NASA-CR-204801

Fault Injection Techniques and Tools

Fault injection is important to evaluating the dependability of computer systems. Researchers and engineers have created many novel methods to inject faults, which can be implemented in both hardware and software.

Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer

University of Illinois at Urbana-Champaign

Dependability evaluation involves the study of failures and errors. The destructive nature of a crash and long error latency make it difficult to identify the causes of failures in the operational environment. It is particularly hard to recreate a failure scenario for a large, complex system.

To identify and understand potential failures, we use an experiment-based approach for studying the dependability of a system. Such an approach is applied not only during the conception and design phases, but also during the prototype and operational phases.[1,2]

To take an experiment-based approach, we must first understand a system's architecture, structure, and behavior. Specifically, we need to know its tolerance for faults and failures, including its built-in detection and recovery mechanisms,[3] and we need specific instruments and tools to inject faults, create failures or errors, and monitor their effects.

DIFFERENT PHASES, DIFFERENT TECHNIQUES

Engineers most often use low-cost, simulation-based *fault injection* to evaluate the dependability of a system that is in the conceptual and design phases. At this point, the system under study is only a series of high-level abstractions; implementation details have yet to be determined. Thus the system is simulated on the basis of simplified assumptions.

Simulation-based fault injection, which assumes that errors or failures occur according to predetermined distribution, is useful for evaluating the effectiveness of fault-tolerant mechanisms and a system's dependability; it does provide timely feedback to system engineers. However, it requires accurate input parameters, which are difficult to supply: Design and technology changes often complicate the use of past measurements. Testing a prototype, on the other hand, allows us to evaluate the system without any assumptions about system design, which yields more accurate results. In *prototype-based fault injection*, we inject faults into the system to

- identify dependability bottlenecks,
- study system behavior in the presence of faults,
- determine the coverage of error detection and recovery mechanisms, and
- evaluate the effectiveness of fault tolerance mechanisms (such as reconfiguration schemes) and performance loss.

To do prototype-based fault injection, faults are injected either at the hardware level (logical or electrical faults) or at the software level (code or data corruption) and the effects are monitored. The system used for evaluation can be either a prototype or a fully operational system. Injecting faults into an operational system can provide information about the failure process. However, fault injection is suitable for studying emulated faults only. It also fails to provide dependability measures such as mean time between failures and availability.

Instead of injecting faults, engineers can directly measure operational systems as they handle real workloads.[2] *Measurement-based analysis* uses actual data, which contains much information about naturally occurring errors and failures and sometimes about recovery attempts. Analyzing these data can provide understanding of actual error and failure characteristics and insight for analytical models. However, measurement-based analysis is limited to detected errors. Furthermore, data must be collected over a long time because errors and failures occur

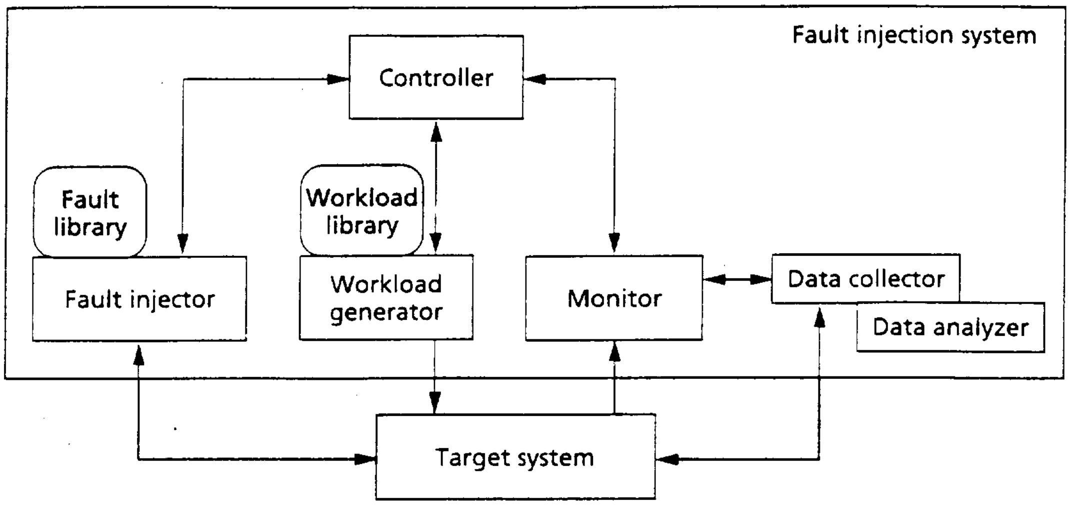


Figure 1. Basic components of a fault injection environment.

infrequently. Field conditions can vary widely, thus casting doubt on the statistical validity of the result.

