FTAPE: A FAULT INJECTION TOOL TO MEASURE FAULT TOLERANCE

Timothy K. Tsai and Ravishankar K. Iyer*
University of Illinois
Center for Reliable and High Performance Computing
Coordinated Science Laboratory
Urbana, Illinois

Abstract

The paper introduces FTAPE (Fault Tolerance And Performance Evaluator), a tool that can be used to compare fault-tolerant computers. The tool combines system-wide fault injection with a controllable workload. A workload generator is used to create high stress conditions for the machine. Faults are injected based on this workload activity in order to ensure a high level of fault propagation. The errors/fault ratio and performance degradation are presented as measures of fault tolerance.

1 Introduction

A method with which the fault tolerance of any computer can be measured is desirable. Such a fault tolerance measure can be used to compare fault-tolerant computers. For purchasers of a new fault-tolerant system, a measure that summarizes the effectiveness and efficiency of the fault tolerance would be a helpful aid. A fault tolerance measure can also be used as feedback to engineers in the evaluation of different fault-tolerant designs.

This paper introduces FTAPE (Fault Tolerance and Performance Evaluator), a tool that characterizes the fault tolerance of a computer with a single measure. The tool combines a fault injector with a workload generator to encourage a high level of fault propagation, which is needed to thoroughly test the fault tolerance mechanism of the system. The fault injector measures the instantaneous workload activity to automatically determine the injection time and location that will maximize fault propagation.

Since the fault detection and recovery mechanisms that comprise the fault tolerance of a computer can only be activated by faults and their corresponding manifestations, fault injection is the most practical means to measure their effectiveness. FTAPE has the ability to inject faults into CPU registers, memory, and disk systems. Because fault tolerance mechanisms are present in many different parts of a fault-tolerant system, faults must be injected into those different parts in order to measure how well the entire system responds to faults.

A synthetic workload generator is used to provide an easily controllable workload that will aid fault propagation. Since fault propagation and the eventual effect of faults are dependent upon the accompanying workload, the ability to control the workload is essential to influencing fault propagation. With the synthetic workload generator, the workload can be specified to exercise the CPU, memory, or disk. The amount of stress that the workload places on each system component (i.e., CPU, memory, or disk) can be given as a distribution over time. Multiple workload processes can be executed.

Stress-based injection is the process of injecting faults based upon a measurement of the current workload activity. Stress in this sense refers to the amount of activity caused by the workload which could encourage fault propagation. The workload is measured and characterized in terms of the level of stress for each system component and the overall system at a particular time. The rate of fault injection is increased during times of greater overall system stress, and faults are injected into the system components with the highest stress level.

Since the main goal of the tool is to characterize the fault tolerance of the system using a single quantity, a metric for that characterization is needed. Several metrics are proposed and measured. The ratio of detected errors to injected faults represents the effectiveness of error recovery, while performance degradation represents the efficiency of error recovery.

In addition to obtaining a measure of the system fault tolerance, FTAPE is also useful for providing more detailed feedback to system designers. When system failures occur, the propagation of the guilty fault can be traced, and that information can be

*Professor, Electrical and Computer Engineering, Associate Fellow AIAA

used to improve the design of the fault containment mechanisms.

FTAPE is designed to be used on a functioning hardware implementation of a fault-tolerant computer. The tool has been implemented on a Tandem Integrity S2 fault-tolerant computer. Experiments using the tool show the effect of different workloads in influencing fault propagation. A measure of the overall system fault tolerance is also obtained. The implementation of FTAPE has been designed to be portable, although the fault injector is dependent to a degree upon the architecture of the measured machine. Plans exist to port the tool to other fault-tolerant machines and compare the fault tolerance of those machines.

2 Related Work

There are several different approaches to fault injection. Presently, FTAPE uses *software-implemented fault injection (SWIFI)*[1], which uses software to emulate the effects of underlying physical faults. Several fault injection tools use SWIFI, such as FIAT[5], FERRARI[3], and FINE[4]. FTAPE differs from these tools by adding a synthetic workload generator and the ability of the fault injector to inject faults based upon dynamic workload measurements.

Part of the dynamic workload measurement is performed using a hybrid monitor-based environment similar to that described by Young[7].

3 Description of Tool

FTAPE is a tool that integrates the injection of faults and the workload necessary to propagate those faults. The tool is composed of three main parts: FI (the fault injector), MEASURE, and WG (the workload generator). Figure 1 shows how these three parts interact. The FI is responsible for performing the fault injection. MEASURE provides a measurement of the current workload activity that is used by the FI to determine the time and location for fault injection. The WG is a synthetic workload generator which creates workloads that are designed to propagate the injected faults. A more detailed description of each part of the tool follows.

