolerance Potential of Non-Fault-Tolerant Designs

It may be desired to estimate the potential for fault-tolerant performance of an architecture or a family of architectures for which fault tolerance was not an initial design criterion. Several approaches may be taken, e.g., (1) assume use of software-only mechanisms in the given machine, or (2) assume a fault-tolerant design based on the given architecture.

In the first approach the assumed software-based mechanisms may include replicated data processing and data storage, programmed error checks, and software-based reconfiguration. Dependence on only such mechanisms will, or course, be very costly in performance. It is extremely unlikely that such a system will be able to satisfy BM/C3 performance requirements. Some designs may seek to reduce performance costs by performing error checks only periodically, and by restricting high-redundancy levels to only the most critical processes. A worthwhile evaluation objective is to determine the trade-off of performance and reliability for such compromises. The system will still be vulnerable to failures at the basic communication level, where single faults may destroy all operations. One set of tests may seek to determine the extent of that vulnerability in the subject design.

In the second approach, new fault tolerance mechanisms must be assumed to be operative at many levels of the system design. A useful approach in evaluating such hypothetical designs would be to assume particular fault-tolerance performance of a subsystem, e.g., the coverage of a module's built-in fault detector, or the time required for error recovery in a single process, and determine its impact on the rest of the system.

The accuracy of results under these various assumed design variations will be poor, but the evaluation exercises may be helpful in uncovering important weaknesses in a particular architecture.

Chapter 5

Specific Evaluation Objectives

This chapter presents several examples of system evaluation objectives. The first set of examples assumes that only a single behavioral or descriptive domain is of concern, e.g., performance or fault tolerance, while successive classes assume several domains are of joint concern, e.g., performance under stress. Restricting an evaluation to only a single domain at a time will hide important interactions, but it may be useful for initial exploration of the evaluation space. For example: in the first stage, a single-domain test may be employed to find the weakest parts of a design in a first domain; then, in a second stage, a second domain might be explored to find the weakest combination, and so forth. This might not be guaranteed to find the very worst performance, but the experience gained in the analyses may be suggestive of an optimal evaluation strategy.

The discussion does not propose the particular combination of evaluation techniques (formal analysis, simulation, test, etc.) that would be most effective for the various objectives. As mentioned previously, a very difficult challenge in evaluation is to separate the effects of hardware, operating system, and application program in evaluating performance.

The following special terms are employed in the discussion of evaluation objectives:

- *Granularity* refers to the size of a meaningful computing aggregate, both in hardware and software processes.
- The term *coverage* appears in several contexts. In discussions where

there may be some ambiguity, the term should be qualified, e.g., fault-detection coverage, error-recovery coverage, reconfiguration coverage, etc.

The remainder of the chapter discusses several important single-domain evaluation objectives, including performance, fault tolerance and security, the objective of stressed-performance evaluation, and several multi-domain tradeoffs.

This section discusses evaluation objectives for performance, performance trade-offs, stress, and security.

5.1 Unstressed-Performance Objectives

The following evaluation objectives aim to determine system performance for various configurations and operations:

Taking a machine-centered view of the performance of a single processor, determine for all combinations of:

- Processing type: Numerical, Logical, Database, I/O
- Program size, including data:
  - Small (containable in processor and cache)
  - Medium (requiring significant access to main memory)
  - Large (requiring access to secondary store).
- Context switching rate:
  - Minimal (single context);
  - Medium (moderate number of different contexts and rate of change)
  - Large (many contexts and/or high rate of change).

Taking a machine-centered view of the performance of multiple processors, determine for all combinations of size of the activity:

- Process synchronization rate: varied rates of synchronization among processes in different processors.

- Data transfer rate: varied rates of movement of data between different processors.
- Activity concentration: varied degrees of concentration of processing activity within the processor set (e.g., at shared memories); varied degrees of concentration of data flow at locations within an interprocessor communication network.
- Process interpenetration: varied degrees of activity of a non-local process within a processor (e.g., remote access to local data, remote execution of local processes).
- Order of multiprocessing: varied sizes of multiple-processor activity with respect to (1) fraction of the processor set engaged in a single process, and (2) number of levels of memory significantly engaged.

Taking a problem-centered performance view, determine single-purpose and combined service for an appropriate range of problem sizes and complexities, on the following problems:

- Surveillance and tracking: Sensor processing, pattern recognition, object classification, track membership, track distinction, track characterization.
- *Inter-platform communication*: Exchange of battle information, communication relay service.
- Data-intensive engagement: Geometric and physical calculations, data-base management.
- Decision-intensive engagement: Estimation, goal searching, planning, optimization, evaluation.

Taking a combined machine and problem-centered performance view, determine the system's performance for representative applications over the full range of input data levels.

5.2 Performance-Independent Fault Tolerance

For all fault types in the assumed fault classes and for all multiplicities of faults of the same type and of different types within the range of assumed fault co-occurrences, determine how faults are manifested and detected as follows.

5.2.1 Fault Manifestation

- Elaborate and determine the distribution of the different ways in which an assumed fault condition in a given functional element affects its behavior, i.e., the probability that it will be manifested as a stuck-output (zero, one, null; fixed or cyclic), an incorrect value, excessive time deviation, a change in stored information, inconsistency in data replication, etc.

- Determine the persistence in time of the effects of a fault on system state at a given point in the system, (ranging from quick self-recovery, to permanent change, to total breakdown).

5.2.2 Fault Detection

- Fault Coverage: what fraction of all possible faults in a defined fault class are detected by a test process?

- Fault Latency: what is the distribution of times required to discover faults in normal processing by an on-line observation process?

- Fault-detection Autonomy: to what degree is a functional unit capable of detecting its own faults?

- Resolution: what is the largest hardware or process granule within which faults are indistinguishable?

- Fault isolation: How far do faults propagate within the computer? This issue may be crucial in shared memory architectures, and difficult to analyze. Isolation may be described in terms of module distance within a system level or level distance within the system hierarchy, as follows:

  - In hardware terms, as the greatest distance, the amount of stored data disturbed or the number of units whose behavior is changed.
  - In process terms, as the extent of process corruption due to a fault, e.g., the longest duration of disturbance for a single process, or the number of processes affected.
  - In system terms, as the extent of penetration of faults within the processing hierarchy, e.g., application, operating system, hardware, fault-tolerance control.

- For large failures, the fraction of processing resources (including critical data and processes) removed from the system.

- The coverage of forward error recovery (*Error Masking*), i.e., the fraction of possible errors (of the expected fault types and multiplicities) completely masked or fully identified but not corrected, for processes of specified granularity.

- Backward error recovery:

  - Error coverage: fraction of errors that are detected and successfully corrected.
  - Error latency: time to detect the existence of an error.
  - Recovery completeness: the fraction of erroneous results corrected;
  - Error containment: the number of processes(ors) that are affected by the recovery process relative to the actual number of erroneous processes(ors). This is intended to measure the impact of the recovery process on non-faulty processing.

- Reconfiguration (elimination of faulty elements):

  - Reconfiguration coverage: the fraction of reconfigurations attempted that are successfully performed (although not necessarily with complete elimination of faulty components).
  - Reconfiguration completeness: the fraction of faulty elements in a multiple-element fault set that are successfully removed from a system by a reconfiguration.
  - Reconfiguration efficiency: the fraction of the resources eliminated by a reconfiguration that are actually faulty
  - Reconfiguration completeness: the fraction of multiple state-settings successfully accomplished.

- Reinitialization: (setting of system state for the resumption of processing following a reconfiguration).

  - Efficiency: the fraction of nonsavable computations relative to the computations that are actually abandoned by a reinitialization.

5.3 Stressed-Performance Objectives

The following evaluation objectives extend the general notion of performability, a measure of performance in faulty systems, to include response to overload conditions in non-faulty systems. Stress types considered in this section are overload, reduced configuration, fault tolerance effort, and untolerated faults.

Overload Stress. Determine the relationship of performance vs. problem size for a range of problem size that substantially exceeds the point of peak performance. A useful rough indicator for problem size in BM/C3 applications may be the number of targets; a more effective indicator would be that number together with factors that reflect the amount of ambiguity (and hence increased computation) in target observation or decision making. The performance loss should be evaluated both in volume and in time, in order to capture transient response, e.g., the amount of time required to recover as a function of the amount of overload.

