Fault Injection

A Method for Validating Computer-System Dependability

Jeffrey A. Clark
*Mitre Corporation**

Dhiraj K. Pradhan
Texas A&M University

Fault injection is an effective solution to the problem of validating highly reliable computer systems. Tools such as React are facilitating its application.

With greater reliance on computers in a variety of applications, the consequences of failure and downtime have become more severe. In critical applications, such as aircraft flight control, nuclear reactor monitoring, medical life support, business transaction processing, and telecommunications switching, computing resource failures can cost lives and/or money.

Computers employed in such applications often incorporate redundancy to tolerate faults that would otherwise cause system failure. A fault-tolerant computer system's dependability must be validated to ensure that its redundancy has been correctly implemented and the system will provide the desired level of reliable service. Fault injection—the deliberate insertion of faults into an operational system to determine its response—offers an effective solution to this problem. In this article, we survey several fault-injection studies and discuss tools such as React (Reliable Architecture Characterization Tool) that facilitate its application.

COMPUTER-SYSTEM DEPENDABILITY

Dependability is a qualitative system attribute that is quantified through specific measures. The two primary measures of dependability are reliability and availability. Reliability is the probability of surviving (without failure) over an interval of time. Availability is the probability of being operational (not failed) at a given instant in time. The mean time to failure (MTTF) and the mean time between failures (MTBF) are also frequently used. Dependability is often evaluated empirically through life testing. However, the time needed to obtain a statistically significant number of failures makes life testing impractical for most fault-tolerant computers. Instead, analytical modeling is typically used to predict dependability.

Analytical dependability models enumerate a system's operational or failed states. Each state represents a unique combination of faults and their effects on system components. The times at which the faults occur are assumed to fit a particular statistical distribution. Several standardized procedures estimate the failure rates of electronic components when the underlying distribution is exponential. However, fault handling beyond this stage has been modeled in many different ways.

Most fault-handling models use coverage parameters to specify the probability of successfully performing the actions needed to recover from a fault. These actions include detecting the fault, identifying the affected component, and isolating that component through system reconfiguration. Each action must be taken quickly, before any additional faults that can overload the system's fault-handling mechanisms accumulate. For this reason, many models incorporate distributions of latency—the time needed to perform each of these actions. Because even small variations in coverage and latency can greatly affect dependability, these parameters should be estimated based on data from the actual system rather than approximated (see "Background" sidebar).

Fault-injection studies can provide this data through many individual experiments that vary how, where, and when the faults are intentionally

* *The views, opinions, and/or findings in this article are those of the authors, and should not be construed as the official positions, policies, and/or decisions of the Mitre Corporation or its government sponsors.*

0018-9162/95/$4.00 © 1995 IEEE

inserted. Large complex systems and time constraints make exhaustive insertion impractical; therefore, only a carefully chosen subset of all possible faults can usually be investigated. Insertion must be controlled so that the type, location, time, and duration of each fault, or the corresponding statistical distributions, are at least approximately known. Faults can be inserted into both the hardware and software components of a realized system or a simulation model that accurately reflects these components' behavior. During each experiment, the system must be operated with a representative work load to obtain a realistic response. The effects of each inserted fault are precisely monitored and recorded with instrumentation.

Besides supplying coverage and latency parameters for analytical models, fault injection can directly evaluate dependability metrics. It is particularly useful for measuring those system attributes that are difficult to model analytically—for example, the work load's influence on dependability. Fault injection aids design when it is used to functionally test a prototype during system development. It can identify implementation errors in fault-tolerance mechanisms and provide feedback on those mechanisms' efficiency. When the system is ready for deployment, fault injection can be used to observe the error or failure symptoms associated with each type of faulty component. Fault dictionaries can then be compiled to support system diagnosis during maintenance actions. Finally, fault-injection experiments provide a means for understanding how computer systems behave in the presence of faults. Such knowledge will ultimately lead to better system designs and higher dependability.

TAXONOMY OF EXPERIMENTS

Fault-injection experiments can be classified according to three general attributes: system abstraction, fault model and injection method, and dependability measure.

System abstractions

Fault-injection studies have traditionally been performed on the actual hardware and software of physical

A *fault* is a deviation in a hardware or software component from its intended function. Faults can arise during all stages in a computer system's evolution—specification, design, development, manufacturing, assembly, and installation—and throughout its operational life. Most faults that occur before full system deployment are discovered through testing and eliminated. Faults that are not removed can reduce a system's dependability when it is in the field. Despite the potential for such latent faults in computer systems, most fault-injection studies focus on the faults that occur during system operation.

Hardware faults occurring during system operation are categorized mainly by duration. *Permanent faults* are caused by irreversible device failures within a component due to damage, fatigue, or improper manufacturing. Once a permanent fault has occurred, the faulty component can be restored only by replacement or, if possible, repair. *Transient faults*, on the other hand, are triggered by environmental disturbances such as voltage fluctuations, electromagnetic interference, or radiation. These events typically have a short duration, returning the affected circuitry to a normal operating state without causing any lasting damage (although the system state may continue to be erroneous). Transients can be up to 100 times more frequent than permanents, depending on the system's particular operating environment. *Intermittent faults*, which tend to oscillate between periods of erroneous activity and dormancy, may also surface during system operation. They are often attributed to design errors that result in marginal or unstable hardware.

Software faults are caused by the incorrect specification, design, or coding of a program. Although software does not physically "break" after being installed in a computer system, latent faults or bugs in the code can surface during operation—especially under heavy or unusual work loads—and eventually lead to system failures. For this reason, software fault injection is employed primarily for testing programs or software-implemented fault-tolerance mechanisms. However, it has not seen widespread use in either application.