Although each of the three experimental methods has its limitations, their unique values complement one another and allow for a wide spectrum of dependability studies.

FAULT INJECTION TECHNIQUES

Engineers use fault injection to test fault-tolerant systems or components. Fault injection tests fault detection, fault isolation, and reconfiguration and recovery capabilities.

Fault injection environment

Figure 1 shows a fault injection environment, which typically consists of the target system plus a fault injector, fault library, workload generator, workload library, controller, monitor, data collector, and data analyzer.

The fault injector injects faults into the target system as it executes commands from the workload generator (applications, benchmarks, or synthetic workloads). The monitor tracks the execution of the commands and initiates data collection whenever necessary. The data collector performs online data collection, and the data analyzer, which can be offline, performs data processing and analysis. The controller controls the experiment.

Physically, the controller is a program that can run on the target system or on a separate computer. The fault injector can be custom-built hardware or software. The fault injector itself can support different fault types, fault locations, fault times, and appropriate hardware semantics or software structure—the values of which are drawn from a fault library. The fault library in Figure 1 is a separate component, which allows for greater flexibility and portability. The workload generator, monitor, and other components can be implemented the same way.

Injection method and implementation

Choosing between hardware and software fault injection depends on the type of faults you are interested in and the effort required to create them. For example, if you are interested in *stuck-at faults* (faults that force a permanent value onto a point in a circuit), a hardware injector is preferable because you can control the location of the fault. The injection of permanent faults using software methods either incurs a high overhead or is impossible, depending on the fault. However, if you are interested in data corruption, the software approach might suffice. Some faults, such as bit-flips in memory cells, can be injected by either method. In a case like this, additional requirements, such as cost, accuracy, intrusiveness, and repeatability may guide the choice of approach. Table 1 summarizes commonly studied faults and injection methods.

HARDWARE FAULT INJECTION

Hardware-implemented fault injection uses additional hardware to introduce faults into the target system's hardware. Depending on the faults and their locations, hardware-implemented fault injection methods fall into two categories:

- *Hardware fault injection with contact.* The injector has direct physical contact with the target system, producing voltage or current changes externally to the target chip. Examples are methods that use pin-level probes and sockets.
- *Hardware fault injection without contact.* The

injector has no direct physical contact with the target system. Instead, an external source produces some physical phenomenon, such as heavy-ion radiation and electromagnetic interference, causing spurious currents inside the target chip.

These methods are well suited for studying the dependability characteristics of prototypes that require high time-resolution for hardware triggering and monitoring (fault latency in the CPU, for example) or require access to locations that cannot be easily reached by other fault injection methods.

Engineers generally model hardware methods on low-level fault models; for example, a *bridging fault* might be a short circuit. Hardware also triggers faults and monitors their impact, thus providing high time-resolution and low perturbation. Normally, the hardware triggers faults after a specified time has expired on a *hardware timer* or after it has detected an event, such as a specified address on the address bus.

Injection with contact

Hardware fault injection using direct contact with circuit pins, often called *pin-level injection*, is probably the most common method of hardware-implemented fault injection. There are two main techniques for altering electrical currents and voltages at the pins:

- *Active probes.* This technique adds current via the probes attached to the pins, altering their electrical currents. The probe method is usually limited to stuck-at faults, although it is possible to attain bridging faults by placing a probe across two or more pins. Care must be taken when using active probes to force additional current into the target device, as an inordinate amount of current can damage the target hardware.
- *Socket insertion.* This technique inserts a socket between the target hardware and its circuit board. The inserted socket injects stuck-at, open, or more complex logic faults into the target hardware by forcing the analog signals that represent desired logic values onto the pins of the target hardware. The pin signals can be inverted, ANDed, or ORed with adjacent pin signals or even with previous signals on the same pin.

Both of these methods provide good controllability of fault times and locations with little or no perturbation to the target system. Note that because faults are modeled at the pin level, they are not identical to traditional stuck-at and bridging fault models that generally occur inside the chip. Nonetheless, you can achieve many of the same effects, like the exercise of error detection circuits, using these injection methods. Active probes attached to the power supply hardware inject power supply disturbance faults. However, this can damage the injected device or increase the risk of destructive injection.