Reduced-Configuration Stress. Determine the ability to handle overload stress as a function of the amount of reduction in configuration size (due to past fault conditions), assuming that reductions were accomplished by successful reconfiguration actions. A simple indicator of overload performance may be the size of application problem that is correctly solved. A more informative indicator would reflect built-in strategies that dynamically make sacrifices in the precision of solutions or in the range of problem type accepted.

Fault-Tolerance Stress. Determine the time function of performance loss during a fault tolerance action for a realistic range of fault severity. For example, how much processing work is lost in recovering from a fault condition? Several levels of performance indicators may be of interest, including degradation of BM/C3 service and loss of critical stored information.

Untolerated-Fault Stress. Determine the performance loss for a significant range of faults that have not been masked or eliminated by fault tolerance or remote maintenance actions. Significant cases of such faults include (a) fault conditions whose type and multiplicity exceed fault-tolerant design

objectives, (b) faults that are within design objectives but are imperfectly handled, and (c) intrusion faults.

5.4 Performance Trade-Offs

This section gives examples of important relations among conflicting application requirements that will affect architectural trade-offs.

5.4.1 Performance Lifetime

A good design will provide for highly precise fault diagnosis and elimination in order to minimize the waste of resources occurring in fault reconfiguration, and thereby maximize lifetime. Such a design would require a sacrifice in performance due to the reduced granularity of observable processing. The following objectives attempt to measure the trade-off between performance and lifetime.

- Determine the relationship between performance and granularity of fault diagnosis and reconfiguration, i.e., the extent to which performance is degraded by the desire to perform fault tolerance at a low level of granularity.
- *Determine the degradation of performance as a function of progressive* loss of resources due to reconfiguration, for varying precisions of fault diagnosis, i.e., the amount of performance that is sacrificed by using imprecise fault location techniques.

5.4.2 Autonomous Performance

The following objectives attempt to measure the trade-off between performance and required duration of autonomous (no ground support) operation. It is assumed that long periods of autonomy will require a much higher level of local fault tolerance capability, in the form of a higher level of redundancy (at least during periods of autonomous operation) or a more complex diagnostic process. Such higher level of local fault tolerance capability will inevitably subtract from local processing power.

- Determine the relationship between level of performance and level of redundancy that is required to survive periods of no remote maintenance service, as a function of period length.

- Determine the performance during periods of autonomy in response to different forms and levels of problem stress, for varied requirements on the duration of autonomy.

5.5 Security

This section reports on current evaluation practice for secure systems, discusses the interaction of security with other dependability design criteria, and gives several examples of insecure behavior that illustrate some of the problems that will be encountered in system evaluation.[1]

5.5.1 Security evaluation methodology

The so-called "Orange Book"[2] provides an extensive list of criteria against which supposedly secure systems can be – and have been – evaluated. These criteria are being used by the National Computer Security Center to evaluate a growing number of systems. The NCSC's Evaluated Products List (periodically updated) summarizes these evaluations.

The criteria include detailed requirements for discretionary access control, object reuse and deallocation, security labels, label integrity, exportation of labeled information, exportation to multilevel devices, exportation to single-level devices, labeling of human readable output, mandatory access controls, user sensitivity labels, device labels, identification and authentication, auditing, trusted paths between users and the system, system architecture, system integrity, security testing, design specification and verification (for higher degrees of security assurance), covert-channel analysis, trusted facility management, configuration management, trusted recovery, and trusted system distribution, plus various documentation requirements. Many of these requirements are such that the noncompliance of a system can result in serious security flaws. The methodology of evaluation necessitates thorough investigation of each relevant criterion and determination of the extent to which it is met. The Orange Book defines a set of levels (C1, C2, B1, B2, B3, A1) that represent progressively more rigorous evaluations and correspondingly greater assurance.

The Orange Book provides a valuable instance of a detailed evaluation methodology for one particular range of requirements, notably those relating

[1] This section is by P.G. Neumann, SRI International.

[2] *Department of Defense Trusted Computer System Evaluation Criteria* (TCSEC), National Computer Security Center, DOD-5200.28-STD, December 1985.

to security. However, the Orange Book still has some serious deficiencies, e.g., it does not apply naturally to computer networks, virtual systems, and various unusual architectural approaches, and it does not adequately consider system integrity and less primitive security policies such as those encountered in databases. Nevertheless, it represents an important starting point.

It is interesting to contemplate whether security and integrity requirements are substantively different from fault tolerance and performance requirements. Indeed, the use of trusted computing bases (TCBs) in developing secure systems relies on the presence of some sort of system boundary outside of which intentional or accidental compromises of the desired TCB properties cannot occur. Similar properties exist with respect to fault tolerance and performance (for example), and thus similar criteria and similar methodologies for evaluation systems against those criteria seem perfectly reasonable. Even so, significant research questions still must be answered before such a comprehensive approach to an evaluation methodology would be feasible.

5.5.2 Interactions among Requirements

The interactions of security with performance and fault tolerance are pervasive, as illustrated by the following difficulties:

- A system that is not secure may have its fault tolerance or its necessary performance undermined: an intruder may be able to crash it, or the system may run amok due to its own lack of self-protection. Security must imply protection not only against intruders and authorized users, but against other parts of the system itself.
- A system that is not fault tolerant may have its security and its performance requirements undermined: its behavior under uncovered fault modes is generally unknown.
- A system that lacks either fault tolerance or security may not be able to fulfill other critical requirements such as human safety.
- A system that is not safe to use (e.g., in a life-critical environment) could still satisfy the security, fault-tolerance, and performance requirements – although those attributes might then be irrelevant.

Fault-tolerance, safety, security, and performance requirements generally are vulnerable to weak links anywhere in the system and its operation. There are serious problems in recognizing and adequately defining all of those requirements and in designing and implementing a system that can satisfy all of them with some meaningful measure of trustworthiness. It is clear that these requirements interact strongly with one another and that the system architecture must encompass the full spectrum of critical requirements from the beginning of system development.

For these reasons (among others), any evaluation methodology must be considered to be incomplete unless it reflects the full set of critical requirements, including security.

5.5.3 Architectural Issues

Neumann gives an example of how system design may provide a unified treatment of dependability requirements (e.g., both fault tolerance and security).[3] His approach is to construct a hierarchy of encapsulated abstract type managers such that each layer in the hierarchy cannot be compromised by higher layers. This approach reflects experience gained in designing kernel-based systems and trusted systems for security and fault-tolerance requirements.

There is great intuitive appeal in the notions of completely embedded systems, autonomous systems, trusted computing bases, and kernels. The use of encryption is also attractive. Unfortunately there are still serious vulnerabilities and other pitfalls in such systems. Simplistic solutions are often badly flawed.

For example, suppose a computer subsystem is designed to execute within a completely embedded implementation in a dedicated system. It is still possible that the communications (whether encrypted or not) can be spoofed, by subverting or bypassing one of the other communicating systems, or that the integrity of the entire application can be subverted by compromising the underlying operating system.

The ARPANET collapse on October 27, 1980 is a case in point.[4] Accidentally corrupted versions of a status message (resulting from dropped bits) were propagated throughout the network and functioned as a data virus that contaminated every node in the network. The result was that

[3] P. G. Neumann, "On Hierarchical Design of Computer Systems for Critical Applications," *IEEE Trans. Software Engineering*, SE-12, pp. 905-920 (September 1986).

[4] Eric Rosen, "Vulnerabilities of network control protocols," *ACM Software Engineering Notes* Vol. 6 (Jan. 1981) pp. 6-8.

the entire network became inoperative. Furthermore, each node had to be shut down manually, because no messages to the nodes could get through. Operation could not be reinitiated until every node had been shut down, because otherwise the accidentally propagating virus condition would again have propagated and recontaminated the network.