3.1 Fault Injector

The main task of the FI is to inject faults into the target system. The method of injection used by the current version of FTAPE is software-implemented fault injection, which uses software to emulate the effects of underlying physical faults. For instance, a bit in a memory location can be flipped to emulate the effect of an alpha particle on a memory bit. This method of fault injection is more controllable than hardware-based injection (e.g., it would be difficult to inject faults into memory using hardware), but software-based injection incurs a higher time overhead. See Section 4.5 for measurements of this overhead.

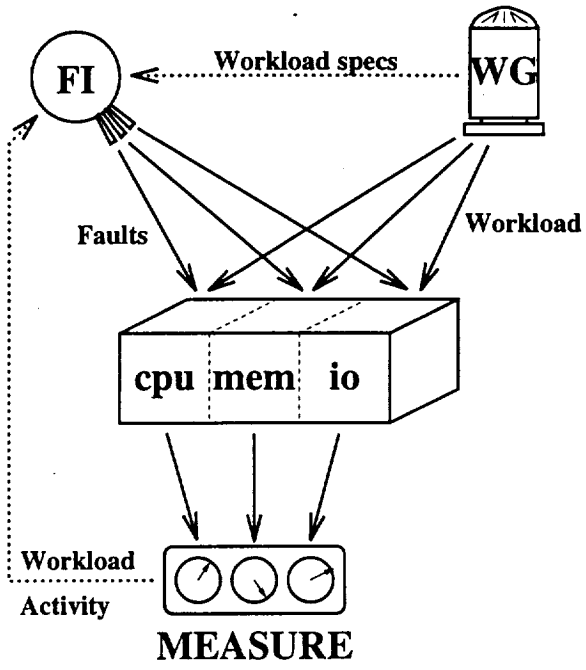

Figure 1: General Block Diagram of FTAPE

The main goal of fault injection is to exercise the error detection and recovery mechanisms in the target system. The best way to do this is to inject faults throughout the entire system. FTAPE partitions the system into three main areas: **cpu**, **mem**, and **io**. For each area a different method of fault injection is required. These areas are also the same areas that are targeted by the WG. Because the same areas are targeted by both the FI and the WG, there is a good chance for the injected faults to be propagated by the workload.

3.1.1 Fault Injection Method

The fault injection methods used by the FI are described below. Note that fault-tolerant systems have widely varying architectures and therefore require different fault injection techniques. The following are for the implementation of FTAPE on the Tandem Integrity S2:

inject_cpu Faults are injected into CPU registers, specifically, saved† general purpose and floating point registers, the program counter, the global pointer, and stack pointer. These registers were chosen because faults in these register have a higher chance of propagation compared to faults in other registers (e.g., temporary registers).

The method of injection involves the following steps:

1. Obtain a copy of the registers.
2. Corrupt the register to be injected.
3. Place the corrupted register value back into the CPU register. The transfer of the CPU values in and out is only performed when the targeted workload process is context switched out of the CPU.

The fault model is a single bit-flip.

inject_mem Faults are injected into local memory. Since only portions of the memory are heavily used by the workload, faults are targeted at those portions. Faults are injected by directly modifying the contents of selected memory locations. The fault model is a single bit-flip.

inject_io Faults are injected into a mirrored disk system. The method of injection involves using a test portion of the disk driver code that sets error flags for the next driver request. Thus, the next request will be detected by the error handler in the driver code, and one half of the disk mirror may be disabled. The fault model includes valid disk error codes.

3.1.2 Fault Selection Method

The time and location for each fault injection is determined using one of the following methods. Some of the methods involve the measurement of workload stress, which is described in the next section:

location-based stress-based injection (LSBI) Faults are injected into the area (CPU, memory, or IO) with the greatest normalized stress.

time-based stress-based injection (TSBI) Faults are injected during the time the composite stress is greater than a specific threshold.

randomly The fault time is selected randomly based on a specified distribution (e.g., an exponential interarrival distribution with a specified mean of 20 seconds), and the fault location is randomly chosen based on a uniform distribution.

If an error is detected, then all injections are suspended until the error is corrected, because an error detection on the Tandem S2 disables the component in which the error was detected (e.g., a detected error in the CPU forces the entire CPU off-line).

The fault models used for **cpu** and **mem** are single bit-flips. For **io**, valid error codes are randomly chosen.

3.2 Workload Generator

The main purpose of the workload generator is to provide an easily controllable workload that can propagate the faults injected by the FI. The workload is synthetic to allow easy specification of the workload, based on a few parameters. The same areas that are used used the FI (**cpu**, **mem**, and **io**) are targeted for workload activity. Each workload is comprised of one or more processes. Each process is composed of a sequence of the following three functions, each of which exercises one of the three main system areas intensively:

use_cpu This function is CPU-intensive. It consists of repeated additions, subtractions, multiplications, and divisions for integer and floating point variables. These operations are performed in a loop containing conditional branches. Memory accesses are limited by using CPU registers as much as possible.

use_mem This function is memory-intensive. A large memory array is created, and locations in this array are repeatedly read from and written to in a sequential manner. The array is larger than the size of the data cache in order to ensure that accesses are being made to the physical memory.

use_io This function is I/O-intensive. A dummy file system is created on a mirrored disk system. Opens, reads, writes, and closes are repeatedly performed.