Table 1. Fault-injection implementation methods by fault model.

Hardware	Software
Open	Storage data corruption
Bridging	(such as register, memory, and disk)
Bit-flip	Communication data corruption
Spurious current	(such as bus and communication network)
Power surge	Manifestation of software defects
Stuck-at	(such as machine level and higher levels)

Injection without contact

These faults are injected by creating heavy-ion radiation. An ion passes through the depletion region of the target device and generates current. Placing the target hardware in or near an electromagnetic field also injects faults. Engineers like these methods because they mimic natural physical phenomena. However, it is difficult to exactly trigger the time and location of a fault injection using this technique because you cannot precisely control the exact moment of heavy-ion emission or electromagnetic field creation.

Selected tools

Messaline,[4] developed at LAAS-CNRS, in Toulouse, France, uses both active probes and sockets to conduct pin-level fault injection. Figure 2 on the next page shows Messaline's general architecture and its environment. Messaline can inject stuck-at, open, bridging, and complex logical faults, among others. It can also control the length of fault existence and the frequency. Signals collected from the target system can provide feedback to the injector. Also, a device is associated with each injection point to sense when and if each fault is activated and produces an error. It can also inject up to 32 injection points simultaneously. This tool has been used in experiments on a centralized, interlocking system employed in a computerized railway control system and on a distributed system for the Esprit Delta-4 Project.

FIST[5] (Fault Injection System for Study of Transient Fault Effect), developed at the Chalmers University of Technology in Sweden, employs both contact and contactless methods to create transient faults inside the target system. This tool uses heavy-ion radiation to create transient faults at random locations inside a chip when the chip is exposed to the radiation and can thus cause single- or multiple-bit-flips. The radi-

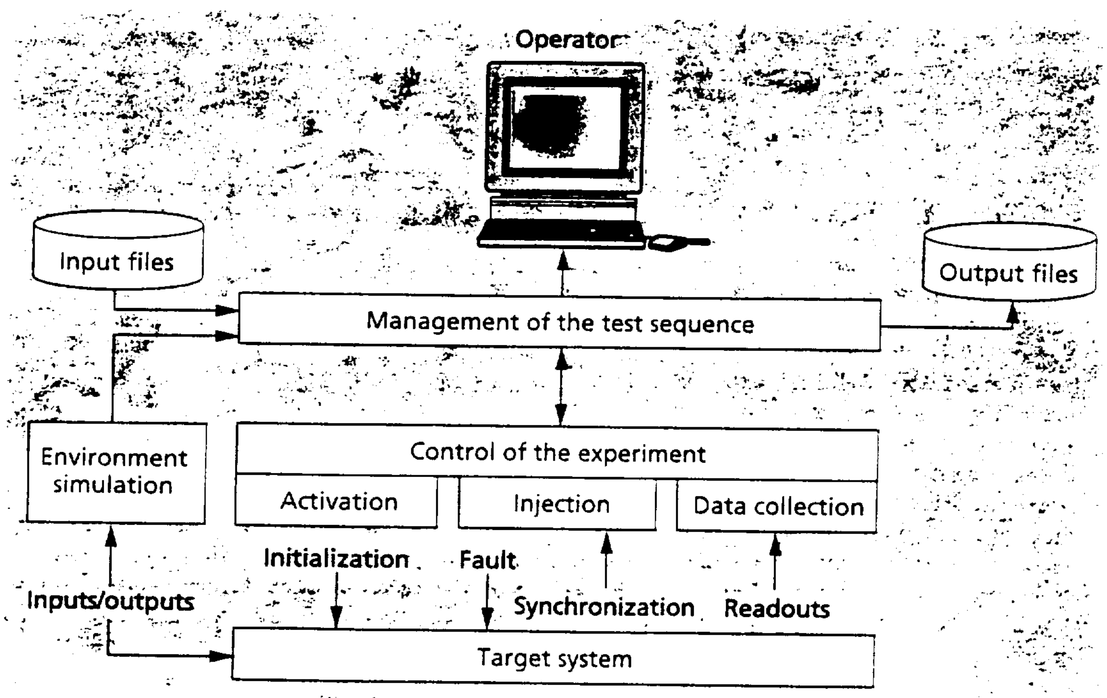


Figure 2. General architecture of Messaline.

ation source is mounted inside a vacuum chamber together with a small two-processor computer system. The computer is positioned so that one of the processors is exposed directly under the radiation. The other processor is used as a reference for detecting whether the radiation results in any bit-flips. Figure 3 illustrates the FIST environment.