A mail message containing hidden control characters and escape sequences is an example of a simple attack that was a great surprise to many people when it was first described in public. Various other flawed interfaces (e.g., containing trap doors or permitting Trojan horses) have been found in systems, conferring the power to undermine the entire system. Indeed there have been numerous recent exploitations of such flaws, including the West German intrusion into NASA and DoE systems.

Attempting to evaluate a system against an incomplete set of requirements poses significant problems. For any one requirement the system design may appear reasonable, whereas attaining all of the requirements concurrently may be impossible. Another problem is that different techniques may be employed for each requirement, and those techniques may not integrate well with one another. Thus, great care must be taken to obtain the proper perspective and to be suspicious of the results of any such partial evaluations.

Multiprocessing architectures pose some special problems for security when several levels of security coexist within a machine. The cost of verifying access rights at every memory access and interprocess communication may be unacceptably high. A less costly approach, which diminishes the flexibility of processor utilization, is to establish distinct security regions within the processing resources of a multiprocessor, and to employ special functions to protect the communication between regions. This approach still has problems if information sharing is nontrivial.

Requirements for secure networks are found in *Trusted Network Interpretation*,[5] the so-called "Red Book" that relates and extends the "Orange Book" to networks.

[5] *Trusted Network Interpretation*, NSC-TG-005 version -1, National Computer Security Center, July, 1985.

Chapter 6

A System Evaluation Laboratory

This chapter will describe the general features and components of a system evaluation laboratory. The first section discusses general objectives. The following sections discuss general capabilities and specific tools.

6.1 General Objectives

The requirements, designs, and evaluation criteria for BM/C3 computers are far too complex to be served by a collection of special-purpose testing tools. System evaluators will wish to view the subject system from many viewpoints, at different levels of abstraction, and operating under many different circumstances. Each viewpoint may require construction and validation of specific models. In order to evaluate some capabilities, e.g., response to overload or recovery from complex fault conditions, numerous interacting processes will have to be synchronized and observed in great detail. These processes may reside in different, specialized test computers, whose configuration must be capable of flexible tailoring to fit particular experiments.

It is clear from these examples that evaluation must be a creative and highly mechanized activity, in which :

- Tools are specially constructed or adapted to serve particular needs.
- Model construction and validation are supported as a unified activity.

- Complex evaluation experiments are treated as formal processes, with designs, plans and high-level controls.
- Tests are designed modularly, so that they may be reused for a variety of system types and application contexts.
- Test designs and results are documented, so that they may be repeated by others.

The following sections will describe the components of a system evaluation laboratory that is intended to provide an integrated work environment for evaluation activities. The proposal is a substantial extension of current activities and trends in performance analysis, computer aided design, and computer aided software engineering.[1]

6.2 An Environment for Work Support and Tool Integration

A suitable evaluation laboratory must provide a convenient environment for evaluation work and powerful, flexible means for integrating the range of evaluation tools. This section discusses requirements for general-purpose computing facilities, software development tools, a database system, and tools for supporting experimentation.

6.2.1 Computing Facilities

Designing and conducting evaluation experiments will require several kinds of computer support, including workstations, stand-alone general-purpose computers, and hybrid configurations of test generators, simulators, analyzers, etc. The complexity of system evaluation places a premium on the ability of evaluators to prototype designs for evaluation experiments, and to modify those designs rapidly. It is therefore essential to provide the highest feasible computing power and flexibility in the computing facility.

An obvious way to meet these requirements is to organize the set of computing facilities as a local-area network, in which processes in different

[1]See, for example, Z. Segal and L. Rudolph, "PIE: A Programming and Instrumentation Environment for Parallel Processing," *IEEE Software*, (November 1985), B. Melamed and R.J.T. Morris, "Visual Simulation: The Performance Analysis Workstation," *IEEE Computer* (Aug. 1985), and "The Performance Analyst's Workbench System," Information Research Associates, Austin, Texas, 1985.

node computers may be combined with great flexibility. Another possibility might be to employ a general-purpose multiprocessor as both the machine integration environment and special test processor, with direct processor interfaces to specialized test machines as needed.

Workstations

Evaluations will be designed using modern workstations, i.e., powerful personal computers as stand-alone environments or with network access to remote stations, databases, and high-powered computing engines. The workstation computers will support flexible software design and documentation, and provide appreciable power for executing small evaluation models. They will also provide control for the execution of large models on the laboratory's more powerful computers. A later section will present the concept of *evaluation scripts*, which are high-level programs for complex evaluation experiments. Design and execution of evaluation scripts will be a major use of the workstations.

Autonomous Evaluation Engines

Large evaluation models in which the target system is simulated should be executed using high-powered, general-purpose computing engines, under the control of a workstation. The use of modern architecture in the engine, such as high-order multiprocessors, should be transparent to the evaluation of the subject computer architecture.

Coarse-granularity multiprocessors are attractive because of their good cost-performance qualities, but fine-granularity multiprocessors, such as the Connection Machine, deserve consideration for their speed and programming convenience in the simulation of highly parallel systems.

Hybrid Simulation Configurations

Configurations of multiple computers have several uses in evaluation experiments. A major example is a combination of the subject computer and a general-purpose computer in which the general-purpose computer simulates the BM/C3 environment. A second example is the use of a general-purpose computer (or network of computers), in conjunction with a subsection of the subject computer, to simulate undeveloped or unavailable portions of the subject computer.

6.2.2 Software Development Tools

To support a wide variety of evaluation views and to provide for efficient analysis of unique system designs, the building and modification of software development tools will be a major activity in the evaluation laboratory. The tools must cover the full range of modern computing technology. It is therefore essential to provide a general-purpose software development environment that will support the treatment of program objects at all levels of computing detail, from gate and bit level upward. The basic programming language for tool design must support expert-level programming, but the tools should have interfaces suitable for professional and nonprofessional users. The environment must also support the programming languages that are employed at the application or system levels of the subject computers.

Tool development should be supported by a software development data base. The database will maintain and integrate module versions for individual tools and will hold libraries of tools and test data.

Tool development must employ modern software practices, e.g., modularity, hierarchy, information hiding, and well-specified and documented interfaces. There is a great amount of current technology in software development environments that can be brought to bear on evaluation tool development.[2]

6.2.3 Experiment Support Tools

As mentioned above, evaluations will be structured into sets of well-defined, interdependent processes, called *Evaluation Tasks*. For example, an evaluation for a given system will be composed of a set of tasks covering different parts and aspects of a system. (1) Each evaluation may be performed for a range of environments and structural assumptions. (2) A subject system may have a set of evaluations having different degrees of abstraction. The same environment and application may be applied to several different subject systems or several versions of a subject system. (4) The results of experiments must be processed by different data analysis tools, and so on.

Due to the complexity of BM/C3 environments and computer systems, much of the experimentation will have to be exploratory in nature. That is, the space of possible conditions must be probed initially to find the most

[2]See especially Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Palo Alto, California, December 9–11, 1986. P. Henderson, ed., *ACM Sigplan Notices*, Vol. 22, No. 1 (Jan. 1987).

critical points of behavior. When they are found, deeper analysis may be conducted to develop precise quantitative results. This initial probing will require rapid definition, execution, and analysis of experiments.

To support this dual mode of experimentation, the work environment must provide integrated support for the following services:

Multitask Planning. Software for planning is widely available, but integration with other services will not be trivial.

Task Design. The requirements for task design are substantial. Design of a task requires specification of all evaluation domains, e.g., input data, system models, parameter values, observations and data recording. In order to simplify the coverage of a range of operating and design conditions, individual specifications will be derived from general domain models, which will be instantiated for particular experiments. Control of the mass of details involved in task design will require a high-level, integrated user interface.

Task Execution. Task execution is also a substantial function. The workstation should provide the evaluator with a high-level command interface with experiments and a window for observing events at any level of detail. Commands will provide for linking, scheduling, and synchronizing data in several physical and logical sources. In complex machine configurations (e.g., with a subject computer embedded in a network of simulation computers), task control will be distributed over several computers. Error conditions must be detected, diagnosed, and reported, and experiments must be correctly suspended and resumed. Special data conditions that have significance for an experiment should be detected (e.g., constancy or instability of output values). The use of Unix-like piping for linking elements of a task would be effective.

