**FAULT INJECTION: A METHOD FOR VALIDATING FAULT-TOLERANT SYSTEM**

Salahuddin Mohammad Masum, Bhanu Chander Reddy Vanteru, Mohammed Yeasin  
*Dept. of Electrical and Computer Engineering, University of Memphis, USA*  
*Email: smasum@memphis.edu, bcvanter@memphis.edu, myeasin@memphis.edu*

**Abstract**  
Fault Injection is an effective solution to the problem of validating highly reliable systems. Fault Injection is the process of corrupting a data state during program execution. Fault injection based testing is the process of determining the effect of that corruption. The testing may consist of simply measuring whether the corrupted state affected a particular output, or the testing may determine whether system attributes such as safety, security, or survivability have been affected. Fault injection based testing is often used in large system development projects. This paper presents a fault injection technique that injects faults into computer system to measure system dependability. Our simulation shows the performance of the fault injection model.

**Keywords:** System Testing, Software Testing, Fault-tolerant System, Fault Injection.

1. **INTRODUCTION**

Testing is a crucial step in the development of a software-intensive system, as it checks the compliance of a system to the end user requirements [1]. The development of software testing systems must be performed in effective and efficient manner. It is easy to see that an effective testing is a very good indicator of the quality product and efficient testing procedure to ensure the faster development cycle that is an imperative requirement for large organization. The prime objective of the System Testing is to cover all forms of the testing techniques related to systems to ensure the successful development and application of software and technology.

With greater reliance on computers in a variety of applications, the consequences of failure and downtime have become more severe. Computers employed in critical 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.

This paper presents a system testing technique that injects faults into computer system to measure system dependability. The rest of this paper includes a discussion about fault, error, and failure in Section II; Section III presents different fault models and injection tools following the proposed fault injection method; Section IV shows the impacts of faults on program behavior in our simulation; At last, Section V concludes this paper.

2. **BACKGROUND - FAULT, ERROR, AND FAILURE**

A fault is a variation in a hardware or software component from its anticipated function. Faults can occur during all stages in a computer system’s evolution-specification, design, development, manufacturing, assembly, and installation-and throughout its operations. Most faults that take place before full system exploitation are discovered through testing and eliminated. Faults that are not detached can reduce a system’s consistency when it is in the field. In spite of the potential for such underlying faults in computer systems, most fault-injection reviews focus on the faults that occur during system operation [2].

Software faults are caused by the incorrect pattern, design, or coding of a program. Although software does not physically “break” after being installed in a computer system, underlying faults or bugs in the code can shell during operation especially under heavy or unusual work loads and eventually lead to system failures [2]. For this reason, software fault injection is employed mostly for testing programs or system or software implemented fault-tolerance mechanisms.
When a fault causes an invalid change in machine state, an error occurs. The time between fault occurrence and the first manifestation of an error is called the fault latency [2]. Although a fault remains localized in the affected code, multiple errors can originate from one fault site and propagate throughout the system. These will cause a propagating error after a period of time, called the error latency [2]. When the fault-tolerance mechanisms perceive an error, they may commence 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.

![Diagram of fault, error, and recovery states](image)

**Figure 1.** Example of a fault, an error, and a failure [2].

Figure 1 provides an example to clarify the definitions of fault, error, and failure [2]. Suppose a permanent stuck-at-0 (s-a-0) fault [2] affects a memory bit with an initial value of logical 0. Sometime 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.

3. **Fault Models and Injection Methods**

Simulated fault injection and most experiments involving substantial hardware and software require choice of a fault model. The popular stuck-at fault model is commonly used for permanent hardware/software faults. However, consequent errors often are of more significance than the faults themselves. This is particularly true for momentary faults, whose volatile origin and relatively short life span make them intricate to differentiate. 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. Some fault-injection models have been proposed in [3][4][5][6][7]. 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.

Trace Injection is a method that first uses custom-monitoring 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.

This paper presents a system testing technique that injects faults into computer system to measure system dependability. In next Section, we like to present our fault injection process and its impact on a particular code during simulation.

4. Fault Injection and Its Impacts on Software Behavior

Simulation of the Fault Injection has been performed on two different versions of Quick sort algorithm. First version is a very basic one which does not have any error or exception handling and the second one has some exception handling component. Theoretically, the second implementation must be more dependable than the first one. The simulation language is Java since it supports Exception Handling and Multi-threading. The pseudo-code for both the implementations is given in Figure 2.