When a fault causes an incorrect change in machine state, an *error* occurs. The time between fault occurrence and the first appearance of an error is called the fault latency. Although a fault remains localized in the affected code or circuitry, multiple errors can originate from one fault site and propagate throughout the system. If the necessary mechanisms are present, they will detect a propagating error after a period of time, called the error latency. When the fault-tolerance mechanisms detect an error, they may initiate several actions to handle the fault and contain its errors. *Recovery* occurs if these actions are successful; otherwise, the system eventually malfunctions and a *failure* occurs.

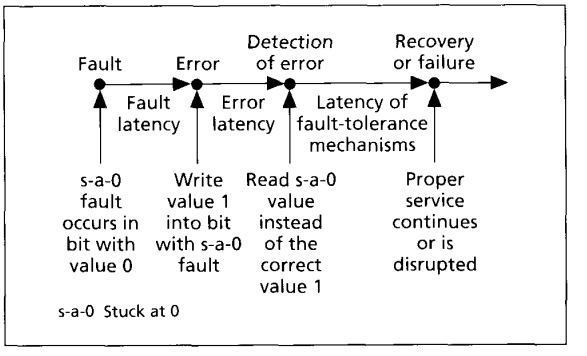

Figure A. Example of a fault, an error, and a failure.

Figure A provides an example to clarify the definitions of fault, error, and failure. Suppose a permanent stuck-at-0 (s-a-0) fault affects a memory bit with an initial value of logical 0. Some time later, an error occurs when a logical 1 is written into this bit. (If the faulty value had been opposite the initial value of this bit, an error would have manifested immediately with no fault latency.) The next read from the memory bit obtains the s-a-0 value instead of the correct value, 1, thereby detecting an error. Proper service continues if the system's fault-tolerance mechanisms can correct or mask this bit error. If not, service is disrupted.

computer systems. High levels of device integration, multiple-chip hybrid circuits, and dense packaging technologies limit accessibility to injection and instrumentation nodes. This makes it difficult to validate the hardware of physical systems. Simulation, on the other hand, has the advantage of relatively uninhibited access to a modeled system's internal nodes. The ability to precisely control and monitor injected faults, coupled with low-cost computer automation, and the potential for earlier application make simulated injection an attractive alternative to physical injection.

Simulated fault injection can support all system abstraction levels—architectural, functional, logical, and electrical. Mixed-mode simulation, where the system is hierarchically decomposed for simulation at different abstraction levels, is particularly useful for fault injection. This technique lets faults be accurately simulated at a low abstraction level, while the system responses are efficiently simulated at higher abstraction levels.

Fault models and injection methods

Simulated fault injection and most experiments involving physical hardware and software require selection of a fault model. The popular stuck-at fault model is commonly used for permanent hardware faults. However, subsequent errors often are of more concern than the faults themselves. This is particularly true for transient faults, whose unpredictable origin and relatively short life span make them difficult to characterize. Therefore, studies involving transients frequently employ an inversion model, where a fault immediately produces an error with the opposite logical value. Software errors arising from hardware faults are often modeled via bytes of 0s or 1s written into a data structure or portion of memory. Experimenters can use various other models, from detailed device-level to simplified functional-level models, to represent faults or their manifestations.

After choosing a fault model, the experimenter must determine how to inject the faults into the computer system. Locations frequently exploited when faults are injected into physical systems include IC leads, circuit board connectors, and the system back plane. The experimenter can generate faults at these external sites by temporarily inserting circuitry that corrupts the signals passing through a node without damaging any system components. Although signal corruption can model many faults that occur inside components, this method usually does not exercise all relevant hardware in the system. Therefore, experimenters cannot investigate the effects of some internal faults with this injection technique.

State mutation is one method of injecting errors inside system components. During normal system operation, processing is halted and special-purpose hardware or software is used to introduce errors. Scan paths, designed for system test and diagnosis, can be used to read the shift-register contents, modify selected bits, and shift the mutated state back into the machine. Privileged system calls and program debuggers can insert errors into a computer system by directly modifying its memory or register state. State mutation is the injection method used most often with simulated fault injection. Computer simulators are typically event driven, updating a modeled system's state at discrete times rather than continuously. Fault injections are easily made between event time boundaries. However, because it requires stopping and restarting the processor to inject a fault, this technique is not always effective for measuring latencies in physical systems.

Several novel approaches exist for injecting internal faults in hardware. ICs are susceptible to single-event upsets (SEUs)—created when an ionizing particle passes through a transistor, generating excess charge. Computer systems in space applications are particularly vulnerable to SEUs from cosmic rays. In the laboratory, transient faults can be induced in a similar way through short-term exposure to heavy-ion radiation. However, these fault-injection experiments must be performed in a vacuum chamber with the lid of the target IC removed, since ions are easily attenuated by air. Radiation flux is distributed uniformly over the chip, and error rates can be adjusted by a change in the distance from the ion source. Shielding can confine faults to a particular region of the IC, but there is no direct control over where and when the injections occur.

Another means for injecting internal hardware faults is through power supply disturbances. Short, pulsed interruptions in power drop the supply voltage to levels that can increase propagation delays and discharge nodes, especially those in memory. Computer systems employed in industrial applications are often subject to similar noise on the power lines. Unlike radiation, which causes SEUs, power supply disturbances simultaneously affect many nodes in the target IC, producing multiple, transient bit faults. Unfortunately, the location of these faults cannot be readily controlled. This injection technique is quite sensitive to the pulse width and amplitude of the voltage disturbances. Effects can also vary widely with different circuit families and fabrication technologies, making it difficult to generalize results from such experiments.