Data Analysis. A wide variety of data analysis tools will be needed. These include standard statistical tools and special-purpose data reduction tools such as event timers and correlators.

Documentation. Documentation is a vital function that can be vitiated by the need to comprehend enormous masses of data. The work environment must provide a high level of automation for documentation in order to ease the task of the author and reader. Automation aids should support

structuring (outlining and indexing) and graphic representation. Advanced forms of documentation, such as interactive text and dynamic illustrations, should be integrated when feasible.

Data Management. Data management is a crucial function. It will be implemented in the form of an evaluation database service, as follows.

6.2.4 Evaluation Database

The function of the evaluation database is to manage the very large and varied body of data objects required for and generated by evaluation tasks, such as system descriptions, mission and input data sets, output data, and analysis tools. Access to data needed for designing a task must be transparent; that is, the designer will specify the elements, versions and order of the objects required for an evaluation process without specifying their physical location, and the database will retrieve and link the appropriate data components. In executing an evaluation task, the database will store the input and output data sets together with full information about the subject configuration, so that evaluations may be readily reproduced or extended.

A relational model appears to be natural for these purposes. The functions of version management, linking of models, tools and data sets, etc., will require a substantial system interface.

6.3 Evaluation Tools

This section discusses some of the most immediately important kinds of evaluation tools. The general orientation is toward system behavior. (Other evaluation goals, such as proof of correctness of design, cost of software development, and potential for realizing a design in different technologies, are left for further study.)

Although the tools are discussed individually, it is crucially important that they are closely integrated so that the evaluator may flexibly change levels of abstraction and introduce new dimensions of operation and evaluation without having to redefine the system for each new view. For example, tools for modeling, simulation, and measurement should share a high-level programming language, and it should be possible to correlate input data and performance observations. Some helpful approaches are the use of hierarchy in tool design, the establishment of standard data structures, and the choice of operating environments, such as Unix, that encourage tool linkage.

The list of tool types follows the general domain model of evaluation presented in this report.

6.3.1 Evaluation Planning

The highest goal of evaluating system performance is to determine relationships among performance properties in a region of operation defined by a set of domains. One relationship, for example, is the trade-off in a particular architecture between the rate of battle management decisions (e.g., assignments of weapons to targets) and the number of processors, for a specified level of fault stress. To determine that relationship, a set of predictions or measurements must be performed for a large number of points in the appropriate domains. As an example, the flow of battle management decisions might be approximated as a Poisson process and the level of fault stress may be represented as a single, general stress variable, representing a wide range of fault possibilities.

Given the complexity of the applications and the subject systems, the result of an evaluation should be *a hierarchical description of capabilities*, i.e., a set of descriptions at different levels of detail.

The evaluation laboratory should provide a tool for planning an efficient sequence of evaluation actions in the specified domains. Generally, this will consist of an iteration of model definition and model validation using either simulations or an actual version of the subject system. The evaluator will specify the domains of interest and select models in each domain for generating specific operating values. For example, there might be a program model in the application problem domain for generating target flows of various sizes and distributions. The planning tool will be used to generate sets of combinations of specific operating values in the several domains. In practice, domains may be explored extensively separately, and then in combination. Planning is also required for observing and recording system response, including the data to be observed and the rules of observation, e.g., sampling rate and precision.

After a set of operating values has been selected at the domain level, substantial work may be required to create actual test values. For example, a test configuration may consist of a real subject computer and a general-purpose computer that simulates the battle environment. In this configuration, fault conditions may be inserted into the subject computer by a special processor, which must be synchronized with software tests in the environment and subject machines.

The output of the planning tool will be a set of programs and data whose execution will exercise the desired set of evaluation conditions. A detailed example of a planning tool for measurement exercises, called Measurement Scripts, is given in a later section.

6.3.2 Mission and Environment Modeling

Modeling the mission and environment domains is essential to the relevance of computer system evaluation for BM/C3 applications. The laboratory must provide tools specifying battle input data, maintenance and control regimes, and environmental stress. The goal is not to validate sufficiency of a design for a specified BM/C3 service, but to generate realistic test data and stress conditions for evaluating the merits of a proposed computer architecture. (Details of test generation are discussed in a later section on workload generators.)

The complexity of these domains will require a hierarchy of models of different kinds, as in the following examples. The tools will be used to construct and exercise the models, and to link the models to those of other selected evaluation domains.

Battle and Maintenance Domain Scenarios

A finite-state model may be appropriate for describing various battle management service modes, e.g., readiness, alert, initial battle, damage recovery, and battle assessment. Battle descriptions would cover different assumptions about major profiles of attack and defense activity, sufficient to generate profiles of data flow. The same or an adjacent model should include remote maintenance and control modes. The model would define probabilities and durations of mode transitions and probability distributions for battle events. Battle descriptions should address the assignment of roles and loads to nodes in a distributed battle management system.

Another set of models will give more detailed descriptions of these scenarios. For example, battles will be described in terms of physical events e.g., patterns and rates of target flow, command decisions and information flow among battle management stations. Maintenance and control will be described in terms of information exchange and impact on the integrity of the subject system. Several kinds of model may be appropriate, including equations, tables, and simulation programs. Descriptions should be well documented and employ graphic representation to aid in validating the model.

Environmental Stress

The term *environment* is used here to cover all operational conditions that may stress the system. Such stresses are rather heterogeneous in nature, and might seemingly be best distributed to other domains, e.g., the data domain or hardware logic domain. They include severe problem overloads, loss of external communication, and faults, both internally and environmentally induced, and affecting both the physical system and its design. As a matter of choice, they may be represented in other domains, but they are brought together in a single domain, redundantly, in order to help the evaluator distinguish all those factors, alone and in combination, that may degrade system performance. A major benefit of this unification is that it should simplify descriptions of coincident, multitype, stress conditions.

Problem overloads and losses of external communication may be represented as time profiles of activity. Such profiles may be produced by Markov models or by parameterized time functions. Particular interest may lie in how fast the subject system can recover from overloads of different peak sizes and durations.

Externally induced system faults include radiation and kine-tic damage. These may cause multiple localized faults among system components, both permanent and temporary, or major loss of resources.

Internal physical faults may have different assumed probability distributions, e.g., constant rate or age-related rates, and their manifestations in component failures and erroneous data will be highly variable in the dimensions of (1) time (permanent, transient, intermittent), (2) multiplicity (single events, bursts, and multiple distributed events of the same or different fault modes), and (3) system level of the resulting error or failure (gate/bit, module/object, processor/process).

Tools must be available to support the definition of fault-occurrence models and failure and error models in particular system components. Markov models are very appropriate for fault occurrences. Identification of where faults and errors are located in the subject system will require special models of fault-error chains from module to module and level to level within the subject system. Detailed descriptions of fault modes would be best included in the system models, which are in a separate domain. Fault-stress models should therefore contain a high-level abstraction, so that they can be readily combined with other stress types.

6.3.3 Requirement and Specification Definition

The domain of requirements and specifications allows the evaluator to define objective standards for system evaluation. Of interest will be not only the standard user-oriented criteria of performance, reliability, availability, etc., but also some criteria that do not have standard definitions, such as autonomy (expected length of time in which the system may have to operate without maintenance), programmability, expandability, and recoverability. Also of great interest are specifications of the trade-offs among criteria that may be demanded by users, e.g. for performability (allowed degradation of performance with fault rate), peak performance vs. expected lifetime, or peak performance vs. autonomy.

The standard performance and reliability criteria must be carefully applied to fit the character of the intended service. For example, performance will inevitably differ for different classes of computation, and reliability requirements will be a function of mission phase and remote maintenance service.

In this view, specifications are definitions of contours in a multidimensional space of evaluation criteria. A useful tool for specifications will help to:

- Define the formal relationships, metrics, and scales for individual requirements.
- Extract and display views of multidimensional requirements in selected coordinate sets.
- Extract and display trade-off relationships among selected requirements.

6.3.4 System Description

System evaluation will require both detailed and abstract descriptions of the system under study. This section discusses the need for detailed descriptions; abstract system modeling is discussed in the following section.

