(4)

REPORT-SD-TR-88-109

Integrated Evaluation of Parallel Systems

C. F. KESSELMAN, M. M. GORLICK, and J. A. BANNISTER
Computer Science Laboratory
Laboratory Operations
The Aerospace Corporation
El Segundo, CA 90245

19 December 1988

Prepared for
SPACE DIVISION
AIR FORCE SYSTEMS COMMAND
Los Angeles Air Force Base
P.O. Box 92960
Los Angeles, CA 90009-2960

APPROVED FOR PUBLIC RELEASE;
DISTRIBUTION UNLIMITED

This report was submitted by The Aerospace Corporation, El Segundo, CA 90245, under Contract No. F04701-85-C-0086-P00019 with the Space Division, P.O. Box 92960, Los Angeles, CA 90009-2960. It was reviewed and approved for The Aerospace Corporation by A. J. Schiewe, Acting Principal Director, Computer Science Laboratory.

Lt C. A. Warack, SD/CNDA, was the Air Force project officer for the Mission-Oriented Investigation and Experimentation (MOIE) Program.

[illegible]
C. A. Warack, Lt, USAF
MOIE Project Officer
SD/CNDA

[illegible]
JAMES A. BERES, Lt Col, USAF
Director, AFSTC West Coast Office
AFSTC/WCO

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY 2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
4. PERFORMING ORGANIZATION REPORT NUMBER(S) TR-0088(3920-05)-3	5. MONITORING ORGANIZATION REPORT NUMBER(S) SD-TR- 88-109
6a. NAME OF PERFORMING ORGANIZATION The Aerospace Corporation Laboratory Operations 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Space Division
6c. ADDRESS (City, State, and ZIP Code) El Segundo, CA 90245	7b ADDRESS (City, State, and ZIP Code) Los Angeles Air Force Base Los Angeles, CA 90009-2960
8a. NAME OF FUNDING / SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F04701-85-C-0086-P00019
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Integrated Evaluation of Parallel Systems

12. PERSONAL AUTHOR(S)
Kesselman, Carl F.; Gorlick, Michael M.; Bannister, Joseph A.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
	FROM ________ TO ________	1988, December, 19	37

16. SUPPLEMENTARY NOTATION

17 COSATI CODES

FIELD	GROUP	SUB-GROUP

18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Dependability, Error insertion, Fault tolerance, Gauge, HERMES, Parallel processing, Performance evaluation, Simulation

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

Parallel (multiple-processor) computer systems are used to meet requirements for high performance. Multiple processors can also be used to achieve dependability through fault tolerance; however, the mere presence of more than one processor does not guarantee dependability. Where there are requirements for both high performance and dependability, the prudent designer of dependable parallel systems must judiciously balance both requirements.

The Computer Science Laboratory of The Aerospace Corporation has developed a sophisticated approach, based on simulation, that is more flexible, accurate, and cost effective than other approaches for investigating how dependability and performance interact. We define the nature of the analysis problem, and we discuss our approach to measuring performance and evaluating dependability in a single environment through the use of two of our integrated tools, HERMES and Gauge.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c OFFICE SYMBOL

DD FORM 1473, 84 MAR   83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Preface

The authors gratefully acknowledge the contributions of Mel Cutler, Mark Joseph, David Lee, and Mike Meyer in the preparation of this report. In addition, the authors thank Eugene D. Brooks III for providing the Cerberus Multiprocessor Simulator. and the Free Software Foundation for their cooperation in the retargeting of GCC.

Contents

Preface **1**

1 Introduction **5**

2 Using Simulation to Evaluate the Performance and Dependability of Parallel Systems **7**

2.1 Evaluating Performance 9

2.2 Evaluating Dependability 11

2.3 The Need for Integrated Tools 16

3 Tools for Evaluating the Performance and Dependability of Parallel Systems **19**

3.1 HERMES 19

3.2 Gauge 23

4 Related Work **31**

5 Conclusion **33**

References **35**

List of Figures

1 Hierarchical Approach to the Simulation of Injected Errors . . 13
2 The Structure of Gauge 23
3 Control Panel for Performance Analysis 25
4 A Typical Experiment Specification 29

1 Introduction

Space Division (SD) plans to meet its projected need for a high level of computational performance by using parallel-processing (multiple-processor) computer systems. Parallel systems are well suited for this purpose, since multiple processors can often improve the rate of processing over that obtained by traditional uniprocessor systems. Parallel systems, because they can provide redundancy, also have the potential for improving dependability. This report proposes a new, integrated approach to evaluating both the performance and dependability of parallel systems.

For the purposes of this discussion, we use the the definition of dependability given by Avižienis and Laprie [1]: "**Dependability** is that property of a computer system that allows *reliance to be justifiably placed on the service it delivers* [original emphases]." Dependability is a qualitative property consisting of the following components: reliability, availability, readiness, maintainability, testability, and safety. We are particularly interested in systems that achieve their dependability through fault tolerance. Looking again to [1], we define fault tolerance as a means "...to provide, *by redundancy* [original emphasis], service complying with the specification in spite of faults having occurred or occurring." If a system is dependable, it has a high probability of delivering a service within a given time after a request.

Unfortunately, when high-performance parallel systems must also be dependable, requirements for both performance and dependability can conflict. The prudent designer must be able to make the tradeoffs that lead to a judicious balance of both qualities for each application.

The level of effort necessary to design and validate parallel systems has increased considerably over that required for the less complex systems of the previous generation. Parallel computing systems (called "embedded" when they are highly integrated, autonomous parts of larger systems) may have to perform functions requiring several hundred million instructions per second (MIPS) and have unattended lifetimes of up to ten years. In general, commercial parallel computing systems (for example, the Sequent Symmetry) are used for interactive or numerical computing. Not only are the performance burdens on these systems less than is expected for systems that interest SD, but commercial systems can be maintained, and be repaired when they fail.

It is essential that fault tolerance be designed into a system at all levels, from primitive circuit elements up to — and perhaps beyond — the program level. Unfortunately, the techniques for designing dependable, high-performance computing systems, particularly for optimizing system software to take full advantage of parallel processors, have not kept up with technology. There is an unmet need for tools and techniques to evaluate both the performance and fault tolerance of new designs.

Our research in the Computer Science Laboratory (CSL) of The Aerospace Corporation focuses on developing simulation-based tools for evaluating the performance and fault-tolerance characteristics of parallel embedded systems. We propose to develop an integrated tool suite that permits the designer or researcher to perform in-depth studies of how well a system performs and tolerates faults.

Our approach to system evaluation is unique in how it addresses the following issues:

- We consider the evaluation of performance and fault tolerance as an integrated process.
- We use *actual*, not modeled, code to study how software and hardware interact.
- We provide an integrated environment for performing experiments, analyzing data, and displaying results.