The last method we consider for introducing faults into a computer system is called trace injection. This method first uses custom-monitoring hardware or software to periodically sample machine state or record memory references on an operational system. Then the acquired trace is used to simulate system behavior, as errors that mimic faults in the instrumented components are inserted into the trace. The quantity of data collected can be very large, limiting most traces to only a brief history of machine activity. It is therefore essential to associate some measure of system load (at the time the trace was obtained) with the results, to distinguish extremes in fault behavior from the norm.

Dependability measures

The traditional objective of fault-injection experimentation has been to estimate coverage and latency parameters for analytical dependability models. However, fault injection can also evaluate other dependability measures, including reliability or availability and MTTF or MTBF. Several failure classification experiments have analyzed how injected faults affect a computer system's service. Fault-injection studies have also investigated error propagation from a fault site to other system components. Finally, researchers have often observed a correlation between a system's dependability and either its computational load or characteristics of its application code. Such work load relationships are frequently explored via fault

injection. Figure 1 summarizes the system abstractions, injection methods, and dependability measures for classifying fault-injection experiments.

APPLICATIONS

Fault injection was first employed in the 1970s to assess the dependability of fault-tolerant computers. For some time afterward, fault injection was used almost exclusively by industry for measuring the coverage and latency parameters of highly reliable systems. Not until the mid-1980s did academia begin actively using fault injection to conduct experimental research. Initial work concentrated on understanding error propagation and analyzing the efficiency of new fault-detection mechanisms. Research has since expanded to include characterization of dependability at the system level and its relationship to work load.

Error propagation in a jet-engine controller

We first examine a study that explored error propagation in an HS1602 jet-engine controller with dual-channel redundancy. Choi and Iyer used the Focus simulation environment to inject transient faults into one of the two microprocessors in this controller.[1] They used mixed-mode simulation at the electrical and logical levels to deposit 0.5 to 9 pico-coulombs of excess charge onto different nodes of the microprocessor as it executed a phase of its application code. The excess charge models transients from the penetration of various heavy ions typically found in cosmic environments. The data in Table 1 is from a comparison of 2,100 simulated fault-injection experiments with a trace of the fault-free simulation. First-order errors are those manifested in the first clock cycle after fault injection. Errors manifested in the second and subsequent clock cycles are called second- and higher-order errors, respectively. Results indicate that nearly 80 percent of the injected transients had no impact, since errors had to be latched (stored in a memory element) to affect the microprocessor's state. Once latched, however, an error had more than a 50 percent chance of reaching a pin and more than a 40 percent chance of causing a functional error on the microprocessor's control outputs. By analyzing the individual contributions to these statistics by each of the HS1602's six functional units, Choi and Iyer discovered the most effective locations for incorporating additional fault-tolerant features.

Table 1. Transient fault severity.

Type	Percentage
First-order latch errors	22.4
Second- and higher-order latch errors	5.7
First-order pin errors	2.1
Second and higher-order pin errors	4.3
Functional errors	9.2

Table 2. Bus affected in the first erroneous cycle.

Bus affected	Heavy-ion radiation (percent)	Power supply disturbances (percent)
Address	64	17
Data	5	1
Control	27	80
Combination	4	2

System abstractions

Physical	Logical
Architectural	Electrical
Functional	Mixed-mode
Injection methods	**Dependability measures**
Signal corruption	Reliability/availability
State mutation	MTTF/MTBF
Radiation	Failure classification
Power supply disturbances	Coverage and latency
Trace injection	Error propagation
	Work load relationships

Figure 1. Summary of the experimental taxonomy.

Radiation and power supply disturbances

Karlsson et al. used radiation and power supply disturbances to investigate the propagation of internal errors to the bus of an MC6809E.[2] They injected transient faults into this microprocessor by exposing it to heavy ions from a Californium source and to – 4.2 V, 50-ns pulses on the microprocessor's 5V power supply. A reference MC6809E ran the same two test programs in lock-step synchronization with the microprocessor under test. Comparison of bus signals from the two microprocessors detected errors. Detection triggered a logic analyzer to record microprocessor activity for 200 bus cycles. Table 2 lists the bus affected in the first erroneous cycle based on 1,000 observations. Errors appeared mainly on the address bus in the radiation experiments, whereas errors on the control bus dominated the power-supply disturbances. Although the initial fault manifestations were quite different, the microprocessor's behavior over an extended period of time was almost identical for both injection techniques. As Table 3 shows, control-flow errors causing permanent divergence

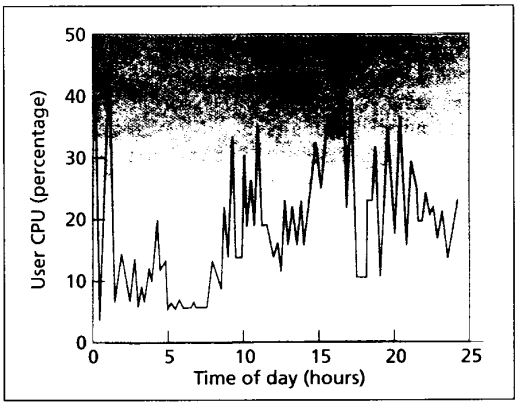

Figure 2. User CPU usage by time of day.

from the correct instruction stream were responsible for over 70 percent of the failures observed with heavy-ion radiation and power-supply disturbances. Karlsson et al. have evaluated the coverage and latency of several different concurrent error-detection schemes using these methods.