Tools in the system description domain must cover the full range of hardware and software objects in subject systems at all levels of detail, and with a variety of viewpoints, e.g., control, data flow, synchronization, and fault mode. Classes of subsystems will include:

- System architecture

- Logic, memory, and communication functions
- Subsystem functions
- Parallel processing structure and logic
- Fault isolation, diagnosis, and reconfiguration.

- Operating system
  - Interface functions
  - Distributed control structure
  - Interprocess communication control
  - Resource management
  - Error detection, recovery, and reconfiguration.

- Applications
  - Algorithms
  - Data structures and database management systems
  - Parallel processing constraints.

Examples of the tactical issues that arise in system description are:

- Which languages (and how many) to use for system description
- How to treat parallelism.

Which System Description Languages?

There are several reasonable viewpoints about which system description languages, and how many, should be employed to describe this large range of objects. From one viewpoint, it would be advantageous to represent all objects by a single language, such as VHDL, Ada, or one of the new object-oriented languages. This would have the benefits of generality in performing simulations, simplicity in studying processes independently of hardware or software implementation, and economy in tool maintenance.

From another viewpoint, use of a single, contemporary language has the disadvantage of placing a burden of syntactic complexity on some analyses that could be performed with much more compact notation; for example,

studies of interprocess communication or fault diagnosis could be better supported by languages that abstract away irrelevant processing details.

Ideally, the environment should allow both a general-purpose language for complete representation of data objects and special languages that simplify particular analyses.

Representation of Parallelism

Current treatments of parallelism give reasonable descriptions of parallel behavior only for highly structured processes, such as pipelines and single-instruction, multiple-data (SIMD) computing. Those descriptions allow estimation of the order of effective parallelism and the maximum time required to complete a computation. Graph models are useful for describing small, fine-grain, multiple-instruction, multiple-data (MIMD) programs, but the general case of coarse-grained MIMD processing lacks good descriptive formalism.

New representations are needed for describing important modes of parallel-processing behavior, such as:

- Dynamic process allocation, in which a computation may consist of a varying number of processes.
- Hot spots, in which some functions are consistently responsible for significant delays.
- Contention for resources among several unrelated sets of multiple processes.
- Coordinated sets of parallel operations within arrays, e.g., image processing and database search.
- Selective error recovery, where some processes in a multiprocess set operate in a recovery mode while others may move forward.

6.3.5 Behavioral Models

Creating, validating and exercising models for system behavior are central activities in an evaluation laboratory. The following are several widely used model types. Considerable literature exists on how these models may be used in single-domains (performance or reliability). Particularly needed are models that describe ¡¡¡multidomain¿¿¿ behavior, e.g., performance during

error recovery, or load balancing under transient overloads. Stochastic activity networks have been applied for this purpose, but more work is needed.

Analytic Probability Models

A very widely employed method of modeling computer performance is to construct polynomials in which the variables are the amounts of time taken for operations at a chosen level of detail and the coefficients are probabilities representing the frequency of those operations. In some cases, e.g., well-designed cache memory systems or bus communication systems, the times may be known accurately, and some frequencies are small enough that their variations are insignificant. Useful values of frequency parameters may be derived from experimental or simulation analyses of subsystems. In cases where the times and frequencies have significant variations, upper and lower bounds may be calculated using judicious assignment of maximum and minimum time and frequency values. Processing times may be known precisely, but good estimates of branching frequencies will require experimentation using real or simulated processes. In more complex cases, other models must be used.

Queueing Models

Queueing models are useful for studying performance at all levels of a system above the logic level.[3] They are widely used to model interconnection systems, memory hierarchies, multiple processors, and entire computer systems. ***Analytic queuing models*** offer either closed form solutions or sets of equations that can be evaluated at relatively low cost. Although they are intuitively appealing, a serious limitation of analytic queueing models is that convenient solutions are available primarily for simple systems and only for a limited range of arrival-distribution functions. The results may be quite different from those for real arrival distributions, and as a result, queueing models are viewed by many with some skepticism. Simulations of queuing systems using *discrete event simulation programs*, in which more realistic arrival distributions may be employed, allow more accurate modeling, at a speed reduction of from two to three orders of magnitude.

[3] C.H. Sauer, E.A. MacNair and S. Silva, "A Language for Extended Queueing Network Models," *IBM Jrnl. of R&D*, Vol. 24, No. 6 (Nov. 1980).

Activity Graph Models

Activity graph models, also called *marked graph models*, are a class of mathematical systems based on directed graphs that exhibit dynamic behavior. These models do not process data values; rather they are used to describe control behavior, e.g., waiting, deadlock, and frequency of process activation.

In an activity graph, nodes (places) of the graph are assigned to individual processes of a computer, and unidirectional branches (arcs) correspond to the communication of process-completion information between processes. A node may experience a transition (firing), corresponding to the activation of a computing process. Markers (tokens) are assigned to those branches whose source node has experienced a transition. Dynamic behavior is modeled by changing the assignment of the markers to the branches (the change is called a Transition). Different types of activity graphs result from different rules for changing marker assignments, as in the following examples:

Petri Networks. When all in-going branches of a place in a Petri network have markers, the place makes a transition in which all in-markers are erased and all out-going branches are marked.[4] Since a branch that never receives a mark will never allow its in-node to fire, Petri nets are useful for modeling deadlock and starvation phenomena in concurrent processing. The basic Petri net model describes only the order of events and not their times. The assignment of markers to all branches of an out-node implies full concurrency of the corresponding processes, which may not be fully realistic. Petri net models have been used at many system levels, from interconnection networks to distributed computer systems. Several versions of the model have been studied, e.g., firing may occur at an n-input node when at least k of the n branches are marked.

Timed Petri Networks. In a timed Petri network, time values are associated with each node of a Petri net to represent the time required by the associated computation.[5] Some versions also associate a predicate that permits a transition to occur only under a specified condition. This allows

[4] J. S. Peterson, "Petri Nets," *ACM Computing Surveys*, Vol. 9, No. 3 (Sept. 1977).

[5] M.A. Marsan, G. Conte, and G. Balbo, "A Class of Generalized Petri Nets for the Performance Evaluation of Multiprocessor Systems," *ACM Tr. Computer Systems*, Vol. 2, No. 2 (May 1984).

more detailed and realistic modeling of data dependencies in parallel computations, as well as estimations of processing time. Other logical extensions to activity networks include queues and switches, whose behavior may be governed by time distributions and probabilities.

Stochastic Activity Network (SAN). In stochastic activity networks, activities (transitions) at places may be either instantaneous or timed and may be subject to enabling predicates.[6] Each activity may be followed by a set of actions whose initiations are regulated by case variables, an abstraction of program branching and process initiation. The dynamic behavior of multiprocessing systems is modeled by assigning time-distribution functions to timed activities and probability functions to the case variables. The latter functions offer a convenient way to define degraded resource sets in performability studies.

Early activity graph models were limited in the size of the system that could be conveniently modeled and in the variety of control behavior that could be defined. Recent models, e.g. timed Petri nets and stochastic activity nets, foster ***abstraction***, which allows orderly treatment of complex systems, and ***logical qualification***, which permits greater variety in control behavior. These qualities help to bridge the gap between simple control models and the much more complex (and more expensive to execute) simulation models.[7]

A significant limitation of activity graphs is their poverty in supporting realistic models of input data. One standard practice in activity nets is to assign an initial marking to the graph that represents input data, and allow the graph to process that marker set. Another is to feed markers to a node, representing arrival of new input data. Again, the variety and range of distribution of this model of input data are quite limited.

Markov Models

A Markov model is a stochastic model consisting of a set of states and a set of state transitions.[8] A state contains no information on the sequence of state

[6] W.H. Sanders and J.F. Meyer, "METASAN: A performability evaluation tool based on stochastic activity networks," *Proc. IEEE-ACM 1986 Fall Joint Computer Conference*, Dallas, Texas, Nov. 1986.

[7] E.g., *The Architecture Design and Assessment System (ADAS)*, Research Triangle Institute, Research Triangle Park, North Carolina.