FIST can inject faults directly inside a chip, which cannot be done with pin-level injections. It can produce transient faults at random locations evenly in a chip, which leads to a large variation in the errors seen on the output pins. In addition to radiation, FIST allows for the injection of power disturbance faults. This is done by placing a MOS transistor between the power supply and the V_{CC} pin of the processor chip to control the amplitude of the voltage drop. Power supply disturbances usually affect multiple locations within a chip and can cause gate propagation delay faults. The experimental results show that the errors resulting from both methods cause similar effects on program control-flow and data errors. However, heavy-ion radiation causes mostly address bus errors, while power supply disturbances affect mostly control signals.

MARS[6] (Maintainable Real-Time System) is a distributed, fault-tolerant architecture developed at the Technical University of Vienna. In addition to using heavy-ion radiation as is used in FIST, MARS uses electromagnetic fields to conduct contactless fault injection: A circuit board placed between two charged plates or a chip placed near a charged probe causes fault injection. Dangling wires that act as antennas placed on individual chip pins accentuate the electromagnetic field effect on those pins. Researchers compared these three methods (heavy-ion radiation, pin-level injection, and electromagnetic interference) in terms of their capability to exercise the MARS error detection mechanisms. Results showed that the three methods are complementary and generate different types of errors. Pin-level injections cause error detection mechanisms outside the CPU to be exercised more effectively than heavy-ion radiation or electromagnetic interference. The latter two methods were better suited for exercising software and application-level error detection mechanisms.

SOFTWARE FAULT INJECTION

In recent years, researchers have taken more interest in developing software-implemented fault injection tools. Software fault-injection techniques are attractive because they don't require expensive hardware. Furthermore, they can be used to target applications and operating systems, which is difficult to do with hardware fault injection.

If the target is an application, the fault injector is inserted into the application itself or layered between the application and the operating system. If the target is the operating system, the fault injector must be embedded in the operating system, as it is very difficult to add a layer between the machine and the operating system.

Although the software approach is flexible, it has its shortcomings.

- It cannot inject faults into locations that are inaccessible to software.
- The software instrumentation may disturb the workload running on the target system and even change the structure of original software. Careful design of the injection environment can minimize perturbation to the workload.