Trace injection to measure latency

Chillarege and Iyer were among the first to measure *fault and error latency in memory via trace injection.*[3] They ran a scanning process on a VAX 11/780 to periodically copy the contents of real memory locations into archival storage. The locations were randomly chosen from 4 to 10 regions in memory of up to 50 Kbytes each. These regions were repetitively scanned every 15 to 20 seconds under a medium to high system work load. Stuck-at bit faults were then simulated in the sampled words to calculate latency distribution parameters (given in Table 4) for a representative set of 960 faults. The mean fault latency was almost five times greater for s-a-0 (stuck-at-0) than for s-a-1 faults. Conversely, the mean error latency of the s-a-1 faults was more than double that of the s-a-0 faults. Chillarege and Iyer attributed the difference in latencies to unequal lifetimes of 0s and 1s in the system due to the way memory is allocated and released. They conjectured that many programs use only a fraction of their allocated memory blocks. This would leave many 0s in memory, because blocks are initially cleared when they are allocated. Optimal memory scrubbing rates—the frequency at which single, transient bit errors are systematically corrected before any additional errors accumulate—are determined from such measurements of fault and error latency.

System work load and memory error latency

Chillarege and Iyer also used trace injection to analyze the relationship between system work load and memory error latency.[4] They collected data by probing the back plane of a VAX 11/780 and sampling physical memory activity at 40-second intervals. They also logged work load profiles during this data acquisition. Figure 2 graphs one measure of system work load, user CPU utilization (percentage of processing capacity in use), over a 24-hour period beginning at midnight. Work load was relatively low until shortly after 7 a.m. (except for a brief period around 1 a.m., when system routines were run), then rose significantly between 8 and 10 a.m., and peaked in the mid- to late-afternoon. Chillarege and Iyer used the memory activity data to simulate inverted bit errors occurring at different times of day. Error latency distributions for faults inserted at midnight and noon appear in Figures 3 and 4, respectively. Mean error latency varied from as long as eight hours at low work load to as short as 44 minutes

Table 3. Classification of processor errors.

Error class	Heavy-ion radiation (percent)	Power supply disturbances (percent)
Control-flow errors		
Permanent divergence	72	74
Temporary divergence	3	4
Not active within 200 cycles	2	0
Data errors		
Data only	5	2
Address/control also affected	15	16
Other errors		
Could cause failure	4	2
Could not cause failure	0	3

Table 4. Memory latency distribution parameters. All latencies are in minutes.

	Stuck at 0		Stuck at 1	
Latency	Mean	Standard deviation	Mean	Standard deviation
Fault	70.4	80.2	14.6	31.9
Error	20.6	31.2	45.4	47.9
Total	91.1	76.9	60.4	47.6

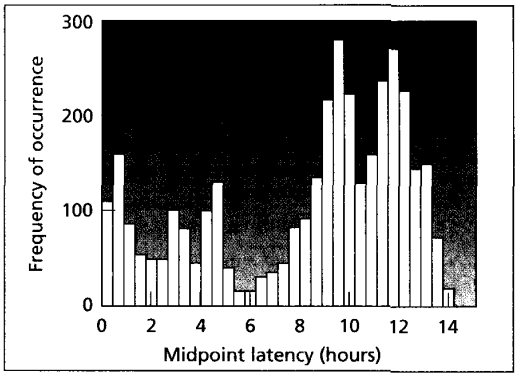

Figure 3. Error latency distribution for a fault at midnight.

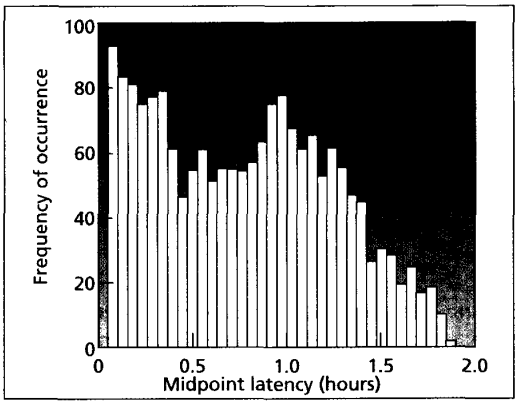

Figure 4. Error latency distribution for a fault at noon.

Table 5. Completion category distributions.

Completion category	Matrix multiplication (percent)	Recursive Fibonacci computation (percent)
Overwritten	64	71
Fatal errors	17	8
Time-outs	7	7
Results wrong	8	8
Results OK	4	6

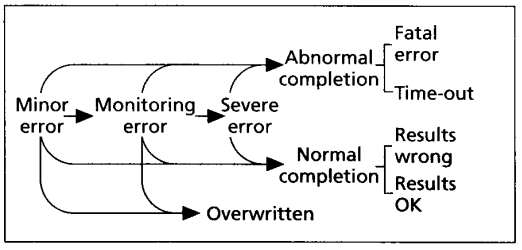

Figure 5. Fault manifestation and error propagation.

at high work load. Notice that a fault occurring at midnight was likely to remain dormant until the sharp increase in work load beginning at 8 a.m., whereas a fault at noon had a high probability of being detected quickly. This clearly demonstrated that error latency strongly depends on the work load following the fault's occurrence.