The simulation is split into three threads: 1) Quick sort implementation thread, 2) Monitoring thread, 3) Fault Injection thread. All the threads in Simulation run in parallel at the execution time. Sleep methods are called in each of the threads at regular intervals so that other threads can resume. The Quick sort thread is not synchronized so that the monitoring and the fault injection threads can read and write variables (data) respectively during the course of execution of the Quick sort implementation thread. Faults are injected into the key variables of the software by the fault injection thread which runs in parallel with the implementation thread. The key variables are the pivot, low and the high. They denote the index numbers of the array in the Quick sort implementation. Quick sort uses recursion and so any change in these index values are carried to the later stages and can a potential hazard to the execution of the program.

(Basic implementation of Quick sort)

function qSort (q, low, high)  
var array less, pivotList, greater  
select a pivot value pivot from q  
for each x in q  
    if x < pivot then add x to less  
    if x = pivot then add x to pivotList  
    if x > pivot then add x to greater  
return concatenate(qSort(less), pivotList, qSort(greater))

(Advanced implementation of Quick sort)

function Quicksort(q, low, high)  
if (low < high)  
pivot-location = Partition(q, low, high)  
    Quicksort(q, low, pivot-location - 1)  
    Quicksort(q, pivot-location+1, high)  
else  
    throw Exception;

Partition(q, low, high)  
pivot = q[low]  
leftend = low  
for i = low+1 to high  
    if (q[i] < pivot) then  
        leftend = leftend+1  
        swap(A[i], q[leftend])  
        swap(q[low],q[leftend])  
    return (leftend)

Figure 2. Pseudocode for Simulation.

The input to the simulation is a random unsorted array of very large dimension (to put load on the system) and it is sorted using first the basic implementation and then with the advanced implementation of Quick sort. In both the cases, faults are injected into the variables (data) and the variables are monitored simultaneously. Table 1 gives the summary of the simulation results.
Table 1: Simulation Results.

<table>
<thead>
<tr>
<th>Description</th>
<th>Basic</th>
<th>Advanced</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of simulations</td>
<td>250</td>
<td>250</td>
</tr>
<tr>
<td>Number of faults injected</td>
<td>250*3</td>
<td>250*3</td>
</tr>
<tr>
<td>% Total failure</td>
<td>32%</td>
<td>19%</td>
</tr>
<tr>
<td>% Partial failure</td>
<td>27%</td>
<td>25%</td>
</tr>
<tr>
<td>% Nothing happened (Wrong output)</td>
<td>25%</td>
<td>34%</td>
</tr>
<tr>
<td>% Nothing happened (Correct output)</td>
<td>16%</td>
<td>32%</td>
</tr>
</tbody>
</table>

Total failure denotes the number of times the program terminated abnormally and also at the early stage of the implementation (these denote that the faults are fatal), partial failure means that the faults introduced minor errors which do not halt the program immediately. In other cases, the program executes successfully, but the output may be correct or wrong. From the results it can be noted that the second implementation of the Quicksort performs better than the first implementation. That means that the second version of the Quicksort is more dependable with respect to fault-tolerant concern.

In our extended phase of simulation, we injected faults into the entire execution time of two versions of Quicksort. We included an error-detection mechanism into this simulation model. An injected fault initially caused a minor error. If the minor error later propagated to and was detected by the model, it became a detected error. A fatal error occurred when a detected error disrupted control flow. The program would then either complete with correct or incorrect results or terminate through a time-out or fatal error.

Figure 3 and Figure 4 report the average outcomes for both work loads. Of the 500 faults injected into each version, 66% was inserted into idle CPU and eventually overwritten. Of those, faults that were not overwritten, approximately 39% leaded to normal program completion, while over 61% produced fatal errors.

5. CONCLUSION

In this paper, we present a Fault Injection method that can be an effective solution for computer system dependability. We present simulation results to validate two versions of Quicksort program for measuring the degree of fault-tolerance. However future investigations could be reducing the large fault space associated with highly integrated systems. This will require improved models that homogeneously symbolize the effects of low-level faults at higher rate.

6. REFERENCES

Figure 4: Category Distributions of Overwritten Errors.