[8] K.S. Trivedi, *Probability and Statistics with Reliability, Queueing, and Computer Science Applications*, (Prentice Hall, 1982).

transitions that preceded arrival at the state; in other words, the present state summarizes the entire past history of the system. Dynamic behavior of a Markov model is summarized in a transition matrix, which defines the condition for transition between every pair of states. A major attraction of Markov models is that the transition matrix may be used in a computationally efficient way to determine steady-state behavior. Versions of Markov models exist for describing both discrete and continuous processes. In the former case, state transitions are summarized in a matrix of probabilities In continuous models, the transition matrix is composed of transition rates derived from a variety of distributions (exponential, Weibull, etc). Nonhomogeneous Markov models are a class of models that can be used to describe transient or other time-varying behavior.

Markov models have found widespread use in modeling computer and communication systems, as well as other physical processes. Examples include (1) fault-tolerant systems, in which states may represent different levels of non-failed resources or special error recovery modes,[9] (2) maintenance cycles, in which a system may switch among modes such as operation, test, and repair, (3) communication channels in which errors may occur in bursts, and (4) multiprocessor systems, in which states represent the number of processors in a particular state (e.g., awaiting data).

The convenience of Markov models has encouraged development of many tools that support particular applications and behaviors.[10]

Simulation Models

In contrast with probabilistic, queueing, and activity graph models, all of which can only describe control behavior, simulation models are capable of representing arbitrary computational behavior at any level of abstraction, from algorithms to devices. This versatility has several important benefits:

- *Data sensitivity.* The system may be driven by arbitrary input data, allowing greater realism in performance estimation.
- *Mixed abstractions.* Different levels of abstraction may be employed for sections of a simulation model, allowing a selective focusing on parts

[9]For a representative list of tools, existing and under development, see "Recent Publication List, Center for Computer Systems Analysis, Duke University," in *ACM Sigmetrics Performance Analysis Review,* Vol. 14, No. 3 (January 1987).

[10]C.A. Liceaga and D.P. Siewiorek, "Towards Automatic Markov Reliability Modeling of Computer Architectures," *NASA Technical Memorandum 89009* (Aug. 1986).

of the system under examination. This allows the chosen functions to be simulated with great detail and high execution cost, while the remainder of the system may be simulated with coarser detail and low execution cost.

- *Convenient definition*. A uniform programming representation makes it easy to define variations in a design at any level of the design hierarchy. This includes variations not only in function but also in the balance of hardware and software implementation.

- *Non-intrusive instrumentation*. Since simulated time is completely under control of the simulation program, it is possible to insert instrumentation at arbitrary points that will measure a process without disturbing it. This may be useful in evaluating the amount of disturbance caused by a real instrument.

- *Arbitrary fault injection*. The response of a design to complex fault and error conditions may be studied without risking possible damage to real hardware. Faults may be inserted at a higher rate than in real fault-injection experiments.

These benefits may be very costly compared to analytic or control-centered models, whose results generally apply to average or steady-state values over a range of input conditions. In contrast, simulations usually yield exact results for specific inputs, and require repeated application to cover a range of input conditions. The execution time of simulation models can be costly, especially if the chosen level of detail is high, because many executions may be needed to attain the desired confidence in the results over a range of input data.

Simulation has been sufficiently attractive to have stimulated the development of a large number of tools, from general-purpose discrete-event simulators[11] to simulation languages that support particular classes of architectures.[12] Full realization of the potential of this powerful technique requires a good environment for developing simulation programs.

The high computational cost of simulation makes the use of high-power simulation engines very attractive. Use of engines in the gigaflop range would

[11] J. Misra, "Distributed Discrete-Event Simulation," *ACM Computing Surveys*, Vol. 18, No. 1 (March 1986).

[12] M.D. Ercegovac, Multiprocessor System Evaluation and Programming Environment, UCLA Computer Science Dept. report, April 1986.

surely allow a more rapid exploration of an evaluation space and more rapid achievement of accurate simulation results than would be available from conventional machines, with savings in labor that may be comparable to the incremental machine cost. Such engines are increasingly available by remote network access.

Integrated Models

Simulation models allow the evaluator to view a system at different levels of abstraction, but at a high computational cost relative to the graph-based models. Several researchers have investigated models that allow a blend of graph and simulation modeling.[13] This trend to integrated models should be encouraged.

6.3.6 System Measurement

As discussed in the section on evaluation objectives, the purpose of measuring the subject system is to validate or calibrate a general model of the system that will predict its capabilities over a range of possible requirements. Considering the high cost of actual measurements compared to model evaluations, measurement experiments should be designed to obtain information that cannot be obtained from available models.

It also was noted that while the ideal evaluation approach is to consider all domain factors simultaneously (e.g., performance of a system near the end of its lifetime, under overload, with concurrent error recovery and fault recovery), for practical reasons it may be necessary to develop the system model in stages, first with separate domains and then with selective combinations of domain conditions.

This section presents the concept of measurement scripts, a technique for organizing complex system measurements. It then discusses particular tools for performance and performability measurement, fault injection, workload generation, monitoring, analysis, and benchmarking.

[13] G. Estrin, R.S. Fenchel, R.R. Razouk, and M.K. Vernon, "SARA (System ARchitects Apprentice): Modeling, Analysis, and Simulation Support for Design of Concurrent Systems," *IEEE Tr. on Software Engineering*, Vol. SE-12, No. 2 (Feb. 1986) and J.C. Browne, D. Neuse, and J. Anderson, "Top-Down Design and Simulation Modeling of Electronic Systems: PAWES," Dept. of Computer Science, University of Texas at Austin, Texas, 1987.

Performance Measurement

Computer performance measurement produces estimates of computing power through a process of stimulus, response, and analysis of a computer system operating within defined constraints. It typically involves a large number of individual tests and a mass of data, and hence is used selectively to study behavior that is too difficult to model analytically. As with all experimental observations, it must be concerned with how to separate the effects of the different underlying phenomena and with minimizing the impact of observation noise. Classical performance measurement assumes a fault-free system. In BM/C3 evaluations, performance measurement must consider the effects of faults and other stresses.

Some special problems of performance measurement for BM/C3 computing are:

- Measurement of parallel processes without stress, to determine basic machine capabilities.
- Measurement of performance under various conditions of stress, e.g., faults and overloads.

In a fault-tolerant multiprocessor, both of these measurement objectives require scheduling, synchronizing, and observing a large number of processes that may have complex and transient interactions. As there are no general techniques for such measurement, it is necessary to design measurement functions for each phenomenon of interest. Measurement scripts, discussed in the following section, are intended to support such design.

Measurement Scripts.

The complexity of multidomain measurements for advanced architectures requires strong control of the many measurements that will be needed to derive meaningful results. A key function of a measurement tool is thus to support the construction of a measurement plan, which for convenience will be called a *measurement script*. In a measurement exercise, a script will be executed repeatedly using a range of data, and various states of interest in the subject system will be observed and recorded. These recordings are called *traces*, and they constitute the raw material for performance data analysis. A script will contain a set of:

- Test scenarios

- Test program segments
- Test data sets

  item Scaling
- Initialization values
- Observables.
- Measurement Configurations

Requirements for these components of a measurement script are discussed in the following paragraphs.

* **Test scenarios.** A measurement script will contain a set of test scenarios that are high-level programs to control the sequencing of program segments and the application of data sets for both normal and stressed operations. In simple tests, only one performance phenomenon will be observed. In complex tests, where several phenomena are being studied or when a stress condition is simulated e.g., a significant overload of input data, the scenario must define and synchronize several event sequences in order to implement searches for worst-case conditions. Scenarios for multiprocessing phenomena may be complex indeed. For example, several multiple-process computations may co-exist within a multiple-processor configuration, and it may be of interest to study the interaction of these computations, e.g., in competition for resources. A valuable feature would be to make the control of the timing of interacting processes depend on data conditions that arise in the measurement; for example, overload conditions may be timed to start when interprocess traffic is at a peak.

* **Program segments.** A script will specify a set of program segments selected from a set of programs that model the application domain or that test particular architectural features of the subject computer. The data sets represent the space of input data for the intended operations, both normal and stressed. The initialization values define the data environment for the performance measurement; they may range from simple settings of program variables to complex arrays of data. The configuration definitions define the machine environment. Variations may represent changes in scale, e.g., numbers of processors or memory sizes, or fault-induced degradations.