Impacts of faults on program behavior

Czeck and Siewiorek employed simulated fault injection to study the effects of gate-level faults on program behavior in the IBM RT PC.[5] They exhaustively injected one-cycle inversion faults into 10 key CPU locations across the entire execution time of a matrix multiplication and a recursive Fibonacci program. They incorporated several different error-detection mechanisms (EDMs) into this processor's simulation model. Figure 5 illustrates possible fault manifestations and error propagation to the EDMs. An injected fault initially caused a minor error. If the minor error later propagated to and was detected by an EDM, it became a monitoring error. A severe error occurred when a monitoring error disrupted control flow. The program would then either complete with correct or incorrect results or terminate through a time-out or fatal error. Table 5 reports the outcomes for both work loads. Of the 18,900 transients injected, 60 to 70 percent were inserted into idle hardware in the processor and eventually overwritten. Of those faults that were not overwritten, approximately 30 to 40 percent lead to normal program completion, while over 60 percent produced severe errors. Czeck and Siewiorek later developed a model predicting faulty system behavior from work load attributes such as instruction type, control flow structure, and instruction mix, based on these experimental results.

Failure acceleration

Chillarege and Bowen introduced the concept of failure acceleration to increase the speed at which a system transitions between the good, erroneous, and failed states during fault-injection experiments.[6] They accomplished this by decreasing fault and error latency and increasing the probability of a fault causing a failure, without altering the fault model. The idea was utilized in a study involving 70 experimental runs that filled a random page of real storage in an IBM 3081 mainframe with bytes of hexadecimal FF. This faulty bit pattern emulates the effects of a software overlay, which arises when a program writes into an incorrect storage area. During the experiment, the system executed simulated on-line database transactions that kept CPU utilization between 85 and 90 percent. The resulting state transition diagram (depicted in Figure 6) indicates that only 16 percent of the injected faults caused the system to quickly crash. One third of the observations were classified as partial failures, representing some loss in service without any adverse effect on the primary application. In 51 percent of the experimental runs, nothing happened within 15 minutes of the fault injection. Roughly half of these responses were later identified as potential hazards, or errors that had caused significant damage to the system but—under the prevailing operating state—would remain dormant. There was adequate time to repair 60 percent of the errors that did not affect

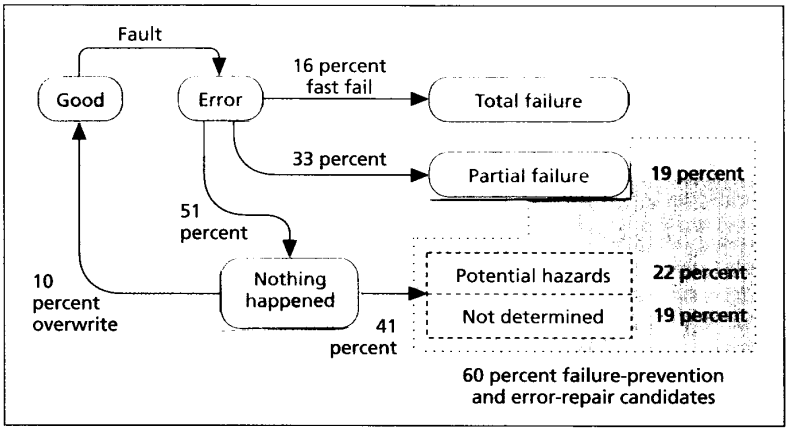

Figure 6. State transitions under failure acceleration. All faults produce errors. 16 percent cause failure quickly, and 33 percent cause a partial failure—with 19 percent being partial failures that are failure-prevention or error-repair candidates. For 51 percent of faults, nothing happens—with 10 percent being overwritten and 41 percent remaining as potential hazards or "not determined." 60 percent of all faults are failure-prevention or error-repair candidates.

the short-term availability of the system. Chillarege and Bowen discussed failure prevention and error repair techniques to detect and remove these errors and avert the loss of primary service.

Transient errors impact availability

Goswami and Iyer explored the impact of latent and correlated transient errors on a commercial fault-tolerant system's availability.[7] The target for this study was the triple-modular redundant (TMR) processing core of the Tandem Integrity S2. Processor modules are triplicated in this machine, and a majority voter masks erroneous outputs from any one processor. Goswami and Iyer used the Depend tool to inject transients into a functional-level simulation of the system's CPUs and memories. They simulated system operation 10 to 60 times, over periods of up to 200 years, to obtain statistically significant MTBF estimates. They considered three different error arrival rates ($\lambda_1 = 1/24$ hours, $\lambda_2 = 1/72$ hours, and $\lambda_3 = 1/120$ hours) and latencies, based on the analysis of real error data collected from other systems. The results graphed in Figure 7 show that latent transients alone did not adversely affect the system's MTBF. However, when 85 percent of the injected errors were correlated by even a small percentage, the degradation in MTBF was enormous. To sustain a high MTBF in the presence of latent errors, Goswami and Iyer suggested frequent memory scrubbing and reducing the time required for a CPU power-on self-test. In other experiments, they measured the coverage and latency of two memory-scrubbing schemes running under a simulated application program.