The remainder of this report is organized as follows. In Section 2 we illustrate the need to integrate the evaluation of performance with that of fault tolerance, and propose an approach based on the efficient simulation of a computer system with its actual software. In Section 3 we describe HERMES and Gauge, tools for the integrated evaluation of performance and fault tolerance. In Section 4 we cite efforts related to our work, and we present conclusions in Section 5.

2 Using Simulation to Evaluate the Performance and Dependability of Parallel Systems

Requirements for the dependability and performance of computer systems are interdependent. Limited resources of space, power, and weight must be carefully allocated to achieve fault tolerance and high performance simultaneously. For example, fault-tolerance requirements for a microprocessor may mean that less chip area is available for on-chip cache memory, thus decreasing processor performance. Also, software devoted to fault-tolerance functions appropriates processor cycles that would otherwise be devoted to application software. An extreme but useful example is the SIFT (Software-Implemented Fault-Tolerance) computer, a fault-tolerant avionics processor whose fault-detection and fault-recovery mechanisms reduced its performance by half [2].

The complex nature of a computer system makes its performance difficult to predict. At unpredictable points an increase in the processing load can result in a severe degradation in system performance; the added demands of fault tolerance can leave an embedded system with insufficient computing reserves to respond adequately to a high-priority request.

For example, real-time systems have performance requirements that specify the maximum acceptable frequency at which response-time violations may occur. The incorporation of fault-tolerance features such as software consensus mechanisms [3], automatic assertions [4], and recovery blocks [5] inevitably increases the chances that response-time violations will occur. The occurrence of an error — although a relatively rare event — also promotes the missing of deadlines. Additionally, an error tends to cause a number of successively missed deadlines. The limit on the number of successive response-time violations is an often-neglected component of real-time performance requirements. Simulation can expose those errors that produce unacceptably long sequences of violations in the modeled system.

By observing the performance characteristics of a program while testing its fault-tolerance properties, we can compare the performance of the original program with that of the fault-tolerant version, thereby quantifying the tradeoff of performance for dependability. It is important to compare performance in both the absence and presence of errors.

It is possible to evaluate the performance and dependability of a system in three fundamental ways. (1) Test a system empirically by physically inducing faults or errors and observing the system's response. (2) Predict the reliability of a system by analyzing mathematical models. (3) Simulate the fault modes of a system and the ways in which fault-tolerance mechanisms react to these modes.

The empirical approach (benchmarking on actual hardware) can provide a high degree of accuracy and confidence in the results. However, it is costly and often comes too late in the design cycle to be of use; this is especially true of limited-production systems. Moreover, technical problems, such as the injection of faults into an integrated circuit, often make the empirical approach infeasible. Mathematical modeling can be applied to systems prior to their construction. However, mathematical modeling often makes simplistic assumptions about how the real system operates, and results must be interpreted judiciously. There is also a limit to the level of detail that can be modeled analytically; the analysis must typically be performed at a very high level because low-level analysis is intractable.

To date, simulation has not been widely used to evaluate dependability, but there are strong indications that simulation will be fruitful in aspects of dependability evaluation that are now neglected. Simulation is also an excellent technique for gaining insight into the critical performance parameters of a computer system: its speed, resource utilization, and efficiency. Simulation not only can predict the performance of proposed computers, but also can provide critical information on aspects of existing systems that are difficult or even impossible to measure directly. This includes the study of error detection and recovery, which have proved difficult to analyze.

Given the high degree of detail necessary to model how errors propagate, simulation is the method of choice. The simulation of computer systems can range from the gate level up to high-level behavior, encompassing a single processor or a complete computer system. We are currently focusing on the instruction-level simulation of parallel-processing systems, emphasizing the performance and accuracy of the simulation.

In Sections 2.1 and 2.2 we discuss our approach to using simulation to explore issues related to both performance and dependability, respectively. We end in Section 2.3 with a discussion of the need for a set of integrated tools.

2.1 Evaluating Performance

The design of a computer architecture for an embedded system is concerned with a single problem: to obtain the maximum performance within the design constraints of a given application. Determining the performance of a computer system for a specific application requires that any simulation of the embedded system account for the whole system: processors, memory, buses, system software, application software, and error-recovery mechanisms.

Appropriate conclusions can be drawn from simulation data only when appropriate input data are provided to the simulation. Traditionally, one measures system performance by executing a random sequence of instructions selected from an instruction mix. An instruction mix (for example, the Gibson mix [6]) is chosen to represent all of the applications a computer might perform. Other mixes (for example, the DAIS [Digital Avionics Information System] mix [7]) are designed to represent a specific class of applications.

However, the simulation of a random collection of machine instructions does not indicate how a computer system will perform when executing the instruction sequence generated by a *specific* program with *specific* input data; thus generalized instruction mixes are inappropriate for measuring the actual performance of an embedded computer system. For example, a computer might have a slow instruction-execution rate (in MIPS) but have a fast floating-point operation rate (as measured in millions of floating-point operations per second, or MFLOPS). In this computer, the performance of operating-system functions would be poor relative to that of matrix multiplication.

The number of simulated instructions executed is also important. The more instructions executed, the more valid the simulation results. A good example of this is the difference between warm-start and cold-start performance in a cache memory. The length of the simulation run must represent actual patterns of system use.

Our approach to simulation takes advantage of the fact that embedded systems, though demanding, are focused on well-defined applications. Given the specifications of a particular application, we can model it under a variety of scenarios:

- The system software exists and we want to see if the proposed architecture is adequate.
- The system hardware is designed and the system software is being developed.
- The system hardware is designed, the system software exists, and we need to improve performance (via "what-if" experiments).
- The system hardware is designed, the system software exists, and we want to determine if performance requirements are met — without resorting to benchmarks that may be expensive or impossible to perform.
- The system hardware is designed, the system software exists, and we want to observe how the system performs in the presence of faults.

In all of the above cases, actual application software and actual input data are used. The data are collected and analyzed, and the performance of the system as a whole — both hardware and software — is then determined.

However, the utility of simulation is limited if the simulated execution of a program takes several orders of magnitude longer than the actual execution. The simulated execution of a parallel application further increases (by at least an order of magnitude) the time required for the simulation. By optimizing the performance of the simulator, we can increase the size and detail of our simulations.

We can obtain a high degree of simulator performance by combining state-of-the-art hardware with the latest software techniques. We strive to obtain a one-to-one ratio of actual execution time to simulated execution time. When simulating a multiprocessor architecture, we execute the simulator itself on a multiprocessor. We have already constructed a prototype simulator for the

Generic VHSIC (Very High-Speed Integrated Circuit) Spaceborne Computer (GVSC) chip set [8]. Preliminary results show that we can simulate the MIL-STD-1750A instruction set at a rate of about 0.150 MIPS, 1/30 the rate of the actual chips.