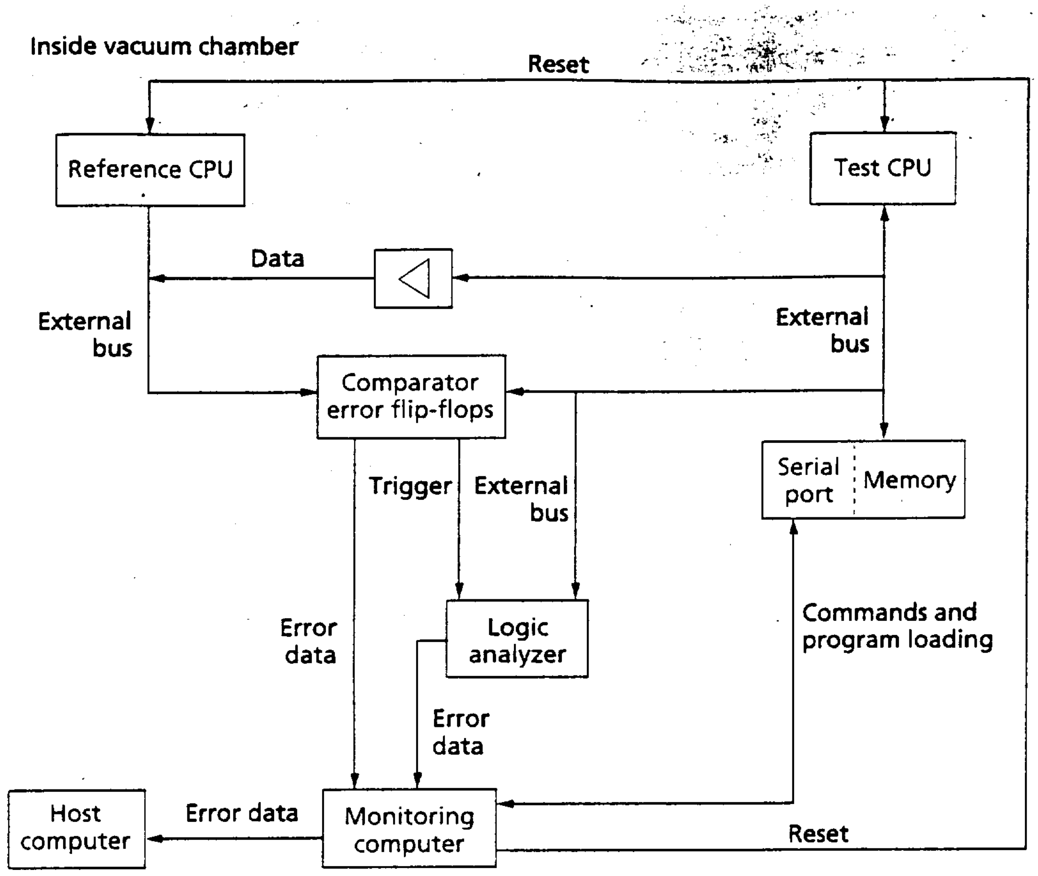


Figure 3. FIST environment.

- The poor time-resolution of the approach may cause fidelity problems. For long latency faults, such as memory faults, the low time-resolution may not be a problem. For short latency faults, such as bus and CPU faults, the approach may fail to capture certain error behavior, like propagation. Engineers can solve this problem by taking a *hybrid approach*, which combines the versatility of software fault injection and the accuracy of hardware monitoring. The hybrid approach is well suited for measuring extremely short latencies. However, the hardware monitoring involved can cost more and decrease flexibility by limiting observation points and data storage size.

We can categorize software injection methods on the basis of when the faults are injected: during compile-time or during runtime.

Compile-time injection

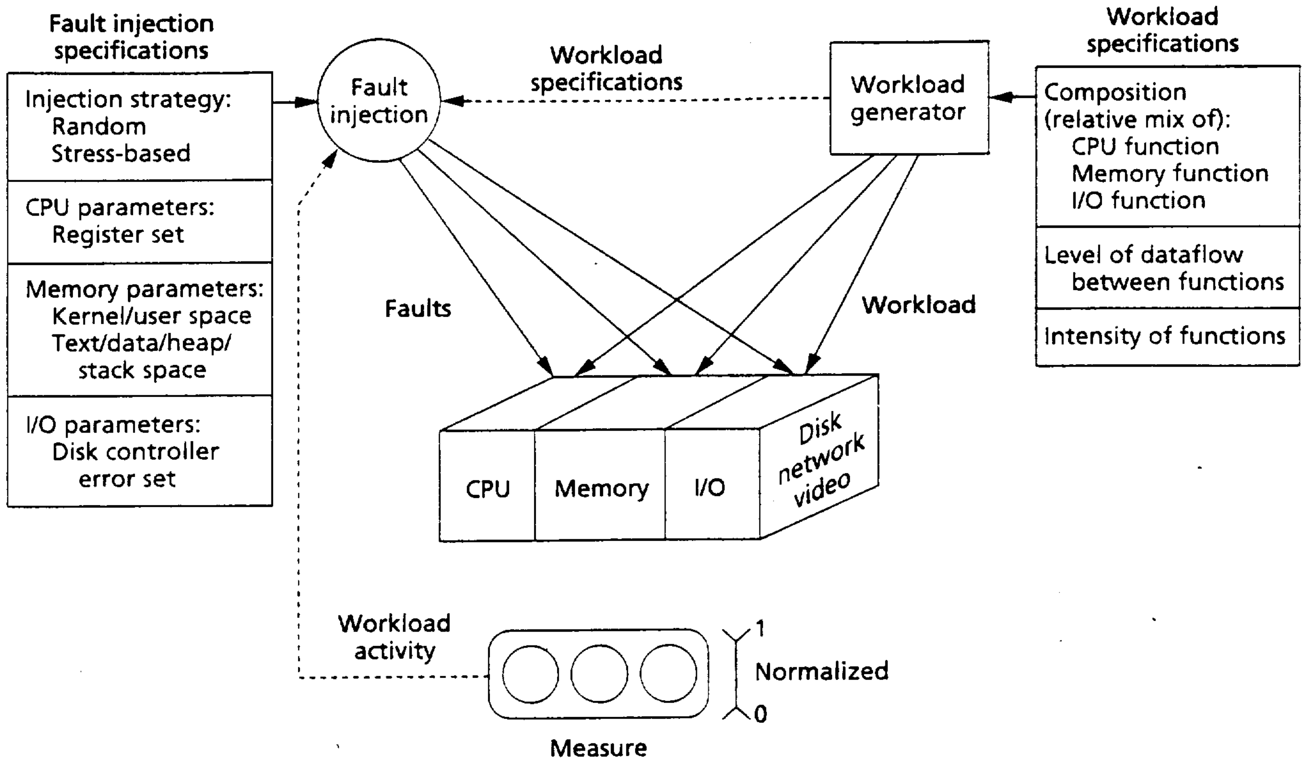
To inject faults at compile-time, the program instruction must be modified before the program image is loaded and executed. Rather than injecting faults into the hardware of the target system, this method injects errors into the source code or assembly code of the target program to emulate the effect of hardware, software, and transient faults. The modified code alters the target program instructions, causing injection. Injection generates an erroneous software image, and when the system executes the fault image, it activates the fault.