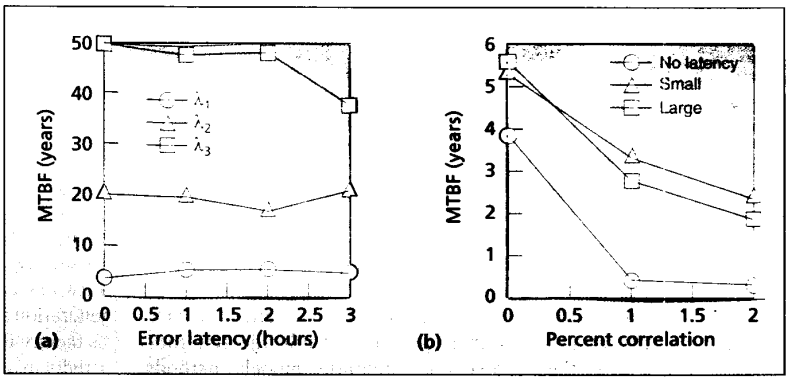

Figure 7. Effect of latent and correlated errors on MTBF (mean time between failures): (a) uncorrelated latent errors; (b) correlated latent errors.

Evaluating proposed designs

The studies discussed so far focused on validating existing systems, but fault injection can also evaluate the dependability of proposed designs. We have used simulated fault injection to analyze the reliability of several alternative TMR architectures.[8] Bidirectional voting (BDV) on both memory read and write accesses is typically performed in TMR systems. We proposed read-only voting (ROV) and write-only voting (WOV) to reduce the voting performance penalty through a small sacrifice in reliability. We used the React (Reliable Architecture Character-ization Tool) fault-injection testbed to empirically compare these three different designs. React simulated each TMR system's processors, memories, and voter at the functional level. The processors executed a synthetic work load, while permanent and transient faults were injected into the system components at exponentially distributed interarrival times. Figure 8 shows the reliability/performance tradeoff obtained via unidirectional voting. One million TMR systems of each type were simulated over a 100-hour mission to generate these plots. For equal processor and memory module failure rates (λ_p and λ_M, respectively) in the upper plot, the reliability was significantly higher for BDV than for either the ROV or WOV architecture. However, when the memory failure rate was 10 times greater than the processor failure rate, the difference between the reliability curves shrank in the lower plot. Our results indicate that in many cases, the unidirectional-voting TMR systems give up a little reliability for a potentially large increase in performance.

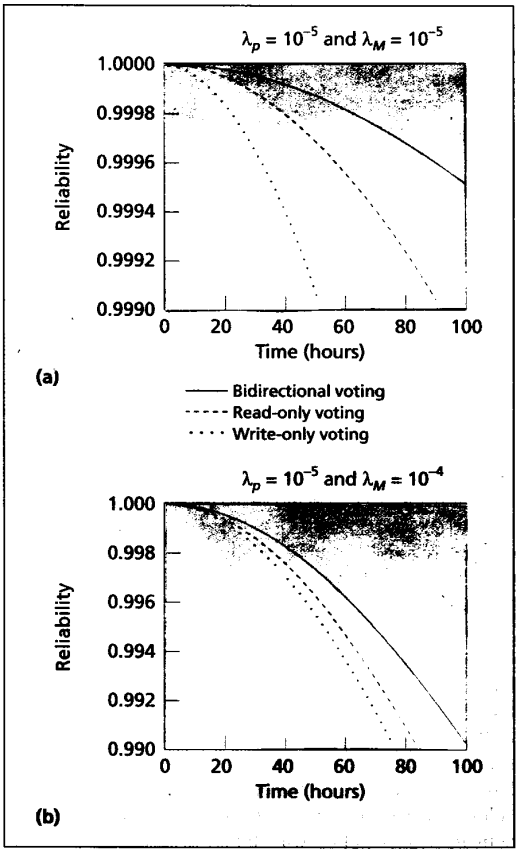

Figure 8. Reliability trade-off of the alternative triple-modular redundant (TMR) designs: (a) equal processor and memory module failure rates ($\lambda_p = \lambda_M = 10^{-5}$ failures/hour); (b) memory module failure rate greater than processor module failure rate.

FAULT-INJECTION TOOLS

Most fault-injection experiments were not designed around a formalized methodology. Experimenters typically developed customized approaches to validate each new system. This makes it difficult to apply specific results from different studies when analyzing other systems. Moreover, the complexity of today's systems can make the fault space (defined as fault type × location × injection time) huge. This means many experiments must be performed to achieve statistical confidence in the dependability metric being measured. To obtain the most accurate results in the shortest time, we must accelerate the injection and measurement processes. Fault-injection tools address these problems by integrating models, methods, and measurements into a generalized framework for conducting automated experiments on a variety of systems.

Messaline

Various fault-injection tools can evaluate physical systems, but few offer the versatility of Messaline, which was developed by LAAS-CNRS (Laboratory for the Analysis of System Architectures at the National Center for Scientific Research), France.[9] Its design is based on a formalized fault-injection methodology. The result is a flexible testbed capable of simultaneously injecting multiple, pin-level faults into different target systems to collect coverage, latency, and error-propagation measurements. A host computer manages fault injection by generating the test sequence, providing runtime execution control, and archiving data for analysis. Messaline has validated a centralized computer interlocking system for railway control and the distributed communication system of the Esprit Delta-4 project.

Fiat

The Fault-Injection-Based Automated Testing environment combines the flexibility of software control with hardware emulation, to evaluate the dependability of fault-tolerant distributed systems.[10] Fiat uses software-implemented fault injection to (erroneously) set and clear bytes in the memory images of programs. The programs execute on a network of machines configured to model a particular system architecture. This tool was realized with four IBM RT PCs connected via a token ring at Carnegie Mellon University. Fiat has been used to measure coverage and latency, classify failures, and investigate the effects of fault type and work load on these metrics.