2.2 Evaluating Dependability

Several tools for dependability modeling are currently available. Unfortunately, the most commonly used — ARIES 82 [9], CARE III [10], HARP [11], and SHARPE [12] — do not provide estimates of *coverage*, a figure of merit for how well mechanisms detect and recover from faults or errors; indeed, these tools typically require that the user provide coverage as an input parameter, and coverage has proven particularly difficult to estimate. Realizing that the term *coverage* has both a quantitative and a qualitative sense, we propose a technique for evaluating the coverage provided by various error-detection and recovery mechanisms. This report illustrates the proposed technique with regard to determining qualitative coverage; however, the technique may be extended to determine quantitative coverage as well.

Coverage at the programming level depends on how well a computer program recognizes that an error has occurred, what it does to contain the spread of the error, and how effectively it recovers from the effects of the error. Because the design of fault-tolerant programs is not yet well understood, many fundamental questions must be answered before software with the high levels of dependability required by current and projected SD applications can be designed. A few of the most important questions include the following:

- What types of errors are statistically most likely to appear in a given combination of hardware and software?
- What is the impact of a particular type of error on various computer programs?
- How do errors propagate through a computer system?
- What kinds of program constructs (data and control structures) are most susceptible to errors?

- What performance penalties are paid by fault-tolerant programs, and how will the occurrence of an error affect real-time performance?

Answering each of these questions is a significant task in its own right. We discuss these questions in detail below.

Since it is not the case that every error in the state of a computation will invalidate the results of that computation, one of the most challenging problems of designing for fault tolerance is how to characterize accurately those errors that can occur without disrupting correct processing. Simulation experiments can reveal the set of error patterns that a program might encounter, as well as the probability of their occurrence. Once these have been identified, we can select and evaluate the appropriate error-detection and correction mechanisms.

Drawing from earlier work [13,14], we take a hierarchical approach to such experiments (see Figure 1), modeling each level of the system in turn. Different fault or error models must be tailored to each level. Adjacent levels are related to each other by an abstract mapping (which is typically not one-to-one) that translates error patterns at the lower level to error patterns at the next higher level. For example, the fault model at the gate level might be based on stuck-at faults; at the next higher level, the register transfer level, these faults can map to invalid operations or erroneous register contents. We point out that this hierarchical approach does not require traditional mixed-mode simulation; rather than simulating higher and lower levels alternately, it extracts error-pattern frequencies exclusively from the lower level and uses these frequencies as inputs to the next higher level. While some detail is thus sacrificed, the overall advantages of this approach are that we avoid the technical difficulty of precisely relating one mode of simulation to another, and we are able to run simulations more rapidly than before.

Given a model for a level, as well as an error set and its associated probabilities of occurring at that level, we can use a mapping to inject likely errors into the next higher level. This leads to a new error set appropriate to the next level. In turn, this process can be repeatedly applied to ever higher levels. For example, at the hardware level we simulate the gate-level operation of a chip. First, by using plausible statistical distributions to estimate the

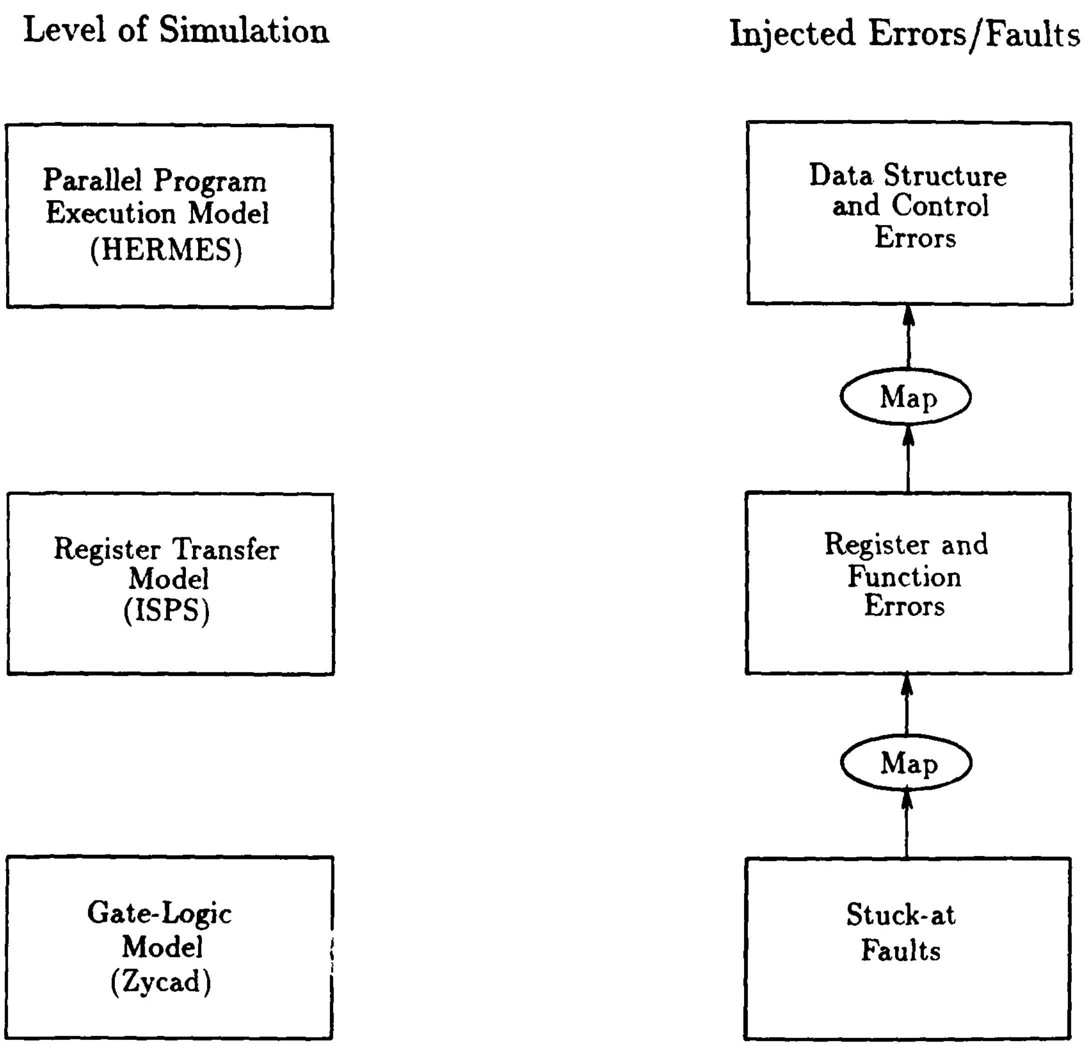

Level of Simulation
Injected Errors/Faults
Parallel Program Execution Model (HERMES)
Data Structure and Control Errors
Map
Register Transfer Model (ISPS)
Register and Function Errors
Map
Gate-Logic Model (Zycad)
Stuck-at Faults

Figure 1: Hierarchical Approach to the Simulation of Injected Errors

type and location of faults in the chips under study, we inject stuck-at faults where they are most likely to occur. This step involves an approximation of reality, since our limited knowledge of what causes faults in a chip forces us to conjecture about the likelihood that a particular location on a chip will be afflicted by a given fault. Second, we collect statistics about the error patterns that would occur in the actual chip. Third, by simulating a collection of such chips at the register-transfer level, we collect statistics about error patterns that manifest themselves in entities visible to the programmer (for example, in general-purpose and floating-point registers, memory locations, and fixed- and floating-point arithmetic units). Finally, these error patterns form the error set of the hardware/software interface.