* **Test data sets.** A script specifies test data selected from the intended input domain. As discussed below under Workload Generators, data will be generated by special programs. These programs should be structured hierarchically, with high-level functions that allow succinct control of how the test space is to be explored with respect to actual data values and rates of occurrence.

* **Initialization values.** Initialization values define the data context for the test programs. They may range from simple settings of program variables to extensive arrays of data.

* **Scaling.** Scripts will specify the scale of system elements in several dimensions, including changes in scale of resources (number of processors, sizes of memories, version of the operating system, etc.) and fault-induced reductions or corruptions.

* **Observables.** Scripts will specify the observations to be made during the course of the test. Dimensions include the points of observation within the subject system, particular data to be recognized, and possible logical conditions on the enablement of particular observations. The trace records of such observations must be identified so that multiple observations may be properly combined for presentation to the analysis tools.

* **Measurement Configurations.** While some measurements may be conducted solely within the subject system, others may require configurations of several general or special purpose computers. Examples of other components that may be used individually or in combination in such configurations are:

  - Fault-injection and monitoring machines
  - Environment simulators
  - Simulators of parts of the subject system design, such as parts of the system that are not available for testing and system processes that are not conveniently observable.

The measurement script must therefore include definition and control of a multiple-machine measurement environment. In such environments, some of the multiple test and observation processes to be synchronized will reside in different machines.

* **Virtual-Machine Definition of the Measurement Configuration.** An appropriate approach to managing the complexities of multiple measurement processes in a multiple processor configuration is to employ some abstraction in the definition of measurement processes. Defining the measurement configuration as a virtual machine would make it easy to modify hardware and software components within a general-purpose, distributed, measurement environment.[14] A virtual machine definition could provide a basis for a general-purpose, distributed, measurement environment.[15]

Fault Injection.

System faults and responses constitute a great challenge to system measurement because of their great variety and complexity. Faults may be permanent or transient, single or multiple, and correlated or non-correlated in time and location. Non-fault tolerant responses may be extremely varied, and fault-tolerant responses may be incomplete. Key issues for fault injection studies are:

- The extent of propagation of errors within and across fault-tolerance boundaries
- Coverage of faults and errors achieved by fault-tolerance functions
- Fault latency (delay in discovering a fault)
- Recovery time and completeness
- Degradation in system resources due to fault elimination
- Permanent and transient impact of faults and fault tolerance functions on system performance.

The last issue in this list is a very complex matter in that it requires coordination of both fault tolerance and performance measurements.

The complexity of fault characteristics and responses raises several technique issues, including hierarchical fault injection and observation, real and

[14]For example, see F. Gregoretti and Z. Segall, "Programming for Observability Support in a Parallel Programming Environment," Carnegie-Mellon University, Computer Science Department Report, 1985.

[15]E.g., Z. Segall, et al., "FIAT-FITB: Fault Injection Based Automated Testing Environment Fault Injection Test Bed," Carnegie-Mellon University, Computer Science Department Report, 1987.

simulated fault and error injection, and treatment of transient and multiple faults. Mastery of the complexities of fault measurement will require a unified, general-purpose test environment.[16]

* **Hierarchical fault injection and observation.** Many details of a system's response to faults may be obtained from analysis or simulation of a design, but some details may require that faults be inserted and observed in the real system. Both the faults inserted and the observation of errors and fault/error responses should be hierarchical, for several reasons. First, while it is true that faults occur at low system levels (circuit or logic levels for physical faults and instruction level for software faults), most of the fault-tolerant responses occur at higher system levels (bit-error correction is an exception). Second, in a multilevel system, fault-tolerant responses will be found at many levels, and it will be much simpler to study high-level responses by injecting faults at adjacent levels rather than at primary levels.

* **Real and simulated faults and errors.** Capabilities are needed for injecting both real and simulated faults and errors. Errors injected at the pins of a hardware module may serve to simulate faults at the outputs of a module or they may represent the errors that would result from faults within the module. Injecting such errors using a special signal generator may be an economical way to generate a large number of logic-level fault conditions. As technology advances, this may become a less satisfactory way of simulating logic faults, because faults in large modules may give rise to very complex behavior at the pin level, e.g., lengthy multiple-pin error sequences. High-level fault and error tolerance functions will be most easily studied by injecting error conditions at adjacent levels that simulate lower level faults. Such injection is readily done by planting special functions in the software of the subject system.

* **Transient and multiple faults.** There is little certain knowledge about the rate and character of transient and multiple faults in computers, but the consensus is that transient faults occur at a rate an order of magnitude greater than that of permanent faults. Operation in a high radiation environment may make multiple faults a significant factor in system reliability, but this possibility has not been quantified. The high dependence of these phenomena on device technology, packaging, and mission characteristics may

[16] As in Segall et al., "FIAT, etc.," ibid.

prevent much further improvement in knowledge. This uncertainty makes transient and multiple fault testing a vital part of system measurement.

Transient testing should include one-time or intermittent fault events, and families of correlated transient faults. Such behavior may be achieved by making time, frequency, and logical predicates (to allow synchronization of faults and normal system processes) be parameters in the error injection function.

Workload Generators

Workload generators are programs that synthesize data for application to the computer under evaluation. As discussed in earlier sections, test data may be based on the application or on the internal characteristics of the subject computer.

An application-based workload generator is a program whose output simulates the data that will be presented to a computer by its actual environment, i.e., it is a computational model of the environment. A good application generator will be heavily parameterized so that a wide range of test conditions may be produced conveniently. It should be hierarchically structured, so that high-level concepts that are close to the physical problem may be defined and used to drive the lower-level data production. In terms of the domain model of evaluation, workload generation spans both the Application Problem and Application Data domains. The first models the application in terms that are meaningful for physical events, information flow, and decision actions, while the second describes detailed data structures and sequences. Separation of these domains in workload generation will help to reduce complexity and hence to increase flexibility and reduce errors in design.

A good architecture-based workload generator will generate test data that exposes the strengths and weaknesses of the subject computer. It thus occupies the system description domain as discussed previously. A powerful workload generator will be directly based on the design of the subject system, and thus will be a model of critical design features of the computer or of the general class to which it belongs.

A difficult challenge in workload generator design for BM/C3 evaluations is how to minimize the variety of test cases that will adequately cover the range of evaluation criteria. Variations in degree of parallelism, time phasing of successive waves of parallel processing, complexity of individual computations, etc., create a very large number of possibly important

processing situations. It is conceivable that some situation that covers two evaluation domains, e.g., the occurrence of a fault just at the moment of a critical decision, may require a unique test input. The workload designer must assure the evaluator that such multidomain events will be adequately exposed.

Monitors and Analyzers

The discussion of evaluation planning emphasized the importance of *hierarchical evaluations*, i.e., evaluations of system capabilities at several levels of detail, e.g., device logic, instructions, processes, programs, applications. In measurement experiments, such descriptions are obtained by observing real processes, using monitoring functions. For example, while a test program exercises a computer system, a monitor will observe the occupancy of various resources in their several states, e.g., number of processors waiting or busy, channel data rate, or delay times in interprocessor communication.

In general, monitoring involves (1) performing an observation at some level of detail, and (2) generating a measurement report at the same or some higher level or levels of detail. For example:

- Observation of interprocess communication may involve performance reports of functions at the hardware and operating system levels.
- Effects of a fault at the device level may be reported at all higher levels.

Low-level monitor reports will contain a vast amount of detailed data. The monitor tools should aggregate such data into meaningful abstractions and propagate them to measurement reports at higher system levels. Monitoring of multiprocessors is complicated by the need not only to observe a single, dynamically changing process, but to correlate the activity of several related processes.[17]

A general problem in experimental observations is *the error introduced by the monitoring process.* A second problem is the degradation in the process observed, especially when measurements are performed sequentially with the process observed. These effects are most serious when the process observed and the observing process are comparable in size or time. For this reason,

[17]e.g., T. Kerola and H. Schwetman, "Monit: A Performance Monitoring Tool for Parallel and Pseudo-Parallel Programs," em ACM Performance Evaluation Review, Special Issue, Vol. 15, No. 1 (May 1987).

manufacturers do not incorporate in-line monitors at low levels of detail. Two strategies for overcoming this problem are (1) using special on-line hardware monitors that observe system processes with minimal intrusion,[18] and (2) calibrating in-line monitors at the same or higher levels of detail by executing nonintrusive simulation models (as discussed in the section on simulation).