This method requires the modification of the program that will evaluate fault effect, and it requires no additional software during runtime. In addition, it causes no perturbation to the target system during execution. Because the fault effect is hard-coded, engineers can use it to emulate permanent faults. This method's implementation is very simple, but it does not allow the injection of faults as the workload program runs.

Runtime injections

During runtime, a mechanism is needed to trigger fault injection. Commonly used triggering mechanisms include:

- *Time-out.* In this simplest of techniques, a timer expires at a predetermined time, triggering injection. Specifically, the time-out event generates an `interrupt` to invoke fault injection. The timer can be a hardware or software timer. This method requires no modification to the application or workload program. A hardware timer must be linked to the system's interrupt handler vector. Since it injects faults on the basis of time rather than specific events or system state, it pro-



Fault injection specifications
Injection strategy: Random Stress-based
CPU parameters: Register set
Memory parameters: Kernel/user space Text/data/heap/ stack space
I/O parameters: Disk controller error set
Fault injection
Workload specifications
Workload generator
Workload specifications
Composition (relative mix of): CPU function Memory function I/O function
Level of dataflow between functions
Intensity of functions
Faults
Workload
CPU
Memory
I/O
Disk network video
Workload activity
1
Normalized
0
Measure

Figure 4. Ftape environment.

duces unpredictable fault effects and program behavior. It is, however, suitable for emulating transient faults and intermittent hardware faults.

- *Exception/trap.* In this case, a hardware exception or a software trap transfer control to the fault injector. Unlike time-out, exception/trap can inject the fault whenever certain events or conditions occur. For example, a software trap instruction inserted into a target program will invoke the fault injection before the program executes a particular instruction. When the trap executes, an interrupt is generated that transfers control to an interrupt handler. A hardware exception invokes injection when a hardware-observed event occurs (when a particular memory location is accessed, for example). Both mechanisms must be linked to the interrupt handler vector.
- *Code insertion.* In this technique, instructions are added to the target program that allow fault injection to occur before particular instructions, much like the code-modification method. Unlike code modification, code insertion performs fault injection during runtime and adds instructions rather than changing original instructions. Unlike the trap method, the fault injector may exist as part of the target program and run at user mode rather than system mode.

Selected tools

Ferrari[7] (Fault and Error Automatic Real-Time Injection), developed at the University of Texas at Austin, uses software traps to inject CPU, memory, and bus faults. Ferrari consists of four components: the initializer and activator, the user information, the fault-and-error injector, and the data collector and analyzer.

The fault-and-error injector uses software trap and trap handling routines. Software traps are triggered either by the program counter when it points to the desired program locations or by a timer. When the traps are triggered, the trap handling routines inject faults at the specific fault locations, typically by changing the content of selected registers or memory locations to emulate actual data corruptions. The faults injected can be those permanent or transient faults that result in an address line error, a data line error, and a condition bit error.

Experiments conducted on Sun SparcStations showed that error detection is highly dependent on the fault type. Faults in the task memory resulted in the highest level of detection, due mainly to the repeated injection of faults when trap instructions were placed in program loops. Also, many faults injected into I/O routines and system libraries went undetected because these routines were less frequently exercised.[7]

The Fault Tolerance and Performance Evaluator (Ftape),[8] developed at the University of Illinois, consists of the components shown in Figure 4. Engineers can inject faults into user-accessible registers in CPU modules, memory locations, and the disk subsystem.