By adopting a hierarchical approach, we break a complex problem into a number of simpler problems. Clearly, performing a number of simulation experiments at each level is far less costly than attempting to model such high-level behavior as program execution in terms of such low-level behavior as gate operation.

However, when the hierarchical approach is used, accuracy is sacrificed for speed. The experiments at each level are run independently of experiments at other levels. Even though separate levels are rarely independent, we postulate that their interdependence is weak, and that it can be accounted for through the statistical characterization of error patterns projected from a given level to the next higher level. The assumption that levels are, for all practical purposes, independent is necessary if we are to make any progress in estimating coverage.

Another area of research investigates how specific programs are affected by similar errors. As different programs use hardware resources in different ways, it is reasonable to expect different programs to behave differently when they are subjected to various types of errors. We propose to study two distinct aspects of this phenomenon:

- How does a program react to different kinds of errors?
- How do different kinds of programs react to the same kinds of errors?

Exhaustive testing inherently injects errors that, although they have specific effects, remain statistically insignificant. Therefore we are interested primarily in investigating those classes of errors that manifest themselves at the program level; that is, by using statistical analysis, how can we determine which faults are most likely to affect critical parts of a running program?

By observing a simulated behavior in the presence of an injected error, we have the opportunity to compare the tested program with an identical program that executes in a fault-free environment. This provides a wealth of data suitable for statistical analysis, such as the following:

- The elapsed time from the manifestation of a lower-level error to the manifestation of an error in the computation
- The elapsed time from manifestation to detection (for those programs that have error-detection mechanisms)
- The most likely program state to be affected
- The rate at which the injected error spreads
- The probability that the injected error will cause an error in the program

This simulation approach, of injecting errors and analyzing their effects statistically, is ideal for enabling us to understand how software executes in parallel architectures.

Although not all faults and errors have effects at the program level, it is extremely important to contain or minimize the effects of those that might. The occurrence of an error in the execution of a program can cause undesirable changes in that program's state; these changes, which result when an erroneous value is stored into the program's variables, can propagate to various degrees, and have a spectrum of effects — from merely incorrect values that are never subsequently read, to programs whose harmful effects spill over into other programs. As it is virtually impossible to design systems able to deal with all eventualities, it is important to determine which errors must be contained or minimized. By understanding the rate and extent of error propagation, we are better able to design firewalls to contain errors.

As some errors are inevitable in any system, programming techniques such as control-flow checking [15,16], assertion-based value checking [4], N-version programming [3], and recovery blocks [5] have been developed to detect and contain errors manifested at the program level. Our simulation approach permits us to evaluate not only the effectiveness of these techniques for fault tolerance, but also their impact on the performance of a specific application.

2.3 The Need for Integrated Tools

As our primary concern is the *mutual performance* and dependability of multiple-processor systems in the presence of faults, the technical issues raised by the simulation of fault tolerance — error injection, the timing of error detection, the performance of error-recovery mechanisms, and the tracing of how incorrect data values propagate — must be taken into account by tools also concerned with the classical measurement and analysis of software performance. Given the variety of performance issues, the varying levels of detail, and the intertwining of fault-tolerance considerations throughout, it is hard to imagine a single tool that could treat all aspects of system performance in the presence of faults.

Also, as the number of experiments can be large (a fault-tolerance study can require thousands of different experiments), careful record keeping is required to correlate software and hardware changes with experimental results. This makes some form of integrated, automated assistance a necessity.

We view transparent tool intercommunication and experiment design and management as the primary means of presenting the user with an environment that automatically integrates data-collection tools with data-analysis tools. A single high-level description of an experiment — which can consist of varying combinations of performance-data specification, data input to the simulation, data collection, data reduction and analysis, and varying configurations of parameters — relieves the user of the burden of specifying low-level details. Given multiple tools, data sharing and tool intercommunication are unavoidable. An ideal analysis system must rely on a versatile yet integrated *set* of tools. Uniform methods for storing and retrieving information are obvious and necessary precursors of this integration.

The following section elaborates on two principal tool sets, HERMES and Gauge, that meet these requirements.

3 Tools for Evaluating the Performance and Dependability of Parallel Systems

Automated aids are essential for dependability studies. For reasons of modularity and flexibility, we have separate tools for simulation and the organization, management, and analysis of resulting data.

Simulations are performed by the High-performance Extensible Retargetable Multiprocessor Emulation System (HERMES), an architecture simulator being implemented by CSL. HERMES can measure both performance and fault-tolerance properties simultaneously.

Experiments are controlled and data are managed by a single system also being developed by CSL. This system, called Gauge, uses a high-level specification of experiments to control simulations under HERMES and other tools.

Both HERMES and Gauge run under the UNIX[1] operating system on a wide variety of hosts, including the Sun workstation, the DEC VAX,[2] the Sequent Symmetry,[3] and the Encore Multimax.[4]

3.1 HERMES

HERMES is an environment for software development and architecture simulation (or emulation). One executes a simulation under HERMES in a manner similar to executing a program on any computer system. Programs are written in a high-level language, then are compiled and linked to create an executable program comprised of the instruction set of the simulated target architecture. The executable program and the operating system are loaded into the simulated memory, the machine state is initialized, and the program is executed.

The core of the HERMES system is an instruction-level simulator that accounts

[1]UNIX is a registered trademark of AT&T.
[2]VAX is a trademark of Digital Equipment Corporation.
[3]Symmetry is a registered trademark of Sequent Computer Systems Corporation.
[4]Multimax is a trademark of Encore Computer Corporation.

for the details of a specific implementation of a specific instruction set. A different emulator is custom built for each implementation of an instruction set. For example, we are currently developing simulators for the MIL-STD-1750A instruction set as implemented by the GVSC chip sets.

The four characteristics of HERMES — its high performance, its extensibility, its retargetability, and its ability to emulate parallel processors — are detailed as follows:

High Performance HERMES is built around a high-performance software emulator that executes programs developed for parallel and fault-tolerant execution. This emulator is based on the techniques developed at Lawrence Livermore National Laboratory for the Cerberus multiprocessor simulator [17]. The emulator is designed to achieve MIPS rates as fast as 1/20 the speed of the actual hardware when executing on that hardware. HERMES itself is written to be executed in parallel, thus adding to the speed at which it can execute simulations. For this purpose CSL uses the Sequent Symmetry, a state-of-the-art commercial parallel processor.