Ferrari

The Fault and Error Automatic Real-Time Injector was designed at the University of Texas to estimate the coverage and latency of fault-tolerance mechanisms.[11] Like Fiat, it uses software-implemented injection to emulate hardware faults. However, instead of injecting errors directly into memory, Ferrari traps instructions affected by the fault so that a routine can be executed to mimic system behavior in the presence of the real fault. Various permanent and transient hardware faults, program control-flow errors, and user-defined faults/errors can be injected. Running on a Sun SparcStation under X Windows, Ferrari has evaluated the effectiveness of several concurrent error-detection techniques embedded in application software.

Focus

The Focus simulation environment conducts fault sensitivity experiments on chip-level designs.[1] Transient faults are injected through a runtime modification of the circuit, whereby a time-dependent current source is added to a device-level node. The current source deposits excess charge on this node to represent the penetration of an alpha particle or other electrical disturbance. The software provides various statistical measures to quantify fault sensitivity, including charge thresholds, error distributions, and two state-transition models that describe error generation and propagation. Focus uses a graphical analysis facility for Sun workstations, letting it visualize fault activity in a chip's functional units and error propagation on the major interconnects to external pins. Focus was developed at the University of Illinois and was used to analyze a dual-channel jet-engine controller.

Depend

The Depend environment is a joint dependability and performability evaluation tool that analyzes fault-tolerant architectures at the system level.[7] This process-based simulator provides a library of objects to behaviorally model a system's hardware components. Using these objects, a control program written in C++ simulates system operation and models system software. The objects automatically inject faults, initiate repairs, and compile statistics—such as the number of failures per component and the component's MTBF—that can be graphically displayed or included in a report. Permanent, transient, and user-defined faults can be injected with latency or at correlated times. A fault-injection scheme based on work load is also available. Depend was developed at the University of Illinois and has been used to analyze the Tandem Integrity S2 commercial fault-tolerant processor and a load-sharing distributed system.

React is a software testbed that abstracts multiprocessor systems at the architectural level.

React

In a cooperative effort between the University of Massachusetts and Texas A&M, our group has produced the Reliable Architecture Characterization Tool.[12] React is a software testbed that abstracts multiprocessor systems at the architectural level. It performs life testing through simulated fault injection to measure dependability. This involves conducting a statistically significant number of experiments or trials, each simulating the operation of an initially fault-free system. Randomly occurring faults are injected into each system until it fails or reaches a specified censoring time. Failure statistics are collected during each trial and are later aggregated over the entire simulation run to compute dependability metrics.

We have incorporated detailed system, work load, and fault/error models into the React software. Figure 9 depicts the system model employed by React. This class of architectures contains one or more processor modules (P)

interconnected via buses (B) to one or more memory modules (M) through a block of fault-tolerance mechanisms. The fault-tolerance mechanisms supply the hardware necessary to detect, correct, or mask errors during memory accesses and to reconfigure the system when modules fail. This framework provides the flexibility needed to represent many different architectures without requiring custom simulation models for each one. React can analyze multiprocessor systems that use *N*-modular redundancy, duplication and comparison, standby sparing, or error-control coding to achieve fault tolerance.

React assumes a synthetic work load. Processors continually perform instruction cycles consisting of several possible memory references and the simulated execution of an instruction. React does not use real application code and data, but allows errors to propagate throughout the system as if the software were actually being executed. The work load model is specified by a mean instruction execution rate, the probabilities of performing a memory read and write access per instruction, and a locality-of-reference model that determines which locations are accessed. These parameters can easily be extracted from memory reference traces collected during application software development.

Permanent and transient faults can be automatically injected into a system's processors, memories, and fault-tolerance mechanisms. Fault occurrence times are sampled from a Weibull distribution. Faults affect a processor's data and control paths and a memory's bit-array and addressing logic. Each faulty component's erroneous behavior is governed by a stochastic model that accounts for both fault and error latency. We derived these stochastic models from the results of other low-level fault-injection experiments. Repair times for failed components are assumed to have a log-normal distribution after a fixed logistics delay. The time required to reintegrate a repaired component back into the system and the time to reboot the system after a critical failure are constant and user specified.

We demonstrated the effectiveness of React by analyzing several alternative multiprocessor architectures. Specifically, we investigated two dependability tradeoffs associated with triple-modular redundant (TMR) systems. The first study explored the reliability/performance tradeoff in voting unidirectionally instead of bidirectionally on either memory read or write accesses. The second study examined the reliability/cost tradeoff in duplicating and comparing (via error-detecting codes) the memory modules rather than triplicating and voting on those modules. Both studies showed that a small sacrifice in reliability can be made for potentially large performance increases or cost reductions compared to traditional TMR design.

FAULT INJECTION HAS BECOME A VALUABLE ASSET for evaluating computer system dependability. It has been used to obtain analytical-model parameters, validate existing fault-tolerant systems, and synthesize more reliable system designs. However, many problems remain.

One challenge is to reduce the large fault space associated with highly integrated systems. This will require improved sampling techniques and models that equivalently represent the effects of low-level faults at higher abstraction levels. The impact of specification and design faults, particularly in software, is another largely unexplored problem. A better understanding of their occurrence is necessary before we consider injecting specification and design faults to validate computer-system dependability. Another obstacle is the difficulty in controlling the injection of environmentally induced faults. In addition, little is known about the relationship between faults in-jected in a laboratory and those actually occurring in the field.