Analysis of measurement data will require use of standard statistical techniques, special preprocessing functions for combining and transforming trace data into standard form, and special postprocessing functions for displaying analyses in convenient form. Standard statistical techniques are provided by several available tools,[19], and others may be developed as part of a monitor design.

It is very desirable to minimize the variety of forms for displaying results of analyses, so that it is possible to perceive relations among analyses in different domains.

Benchmark programs

In the absence of scientific methods for testing computers, a set of benchmark programs has evolved within the user community. This section discusses the use of existing standards and the prospect of developing standards for BM/C3 evaluations.

*** Standard Benchmark Programs.** Benchmark programs are based either on sample programs from important or broadly interesting applications, e.g., nuclear codes or standard mathematical functions, or on application-free tests of architectural features, e.g., floating point precision, loop execution or memory access rates. A set of standard benchmark programs has evolved in the user community, e.g., Whetstone, Livermore Loops, and is now available from several repositories.[20]

At their best, benchmarks help to compare different computers within an architectural class. At their worst, benchmarks may seriously mislead users about the productivity of a subject machine in actual usage.[21] The

[18]Alan Mink, "NBS Multiprocessor Measurement Experience," National Bureau of Standards, unpublished.

[19]E.g., the SAS, Inc., product.

[20]J.J. Dongarra and E. Grosse, "Distribution of Mathematical Software Via Electronic Mail," *Comm. ACM*, Vol. 30, No. 3 (May 1987).

[21]J. Dongarra, J.L. Martin, J. Worlton, "Computer benchmarking: paths and pitfalls,"

differences in performance arise from differences in the matches among features of algorithms, language constructs, operating systems, and machines. The value of community-standard benchmarks will be especially low for advanced multiprocessor architectures because of the many new dimensions of computer design, e.g., bussed vs. switched vs. networked communication, and fine-grained vs. coarse-grained processing. Each architectural approach will require special programming methods to take advantage of its unique qualities, and programs that work well on one class of multiprocessors may do very poorly on another class.

If these warnings are kept in mind, there still may be value in using standard benchmarks in multiprocessor evaluation. Functions such as matrix inversion, sorting, and searching are sufficiently universal that it will be useful to have a set of applications that can compare machines within an architectural class. For comparison of machines that employ different kinds of parallelism, different versions of these basic functions should be developed.

* **Benchmark Programs for BM/C3 Evaluations.** Proper evaluation of subject computers for BM/C3 applications will require testing of the computers for an wide range of program types. This range is not presently well defined, and it may be expected to evolve in time. Given the large number of properties of interest in evaluation (performance, error recovery, fault recovery, etc.) it would be very costly to test each of those properties for all programs within a BM/C3 application suite. It will therefore be very desirable to construct a small set of representative benchmark programs that can be used for the full range of evaluation tests. The benchmarks programs will have to be rewritten to take advantage of the particular forms of parallelism employed in different candidate computers. At this time, it is not clear how large the suite of derived benchmarks will have to be to adequately represent the BM/C3 application.

IEEE Spectrum, (July 1987) pp. 38-43.

Chapter 7

Conclusions

This report has presented an approach to methodology for evaluating high-dependability, high-performance multiprocessor computers for BM/C3 application. A general framework has been presented based on the concept of evaluation domains, which are formal definitions of all significant aspects of the application problem, the application environment, and the computer system under evaluation.

A central function of evaluation work will be the design and validation of system models that will allow (1) estimation of performance, reliability, and other concerns over a wide range of possible BM/C3 applications and environments, (2) study of interactions among performance and reliability objectives, and (3) comparison of different subject computer architectures. The domains selected as relevant to an evaluation objective are employed to construct an comprehensive and efficient set of evaluation experiments. The notion of performance stress, one of the possible evaluation domains, has been introduced as an abstraction of all the forces (including input overload and major fault conditions) that tend to produce significant departures from desired behavior.

Suggestions are given for a powerful, integrated evaluation laboratory. The laboratory should employ powerful computing resources (workstations and simulation engines, in a flexible local network) and software development aids for conducting evaluation experiments and for developing special tools. Evaluations are organized by Scripts, which are plans for test system configurations, complex scenarios of test data generation and data collection and analysis. The work environment should help the evaluator to take views of arbitrary system elements at different levels of abstraction, and to

assemble the views into a unified system representation.

References are given to some current work in evaluation. The rapid growth in the art of performance and fault tolerance evaluation in the past two years makes the vision presented here feasible with a reasonable amount of effort. Most existing tools are still rudimentary and exploratory, and much further development will be required to achieve robustness, breadth, and convenience. While the initial tools will probably be imported and adapted, achievement of substantial evaluation power will require an integrated suite of powerful tools and models.

More research is needed on theories and tools for evaluating parallel processing, performability, security, programmability, and fault tolerance in multiprocessors, and the interrelation of algorithmic and machine parallelism.

Bibliography

These references, together with those cited in the text of this report, are representative of a substantial and growing literature on performance and dependability evaluation.

1. *ACM Computing Surveys*

   Comprehensive surveys of research results, with frequent articles on performance and reliability.

2. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, August 1985, Austin, Texas. Digests in *ACM Performance Analysis Review*, Vol. 13, No. 3–4 (November 1985).

3. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, May 1987, Banff, Alberta. Digests in *ACM Performance Analysis Review*, Vol. 15, No. 1 (May 1987).

4. *ACM Transactions on Computer Systems.*

   Performance modeling is a major theme.

5. M. Chellappa and K.K. Somalwar, *An Annotated Bibliography for Integration of Performance and Reliability with Emphasis on Multiprocessor Systems.* Dept. of Computer Science, University of Texas, Austin, Texas (March 1987).

6. IBM *Systems Journal*, IBM Corporation, Armonk, New York.

   A quarterly journal with a wide range of articles on system development and evaluation.

7. *IEEE Computer.*

   Numerous articles on multiprocessor architecture and system evaluation.

8. *IEEE International Symposium on Fault-Tolerant Computing, Annual Digests.*

   The preeminent conference on design and evaluation of fault-tolerant systems.

9. *IEEE Software.*

   A monthly magazine on software methodology, techniques and tools.

10. *IEEE Transactions on Computers.*

    Substantial publications of well-reviewed papers in performance and fault tolerance, and occasional relevant special issues.

11. *IEEE Transactions on Computers,* Special Issue on Performance Evaluation of Multiple Processor Systems, Vol C-32, No. 1 (Jan. 1983).]

12. M. Malek and A. Matin, "Survey of Tools and Techniques for Performance Evaluation and Measurement." Dept. of Electrical and Computer Engineering, University of Texas, Austin, Texas, 1987, unpublished.

13. *Performance Evaluation.* Elsevier Science Publishers B.V. (North-Holland), The Netherlands.

    An international journal of theory.

14. "Performance Modeling and Measurement," Track MM–1, and "Fault-Tolerant Computing Evaluation," Track CD–1.2, *Proceedings, Fall Joint Computer Conference*, Nov. 2–6, 1986, Dallas, Texas, Computer Society Press.

15. *Proceedings, Performance Evaluation of Parallel Computers*, National Bureau of Standards Workshop, SBSIR 86-3395, Gaithersburg, Maryland (July 1986).

16. *Research Review*, 1986. Center for Supercomputing Research and Development, University of Illinois, Urbana, Illinois.

17. Z. Segall and L. Snyder, eds., *Proceedings, Workshop on Performance Efficient Parallel Programming.* Sponsored by the National Science Foundation and Carnegie Mellon University. Report 86-180, Dept. of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania (1986).

☆ U.S. Government Printing Office 1988—511-117/64104

END

DATE

FILMED

6-1988

DTIC