Extensibility HERMES instruction-level simulators can be combined with other simulation modules to simulate a complete parallel-processing system. HERMES can simulate a large number of different architectures based on a single processor. For example, one can study how varying the number of independent memory banks in a shared-memory parallel processor affects the performance of a specific algorithm. We can also easily change the processor being modeled by substituting one processor simulator for another.

Retargetability Because we want to study the effects of different instruction sets on performance, the software development tools in HERMES (the compiler, assembler, linker, and loader) must be easily adaptable, or *retargetable*, to new instruction sets. The compiler and assembler used in HERMES have already been retargeted to four different instruction sets (DEC VAX, National Semiconductor NS32032, Motorola M68020, and MIL-STD-1750A).

HERMES supports the C programming language directly. The FORTRAN programming language is provided by a commercially available FORTRAN-to-C translator. The HERMES C compiler [18] performs most well-known global-optimization techniques and peephole optimization in a completely machine-independent manner. The code generated by this compiler is superior to that generated by many commercial compilers. New instructions can also be easily added, facilitating the study of the effect of minor changes in the instruction set of the processor being simulated. Also, pipeline scheduling, crucial for the efficient operation of RISC-type (Reduced Instruction Set Computer) processors, will be added to the compiler in the near future.

Multiprocessor Emulation To simulate more than one processor simultaneously, HERMES maintains individual states for each instantiation of the emulator. Interactions between processors are modeled exactly, not probabilistically. This modeling precision provides the most accurate data possible on actual program operation. For example, if a processor needs to obtain a lock that is already being held, the amount of time the requesting processor must wait is determined by the time at which the holding processor releases the lock.

In addition, one can program parallel algorithms by using architecture-independent extensions [19] to the C programming language. These extensions enable one to write a parallel program that uses a variety of parallel-programming constructs (for example, locks, barriers, message passing, and shared memory).

Clearly HERMES is well suited for studies of the total execution time of parallel, fault-tolerant systems. However, HERMES also provides for a variety of other useful measurements. It is not the purpose of HERMES to provide an exhaustive set of measurements; rather, it implements only those measurements that are most critical and do not significantly slow down the simulation. Although this obliges the researcher to discover how best to extract complex performance parameters from data that can be gathered efficiently, HERMES can execute much larger simulations than are possible with other tools. For example, the HERMES compiler can instrument a program by strategically placing counters and instructions to increment the counters. After the program has executed, the values of these counters can be used to determine the execution times of various parts of the program [20]. Not only does this result in less interference with the program, but it also does not significantly slow down the emulator. Similar techniques can be used for fault-tolerance and hardware-performance measures.

HERMES obtains performance data by using four built-in mechanisms: a simulation clock, counters, shadow store, and special-purpose instructions.

Simulation Clock During the simulation HERMES maintains a clock for each processor. For every instruction executed, this clock is incremented by an appropriate amount — possibly even zero. An increment of zero enables one to manipulate the simulation from within the program, without affecting visible behavior.

Counters For software studies HERMES provides a program with counters to record the occurrence of a variety of events. These counters are incremented and read by means of additional machine instructions that execute in zero simulated time. Counters can be used to obtain information about many aspects of the program being studied.

Shadow Store For studies of how hardware operates at the instruction level, HERMES "shadows" every storage element in the programming model. This shadow store is used to maintain information about the corresponding storage. For example, when a hardware error occurs, the emulator marks the erroneous value by setting a bit in the corresponding shadow store. This "error" bit is propagated to other shadow stores, thus paralleling the propagation of the original error. This is a valuable technique for determining how well fault-tolerance mechanisms contain errors. In addition to tracing single bits, the shadow-store technique can provide information on other aspects of a system (for example, the number of times registers are accessed and updated).

Special-Purpose Instructions As HERMES was designed to be flexible, adding new instructions to the emulator is straightforward. For example, one can simulate the effect of an error in a functional unit by using a new "mutated" instruction in place of the original one; this can be used to produce the same effect as the compiler-induced mutations of [21]. When combined with shadow store, the use of mutated instructions is an extremely powerful technique.

3.2 Gauge

Gauge is an environment for controlling multiple simulators, recording experimental data, and analyzing the results [22]. The structure of Gauge is illustrated in Figure 2. Gauge distinguishes between the *platform* under test (some combination of hardware and software whose performance is the subject of investigation) and the *environment* (the probes and tools) used to analyze the platform.

The programmer communicates with the platform via Gauge, sending experiments to the platform and receiving results in reply. A standard interface defines the instrumentation and communication services that Gauge expects the platform to support. Experimental data are collected by the execution of an instrumented program (instrumentation is fully automatic). That program executes on a platform modified to generate the required measurements.

The key features of Gauge are summarized below.

Multiple Shared Data Bases Gauge provides a simple but extremely powerful data-base facility that allows tools to create, modify, share, and destroy data bases freely. Data bases and directed streams attached to data bases are the only means of intertool communication. The data bases contain all information of interest to Gauge. By creating new data bases from old, the user obtains different views of performance or

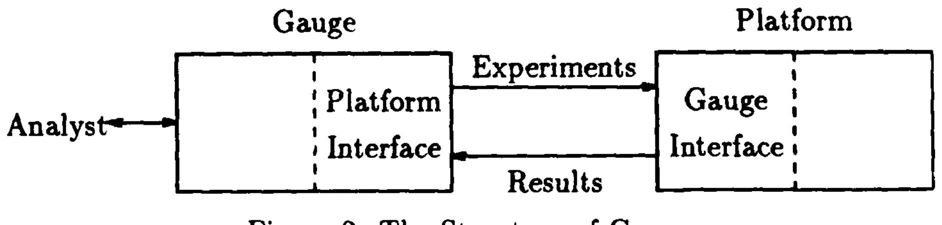


Figure 2: The Structure of Gauge

fault-tolerant behavior. Data bases may be read and written as ASCII files, permitting Gauge to communicate with other tools or environments.