Table 2. Characteristics of fault injection methods.

	Hardware		Software	
	With contact	Without contact	Compilation	Runtime
Cost	High	High	Low	Low
Perturbation	None	None	Low	High
Risk of damage	High	Low	None	None
Monitoring time resolution	High	High	High	Low
Accessibility of fault injection points	Chip pin	Chip internal	Register, memory, software	Register, memory, I/O controller/port
Controllability	High	Low	High	High
Trigger	Yes	No	Yes	Yes
Repeatability	High	Low	High	High

The faults are injected as bit-flips to emulate error as a result of faults.

Disk system faults are injected by executing a routine in the driver code that emulates I/O errors (bus error and timer error, for example). Fault injection drivers added to the operating system inject the faults, so no additional hardware or modification of application code is needed. A synthetic workload generator creates a workload containing specified amounts of CPU, memory, and I/O activity, and faults are injected with a strategy that considers the characteristics of the workload at the time of injection (which components are experiencing the greatest amount of workload activity, for example). Ftape has been used on several Tandem fault-tolerant computers and serves as the basis of a benchmark for fault tolerance, which measures the occurrence of system failures and the amount of performance degradation under fault conditions.

Doctor[9] (Integrated Software Fault Injection Environment), developed at the University of Michigan, allows injection of CPU faults, memory faults, and network communication faults. It uses three triggering methods—time-out, trap, and code modification—to trigger fault injection. Time-out triggers memory fault injection. Once time-out occurs, it triggers the fault injector to overwrite the memory content to emulate occurrence of a memory fault. For nonpermanent CPU faults, traps trigger fault injection. For permanent CPU faults, fault injection is done by changing program instructions during compilation to emulate instruction and data corruptions due to the faults. Doctor has been used on Harts, a distributed, real-time system, to investigate the effect of intermittent message losses between two adjacent nodes and the effect of routing using failure data. The researchers used experimental results to validate a message delivery model and to evaluate different message delivery methods.

Xception,[10] developed at the University of Coimbra in Portugal, takes advantage of the advanced debugging and performance monitoring features present in many modern processors to inject more realistic faults. It requires no modification in application software and no insertion of software traps. Xception, in fact, uses a processor's built-in hardware exception triggers to trigger fault injection. The fault injector is implemented as an exception handler and requires modification of the interrupt handler vector. Xception faults are triggered based on access to specific addresses (rather than on a time period following an event), so the experiments are reproducible. The following events can trigger fault injection:

- opcode fetch from a specified address,
- operand load from a specified address,
- operand store to a specified address,
- a specified time passed since start-up, and
- a combination of the above fault triggers.

Each fault has a specified *fault mask*: a set of bits that determines which corresponding bits in the target location will be injected. Bits in the fault mask set to 1 can use several bit-level operations: stuck-at-zero, stuck-at-one, bit-flip, and bridging. Xception has been implemented on a Parsytec parallel machine based on the PowerPC 601 processor. Experiments revealed the deficiency in the error detection mechanisms by showing that up to 73 percent of injected faults resulted in incorrect results that were undetected for certain processor functional units.

Table 2 classifies the hardware and software fault injection methods.

The contrast between the hardware and software methods lies mainly in the fault injection points they can access, cost, and level of perturbation. Hardware methods can inject faults into chip pins and internal components, such as combinational circuits and registers that are not software-addressable. On the other hand, software methods are convenient for directly producing changes at the software-state level (memory, register, for example). Thus, we use hardware methods to evaluate low-level error detection and masking mechanisms and software methods to test higher level mechanisms. Software methods are less expensive, but they also incur a higher perturbation overhead because they execute software on the target system. ❖

Acknowledgments

This research was supported in part by the US Department of Defense Advanced Research Projects Agency under contract DABT63-94-C-0045, ONR contract N00014-91-J-1116, and by NASA Grant NAG 1-613, in cooperation with the Illinois Computer Laboratory for Aerospace Systems and Software (ICLASS).