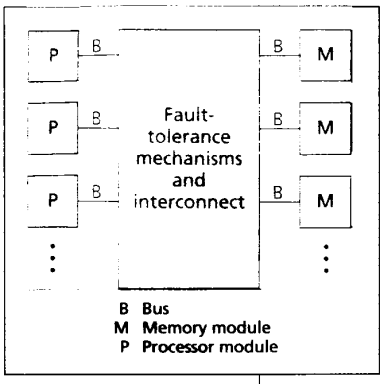

Figure 9. Class of architectures that React can analyze.

Finally, most fault-injection experiments are essentially case studies of particular systems. We must develop ways of generalizing machine-specific results to expand their applicability to other systems. Growing dependence on computers in life- and cost-critical applications makes this essential. ■

Acknowledgments

This material is partially based on work supported by the Texas Advanced Technology Program under grant 999903-029 and by the Air Force Office of Scientific Research under grant F49620-94-1-0276.

References

1. G.S. Choi and R.K. Iyer, "Focus: An Experimental Environment for Fault Sensitivity Analysis," *IEEE Trans. Computers*, Vol. 41, No. 12, Dec. 1992, pp. 1,515-1,526.
2. J. Karlsson et al., "Two Fault-Injection Techniques for Test of Fault-Handling Mechanisms," *Proc. Int'l Test Conf.*, IEEE CS Press, Los Alamitos, Calif., Order No. 2156, 1991, pp. 140-149.
3. R. Chillarege and R.K. Iyer, "An Experimental Study of Memory Fault Latency," *IEEE Trans. Computers*, Vol. 38, No. 6, June 1989, pp. 869-874.
4. R. Chillarege and R.K. Iyer, "Measurement-Based Analysis of Error Latency," *IEEE Trans. Computers*, Vol. C-36, No. 5, May 1987, pp. 529-537.
5. E.W. Czeck and D.P. Siewiorek, "Effects of Transient Gate-Level Faults on Program Behavior," *Proc. 20th Int'l Symp. Fault-Tolerant Computing*, IEEE CS Press, Los Alamitos, Calif., Order No. 2051, 1990, pp. 236-243.
6. R. Chillarege and N.S. Bowen, "Understanding Large-System Failures: A Fault-Injection Experiment," *Proc. 19th Int'l Symp. Fault-Tolerant Computing*, IEEE CS Press, Los Alamitos, Calif., Order No. 1959, 1989, pp. 356-363.
7. K.K. Goswami and R.K. Iyer, "A Simulation-Based Study of a Triple-Modular Redundant System Using Depend," *Proc. Fifth Int'l Conf. Fault-Tolerant Computing Systems*, IEEE Press, Piscataway, N.J., 1991, pp. 300-311.
8. J.A. Clark and D.K. Pradhan, "Reliability Analysis of Unidi-

rectional Voting TMR Systems Through Simulated Fault Injection," *Proc. 1992 Workshop Fault-Tolerant Parallel and Distributed Systems*, IEEE CS Press, Los Alamitos, Calif., Order No. 2871, 1992, pp. 72-81.
9. J. Arlat et al., "Fault Injection for Dependability Validation: A Methodology and Some Applications" *IEEE Trans. Software Engineering*, Vol. 16, No. 2, Feb. 1990, pp. 166-182.
10. Z. Segall et al., "Fiat: Fault-Injection-Based Automated Testing Environment," *Proc. 18th Int'l Symp. Fault-Tolerant Computing*, IEEE CS Press, Los Alamitos, Calif., Order No. 867, 1988, pp. 102-107.
11. G.A. Kanawati, N.A. Kanawati, and J.A. Abraham, "Ferrari: A Tool for the Validation of System Dependability Properties," *Proc. 22nd Int'l Symp. Fault-Tolerant Computing*, IEEE CS Press, Los Alamitos, Calif., Order No. 2876, 1992, pp. 336-344.
12. J.A. Clark and D.K. Pradhan, "React: A Synthesis and Evaluation Tool for Fault-Tolerant Multiprocessor Architectures," *Proc. 1993 Annual Reliability and Maintainability Symp.*, IEEE Press, Piscataway, N.J., 1993, pp. 428-435.

Jeffrey A. Clark *is a member of the technical staff in the Reliability and Maintainability Center at the Mitre Corporation. His research interests include fault-tolerant computing, parallel processing, and system dependability modeling and simulation. He received a BS in electrical and computer engineering (ECE) from Carnegie Mellon University in 1987, and an MS and PhD in ECE from the University of Massachusetts at Amherst in 1989 and 1993, respectively. He is a member of the IEEE Computer Society and the Reliability Society.*

Dhiraj K. Pradhan *holds the College of Engineering Endowed Chair in computer science at Texas A&M University. He has been actively involved in research of fault-tolerant computing, parallel processing, and VLSI testing over the last 20 years, presenting and publishing numerous papers. He has served as guest editor for special issues on fault-tolerant computing in* IEEE Transactions on Computers *and* Computer, *and is an editor for several journals, including* IEEE Transactions on Computers *and* JETTA. *Pradhan has also served as general chair for the 22nd Fault-Tolerant Computing Symposium and as program chair for the IEEE VLSI Test Symposium. He is a fellow of the IEEE and a recipient of the Humboldt Distinguished Scientist Award.*

Readers can contact Clark at the Mitre Corporation, Mailstop H113, 202 Burlington Road, Bedford, MA 01730; e-mail jeclark@mbunix.mitre.org; and Pradhan at the Computer Science Department, H.R. Bright Building, Texas A&M University, College Station, TX 77843; e-mail pradhan@cs.tamu.edu.