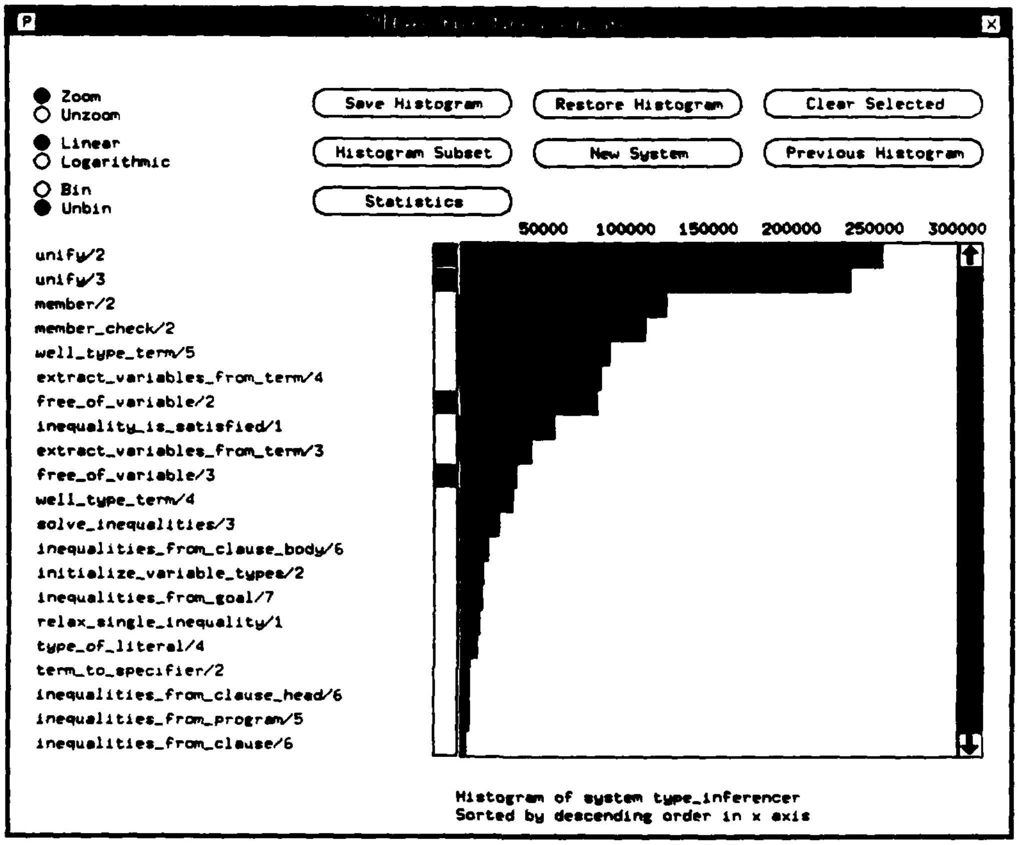
Tool Integration Gauge provides as a basic service the integration of goal-directed tools. First the user states, in a declarative manner, the desired effect; then Gauge automatically plans and executes a network of tool invocations to achieve that effect. As new tools are added to the Gauge environment, they are automatically taken into account during the planning phase. New tools can be added and old tools changed without concern for their impact on other tools in the environment.

Experiments Gauge is designed to be a hospitable environment for conducting, recording, and analyzing performance and fault-tolerance experiments. This includes a high-level language for the design and specification of performance and fault-tolerance experiments; tools for conducting and controlling experiments; and simple, standard interfaces for downloading experiments onto platforms and obtaining results in turn. The mechanisms of Gauge encourage the building of specialized. easily interconnected performance-analysis tools that can either generate experiments themselves or take advantage of experimental results.

Data Reduction Gauge includes standard tools for a variety of analyses that commonly arise in performance and fault-tolerance studies. The tools cover basic descriptive statistics, multivariate linear regression, correlation, and sensitivity. Other tools provide graphics for the display of experimental data (a variety of graphics formats are available). In particular, Gauge can succinctly display performance data from several hundred processors simultaneously.

Interactive Exploration Gauge is conducive to the interactive exploration of performance data. It automatically tracks and checkpoints each data tour, allowing a user to pursue an interesting side issue without losing the main trail of investigation. The entire state of the analysis can be saved and later restarted at the point of interruption.

Graphics All of Gauge is driven by convenient menus and control panels. Gauge includes a menu compiler that allows the user to expand the



● Zoom
○ Unzoom
● Linear
○ Logarithmic
○ Bin
● Unbin

unify/2
unify/3
member/2
member_check/2
well_type_term/5
extract_variables_from_term/4
free_of_variable/2
inequality_is_satisfied/1
extract_variables_from_term/3
free_of_variable/3
well_type_term/4
solve_inequalities/3
inequalities_from_clause_body/6
initialize_variable_types/2
inequalities_from_goal/7
relax_single_inequality/1
type_of_literal/4
term_to_specifier/2
inequalities_from_clause_head/6
inequalities_from_program/5
inequalities_from_clause/6

Histogram of system type_inferencer
Sorted by descending order in x axis

Figure 3: Control Panel for Performance Analysis

interface to include new tools or options and a control-panel compiler that allows the user to create specialized control panels for specific analyses. Menus and control panels may be nested, permitting the user to build on existing interfaces. The menu and control-panel compilers accept a descriptive language that describes the use of the interface. Issues of graphic presentation and organization are dealt with by the compiler. A sample control panel for performance analysis is shown in Figure 3.

Below we discuss two of the most important aspects of Gauge, tool integration and experiment management.

The analysis of performance data can be very complex, because of the diversity of performance data and their interrelationships. Gauge provides an automatic integration mechanism, called Protean, for performing experiments and analyzing the results.

Traditionally, the addition of each new analysis tool requires one to consider how the new tool interacts with *all* other tools in use; from the user's point of view, each new tool entails yet another set of conventions to memorize and master. With Gauge, however, Protean also standardizes the interaction between tools, allowing new tools to be integrated easily.

We strongly believe that users should obtain results by simply describing their intent, and *that it is the responsibility of the environment to construct* and execute a detailed plan for attaining that goal. Protean is driven entirely by a set of transformation rules that describe the input/output relations of tools; *its behavior is changed by the substitution of one set of rules for another.* Embedding a new tool inside Gauge means adding one or more transformation rules. From the descriptions given by the transformation rules, Protean automatically weaves together a directed network of tool invocations that manage sequencing, interconnectivity, and intertool communication.

Since Protean is the sole interface to the tool-composition services of Gauge, it encourages a high degree of tool modularity. Furthermore, because Protean allows tools to be easily and automatically connected to other tools for services, each individual tool can be more specialized. Protean synthesizes large tools from small subtools, connecting them via a uniform strategy.

One benefit of this scheme is the elimination of unnecessary duplications. For example, with Protean all tools share a common, interactive graphics interface (itself just a collection of Protean-mediated tools) or a set of basic statistics tools. Furthermore, since any individual tool is likely to be smaller and less comprehensive than it might be in a less integrated environment, we have greater confidence in its correctness.

Finally, Protean encourages the rapid prototyping of new tools, since users are able to *either* build upon *or* adapt existing tool fragments. One can combine old tools in new ways by modifying the transformation rules that describe the behavior of these tools. In particular, an analyst can seriously consider building specific tools for system- or application-dependent experiments and measures (for example, those dealing with correlating error recovery with paging).

The analyst interacts with Protean by stating his needs as relations that Protean tries to satisfy. The relation

```
histogram(work,procedure-(execution_time/procedure))
```

produces a histogram of the execution times of all procedures called by the procedure `work` and its descendants. Protean produces the display by stringing together nine tools to compute the reflexive and transitive closure of the call graph of `work`, interpret the performance results from the latest experiments, and extract the histogram. From Protean's point of view, performance experiments are simply a binary relation between an experiment and a set of results that is satisfied by downloading the test code to the platform, running the test program, and uploading the measurements.