References

1. J.A. Clark and D.K. Pradhan, "Fault Injection: A Method for Validating Computing-System Dependability," *Computer*, June 1995, pp. 47-56.
2. D. Tang and R.K. Iyer, "Experimental Analysis of Computer System Dependability," in *Fault-Tolerant Computer System Design*, D.K. Pradhan, ed., Prentice-Hall Prof. Tech. Ref., Upper Saddle River, N.J., pp. 282-392.
3. J.A. Abraham, "Challenges in Fault Detection," *Proc. 25th Ann. Int'l Symp. Fault-Tolerant Computing*, IEEE CS Press, Los Alamitos, Calif., 1995, pp. 96-114.
4. J. Arlat, Y. Crouzet, and J.C. Laprie, "Fault Injection for Dependability Validation of Fault-Tolerant Computer Systems," *Proc. 19th Ann. Int'l Symp. Fault-Tolerant Computing*, IEEE CS Press, Los Alamitos, Calif., 1989, pp. 348-355.
5. O. Gunnetlo, J. Karlsson, and J. Tonn, "Evaluation of Error Detection Schemes Using Fault Injection by Heavy-ion Radiation," *Proc. 19th Ann. Int'l Symp. Fault-Tolerant Computing*, IEEE CS Press, Los Alamitos, Calif., 1989, pp. 340-347.
6. J. Karlsson, J. Arlat, and G. Leber, "Application of Three Physical Fault Injection Techniques to the Experimental Assessment of the MARS Architecture," *Proc. Fifth Ann. IEEE Int'l Working Conf. Dependable Computing for Critical Applications*, IEEE CS Press, Los Alamitos, Calif., 1995, pp. 150-161.
7. G.A. Kanawati, N.A. Kanawati, and J.A. Abraham, "FERRARI: A Tool for the Validation of System Dependability Properties," *Proc. 22nd Ann. Int'l Symp. Fault-Tolerant Computing*, IEEE CS Press, Los Alamitos, Calif., 1992, pp. 336-344.
8. T.K. Tsai and R.K. Iyer, "An Approach to Benchmarking of Fault-Tolerant Commercial Systems," *Proc. 26th Ann. Int'l Symp. Fault-Tolerant Computing*, IEEE CS Press, Los Alamitos, Calif., 1996, pp. 314-323.
9. S. Han, K.G. Shin, and H.A. Rosenberg, "Doctor: An Integrated Software Fault-Injection Environment for Distributed Real-Time Systems," *Proc. Second Annual IEEE Int'l Computer Performance and Dependability Symp.*, IEEE CS Press, Los Alamitos, Calif., 1995, pp. 204-213.
10. J. Carreira, H. Madeira, and J.G. Silva, "Xception: Software Fault Injection and Monitoring in Processor Functional Units," *Proc. Fifth Ann. IEEE Int'l Working Conf. Dependable Computing for Critical Applications*, IEEE CS Press, Los Alamitos, Calif., pp. 135-149.

Mei-Chen Hsueh is a visiting research associate professor at the Coordinated Science Laboratory in the University of Illinois at Urbana-Champaign. She is on a leave of absence from Digital Equipment Corporation. Her interests are in high-performance and dependable system design, fault-tolerance computing, and large complex system design. Hsueh received a PhD in computer science from the University of Illinois at Urbana-Champaign. She is a member of the IEEE and the IEEE Computer Society.

Timothy K. Tsai is a member of the technical staff at Bell Labs, Lucent Technologies. His research interests include fault-tolerant computing, software engineering, and distributed systems. Tsai received a BS in electrical engineering from Brigham Young University and an MS and a PhD in electrical engineering from the University of Illinois at Urbana-Champaign. He is a member of the IEEE and the IEEE Computer Society.

Ravishankar K. Iyer is a professor in the departments of Electrical and Computer Engineering and Computer Science, and at the Coordinated Science Laboratory, at the University of Illinois at Urbana-Champaign. He is also codirector of the Center for Reliable and High-Performance Computing and the Illinois Computing Laboratory for Aerospace Systems and Software, a NASA Center for Excellence in Aerospace Computing. Iyer's research interests are in the area of reliable computing, measurement and evaluation, and automated design. He is an associate fellow of the AIAA and a fellow of the IEEE.

Readers can contact Hsueh and Iyer at the Coordinated Science Laboratory, University of Illinois, Urbana, IL 61801; hsueh, iyer@crch.uiuc.edu. Contact Tsai at Lucent Technologies, 600 Mountain Ave., Murray Hill, NJ 07954; ttsai@research.bell-labs.com.