Relations are implemented as tools that accept data bases as inputs and produce data bases as outputs. They are narrow and highly specialized — each is designed to do one job extremely well. Tools are ignorant of the source of their inputs and indifferent to the destination of their outputs. No input is ever changed, and any side effects that tools produce are local to their outputs. To give an idea of the scope and construction of these tools, we present two examples below.

1. The `histogram_display` tool is responsible for displaying a histogram on a bit-mapped display. Its input is a data base that refers to zero or more instances of the relation `X-Y`, where `X` is any datum (usually a string denoting the name of a procedure) and `Y` is always a number. The tool's output is a window that contains the display of the histogram described by the input data base.

2. The `reflexive_transitive_closure` tool computes the closure of the call graph of a procedure P by recursively collecting the contents of the call graph of P and its descendants.

Gauge mediates all performance and fault-tolerance experiments. The user controls a platform via requests to Gauge. Gauge in turn interacts with the platform by constructing an *experiment*, a data base given to the platform for interpretation that describes the design and results of a test. The design of a performance experiment has three parts:

- a description of the components of the software under test that specifies how they are combined to form a complete working system;
- the environment of the test [for example, the nature of the processor(s), the quantity of memory, the memory allocation policies, the type and connectivity of communication streams]; and
- a definition of the test case.

A typical experiment specification is shown in Figure 4.

Experiment results have two parts. The first part specifies the environmental values that were finally obtained for the experiment. The initial environment of the experiment may have given some leeway (for example, by stating that a range of three to five megabytes of memory was sufficient for an experiment). The environment portion of the results informs Gauge of the amount of memory that was finally allocated for the experiment, say four megabytes. The second part consists of the platform-dependent measurement data.

Experiments and results are transferred between Gauge and the platform in a simple ASCII format consisting of sequences of name/value pairs. Standard routines are used on both sides to translate to and from the external representation. All phases of the experiment are recorded in data bases for future reference or reuse. Portions of earlier experiments can be combined to construct new experiments, or old experiments can be rerun to validate results. Gauge automatically catalogs and indexes all the information and data pertinent to an experiment, leaving the analyst free to concentrate on interpretation and diagnosis.

```
% System Description
software(attitude_control).
version(3.5).
release(2a).

...

% Test Environment
processor(honeywell(gvsc)).
nodes(8).
topology(hypercube).

...

% Test Case
measure(bus_traffic).
units(words/second).
input(test_file_007).

...
```

Figure 4: A Typical Experiment Specification

4 Related Work

Although it is a considerable advance in the state of the art, the work and approaches described above draw upon a large body of recent research and tools. We have built HERMES upon the foundation provided by Cerberus [17], and have rewritten about 95% of the original software. Also, with the exception of the simulation clock, we have added all of the performance analysis features described in this report. The Rice Parallel Processing Testbed (RPPT) [23] is a simulation-based tool for evaluating the performance of parallel C programs. RPPT uses "execution-driven" simulation, which has a coarser granularity than HERMES's finer-grained "instruction-driven" simulation. This difference makes HERMES better suited for dependability evaluation than RPPT.

HERMES significantly extends the work of Hua and Abraham [21], who evaluated the fault-tolerance properties of self-checking programs by using mutations of program statements to mimic the injection of hardware faults. The *program flow monitor*, another example of program-level fault tolerance, was studied by Schuette and Shen [15], who used physical fault injection; we plan to repeat the analogous simulation version of such real-life experiments and compare our results with theirs. The Fault Injection–based Automated Testing (FIAT) environment [24] injects representative error patterns into executing real-time distributed software. FIAT has been implemented for error injection into physical hardware, and its primary use is to test an actual system for fault-free behavior. However, FIAT does not cleanly integrate the evaluation of performance and dependability as does HERMES. Furthermore, it is unclear how FIAT generates representative error patterns, since there is no notion comparable to the hierarchical fault/error mapping used by HERMES. Presumably, in FIAT one must have a very clear concept of which *errors* one expects to see, whereas in Gauge the error patterns can be based upon the underlying physics of the hardware being modeled.

A variety of monitoring systems have been developed in response to the need to analyze the performance of multiple-processor systems. Close in structure to the combination of HERMES and Gauge is IIE [25], an integrated instrumentation environment for multiprocessors. Like Gauge, IIE acknowledges that the requirements of a performance-analysis environment are distinctly different from those of most programming environments, it emphasizes experiment *management as a fundamental contribution of performance* analysis environments, and it uses data bases as the common glue that unites analysis tools. However, Gauge differs significantly from IIE in three ways. First, Gauge makes available a much wider variety of analytic and graphics tools than IIE. Second, the Gauge data-base model is potentially much richer than the model adopted by IIE (see [26]). Finally, Gauge's novel use of Protean as a central organizer makes it possible to change the underlying tool set with an unusual ease and economy of expression; the flexibility and power of Gauge in this respect exceed considerably that of other similar tools known to us.

5 Conclusion

Because system performance and dependability are related, the complete analysis of these qualities must be performed in an integrated manner. CSL has developed, and is refining, tools for this purpose. These tools, HERMES and Gauge, work together, providing the system designer with previously unobtainable information about system performance and dependability.

The features of these tools can be summarized as follows:

- HERMES makes large experiments feasible by providing high-performance multiprocessor emulation with facilities for error injection and tracing.
- HERMES has a retargetable compiler and a modular emulator that together permit the consideration of alternative architectures and organizations.
- Gauge provides the analyst with intellectual control over the experiments and assistance in running them and collecting results.
- Gauge provides a uniform set of facilities for data collection, analysis, and display. These facilities also provide for the easy integration of new and existing tools.

The above tools must be used intelligently and the results analyzed carefully. However, with the information these tools provide, the system designer can judiciously trade off different processors, system architecture, and software to meet specific requirements for performance and dependability.

References

[1] A. Avižienis and J.-C. Laprie, "Dependable Computing: From Concepts to Design Diversity," *Proceedings of the IEEE*, Vol. 74, pp. 629–638, May 1986.

[2] J. H. Wensley *et al.*, "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control," *Proceedings of the IEEE*, Vol. 66, pp. 1240–1255, October 1978.

[3] A. Avižienis, "The N-Version Approach to Fault-Tolerant Software," *IEEE Transactions on Software Engineering*, Vol. SE-11, pp. 1491–1501, December 1985.

[4] D. M. Andrews, "Using Executable Assertions for Testing and Fault Tolerance," in *Proceedings of the Ninth International Fault Tolerant Computing Symposium*, pp. 102–105, June 1979.

[5] B. Randell, "System Structure for Fault Tolerance," *IEEE Transactions on Software Engineering*, Vol. SE-1, pp. 220–232, March 1975.

[6] J. C. Gibson, "The Gibson Mix," IBM Report TR00.2043, IBM, June 1970.

[7] "AN/AYK-15A Development Specification," Wright–Patterson Air Force Base, Ohio, April 1979. Air Force Wright Aeronautical Laboratories Document Number SA421205.

[8] W. R. Iversen, "Control Data Launches CMOS/SOS Space Computer," *Electronics Magazine*, pp. 27–28, July 10, 1986.

[9] S. V. Makam *et al.*, "UCLA ARIES 82 Users' Guide," Technical Report CSD-820830, UCLA Computer Science Department, Los Angeles, California, August 1982.

[10] J. Stiffler *et al.*, "CARE III Final Report, Phase I," NASA Contractor Report 159122, NASA Langley Research Center, November 1979.

[11] S. J. Bavuso *et al.*, "Analysis of Typical Fault-Tolerant Architectures using HARP," *IEEE Transactions on Reliability*, Vol. R-36, pp. 176–185, June 1987.

[12] R. A. Sahner and K. S. Trivedi, "Reliability Modeling Using SHARPE," *IEEE Transactions on Reliability*, Vol. R-36, pp. 186–193, June 1987.

[13] J. Abraham *et al.*, "Application of Fault Tolerance Technology," technical report, The Aerospace Corporation, El Segundo, California, October 20, 1987.

[14] M. K. Joseph and J. Bannister, "Coverage Estimation and Validation," technical report, The Aerospace Corporation, El Segundo, California, August 5, 1988.

[15] M. A. Schuette and J. P. Shen, "Processor Control Flow Monitoring Using Signatured Instruction Streams," *IEEE Transactions on Computers*, Vol. C-36, pp. 264–275, March 1986.

[16] S. S. Yau and F.-C. Chen, "An Approach to Concurrent Control Flow Checking," *IEEE Transactions on Software Engineering*, Vol. SE-6. pp. 126–137, March 1980.

[17] E. D. Brooks III, T. S. Axelrod, and G. A. Darmohray, "The Cerberus Multiprocessor Simulator," in *Proceedings of the Third SIAM Conference on Parallel Processing*, 1987.

[18] R. Stallman, *Internals of GNU CC.* Free Software Foundation, 1988.

[19] J. Boyle *et al.*, *Portable Programs for Parallel Processors*, (New York, New York: Holt, Rinehart and Winston, Inc., 1988).

[20] M. M. Gorlick and C. F. Kesselman, "Timing Prolog Programs Without Clocks," in *Proceedings 1987 Symposium on Logic Programming*, pp. 426–432, IEEE Computer Society Press, 1987.

[21] K. A. Hua and J. A. Abraham, "Design of Systems with Concurrent Error Detection Using Software Redundancy," in *Proceedings of the ACM/IEEE Fall Joint Computer Conference*, (Dallas, Texas), pp. 826–835, November 1986.

[22] M. M. Gorlick and C. F. Kesselman, "Gauge: A Workbench for the Performance Analysis of Logic Programs," in *Proceedings of the Fifth International Conference on Logic Programming*, MIT Press, August 1988.

[23] R. C. Covington *et al.*, "The Rice Parallel Processing Testbed," *Performance Evaluation Review*, Vol. 16, pp. 4–11, May 1988. Special Issue on the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems; May 24–27, 1988; Santa Fe, New Mexico.

[24] Z. Segall *et al.*, "FIAT — Fault Injection Based Automated Testing Environment," in *Proceedings of the 18th International Fault-Tolerant Computing Symposium*, (Tokyo, Japan), pp. 102–107, IEEE Computer Society, June 1988.

[25] Z. Segall *et al.*, "An Integrated Instrumentation Environment for Multiprocessors," *IEEE Transactions on Computers*, Vol. C-32, pp. 4–14, January 1983.

[26] R. T. Snodgrass, "A Relational Approach to Monitoring Complex Systems," *ACM Transactions on Computer Systems*, Vol. 6, pp. 157–196, May 1988.

LABORATORY OPERATIONS

The Aerospace Corporation functions as an "architect-engineer" for national security projects, specializing in advanced military space systems. Providing research support, the corporation's Laboratory Operations conducts experimental and theoretical investigations that focus on the application of scientific and technical advances to such systems. Vital to the success of these investigations is the technical staff's wide-ranging expertise and its ability to stay current with new developments. This expertise is enhanced by a research program aimed at dealing with the many problems associated with rapidly evolving space systems. Contributing their capabilities to the research effort are these individual laboratories:

Aerophysics Laboratory: Launch vehicle and reentry fluid mechanics, heat transfer and flight dynamics; chemical and electric propulsion, propellant chemistry, chemical dynamics, environmental chemistry, trace detection; spacecraft structural mechanics, contamination, thermal and structural control; high temperature thermomechanics, gas kinetics and radiation; cw and pulsed chemical and excimer laser development including chemical kinetics, spectroscopy, optical resonators, beam control, atmospheric propagation, laser effects and countermeasures.

Chemistry and Physics Laboratory: Atmospheric chemical reactions, atmospheric optics, light scattering, state-specific chemical reactions and radiative signatures of missile plumes, sensor out-of-field-of-view rejection, applied laser spectroscopy, laser chemistry, laser optoelectronics, solar cell physics, battery electrochemistry, space vacuum and radiation effects on materials, lubrication and surface phenomena, thermionic emission, photosensitive materials and detectors, atomic frequency standards, and environmental chemistry.

Computer Science Laboratory: Program verification, program translation, performance-sensitive system design, distributed architectures for spaceborne computers, fault-tolerant computer systems, artificial intelligence, microelectronics applications, communication protocols, and computer security.

Electronics Research Laboratory: Microelectronics, solid-state device physics, compound semiconductors, radiation hardening; electro-optics, quantum electronics, solid-state lasers, optical propagation and communications; microwave semiconductor devices, microwave/millimeter wave measurements, diagnostics and radiometry, microwave/millimeter wave thermionic devices; atomic time and frequency standards; antennas, rf systems, electromagnetic propagation phenomena, space communication systems.

Materials Sciences Laboratory: Development of new materials: metals, alloys, ceramics, polymers and their composites, and new forms of carbon; nondestructive evaluation, component failure analysis and reliability; fracture mechanics and stress corrosion; analysis and evaluation of materials at cryogenic and elevated temperatures as well as in space and enemy-induced environments.

Space Sciences Laboratory: Magnetospheric, auroral and cosmic ray physics, wave-particle interactions, magnetospheric plasma waves; atmospheric and ionospheric physics, density and composition of the upper atmosphere, remote sensing using atmospheric radiation; solar physics, infrared astronomy, infrared signature analysis; effects of solar activity, magnetic storms and nuclear explosions on the earth's atmosphere, ionosphere and magnetosphere; effects of electromagnetic and particulate radiations on space systems; space instrumentation.