The parameters for each function are specified in a parameter file. In practice, each function is usually specified to last the same amount of time (e.g., one second). Then the composition of each workload process can be specified to contain a specific proportion of each function. For instance, a workload that is CPU-intensive with a small amount of memory and I/O activity can be specified to contain

†*Saved* registers are those registers who values must be preserved across procedure calls.

90% of the **cpu** function and 5% of the **mem** and **io** functions. Such a workload would be said to have a *composition* of 90/5/5. When the workload process is executed, each function will be randomly chosen according to corresponding probabilities.

Each function also reads and writes data from a special global *interdependence array* which forces data flow among functions. This is necessary to encourage fault propagation among functions. Otherwise, a data fault in one function is usually overwritten if the fault influences only variables local to that function and the system doesn't detect the error before the end of the function.

The *intensity* is the amount of activity in each function relative to the maximum possible activity. The intensity of each function can be controlled. This is useful for studying the impact of the workload activity level on fault propagation. For most of the workloads used in the experiments in Section 4, the intensity is varied from 100% to 20% over a period of about nine minutes[‡]. Varying the intensity emphasizes the effect of high and low workload activity on the amount of fault propagation.

Finally, the workload sends to the FI information needed to determine the location of certain faults, such which processes are currently executing and what portions of memory are being used.

3.3 MEASURE

MEASURE is a tool that monitors the actual workload activity. Each workload is specified by its associated parameter files to contain a certain relative amount of **cpu**, **mem**, and **io** activity. Although each workload function is designed to be very intensive for one system area, each function must necessarily cause activity in other system areas. For instance, the **io** function must also use the CPU and perform memory reads and writes as well as accessing the disk. Thus, the MEASURE tool is necessary to measure the actual activity caused by the workload.

MEASURE returns the level of workload *stress* for each system area as well as for the system as a whole. The stress is the amount of workload activity — especially that which can aid fault propagation. As with the FI, the methods needed to obtain the stress measures for each system area are system dependent to a large extent. For each system area, the following methods are used to obtain the workload stress:

measure_cpu The stress measure is based upon the CPU utilization. On the S2, the **sar** utility returns the CPU utilization.

measure_mem The stress measure is based upon the number of reads and writes per second to the memory space used by the workload. Since any software method of obtaining this information would incur an unacceptable amount of overhead, a hardware method is used. A Tektronix DAS 9200 logic analyzer is used to count the number of memory accesses. This count is automatically sent to the MEASURE program every 10 seconds.

measure_io The stress measure is based on the number of disk blocks accessed per second. On the S2, the **sar** utility returns the number of disk blocks accessed per second.

A detailed description of the setup needed to measure **mem** stress can be found in Young[6].

Each stress measure is normalized in order to compare the different measures. The normalization is performed by running a set of various workloads[§] and obtaining a distribution of the raw stress measures (i.e., CPU utilization, memory accesses/second, and disk blocks/second). Each raw stress measure was normalized to a value between 0 and 1, inclusively, based on the following formula, where X_{min} is the 5th percentile value and X_{max} is the 95th percentile value in the raw stress distribution:

$$X_{normal} = \min\left\{\max\left[\left(\frac{X - X_{min}}{X_{max} - X_{min}}\right), 0\right], 1\right\}.$$

One disadvantage of the current methods is the relatively long amount of time between measurements (about 10 seconds). This is mainly due to the amount of time required by the logic analyzer to count memory accesses. However, most of this time is used to set up the logic analyzer; the actual count only takes about one second. A newer logic analyzer will be used in the future to significantly decrease this setup time.

4 Experiments

The main goals of the following experiments are

- to see how FTAPE can be used to investigate how a specific machine (the Tandem Integrity S2) performs under faults and

[‡]This time period needs to be long enough for the MEASURE tool and FI to react to the corresponding workload activity.

[§]These workloads had compositions of 33/33/33, 20/20/60, 20/60/20, and 60/20/20.

- to illustrate the effectiveness of stress-based injection.

The target machine for these experiments is the Tandem Integrity S2 fault-tolerant computer. A brief description of the S2 is given in Section 4.1. The general experimental procedure is described in Section 4.2. The first set of experiments, described in Section 4.3, involves injecting coordinated faults (i.e., faults that are injected into areas of greatest workload stress) and uncoordinated faults (i.e., faults that are injected into areas of least workload stress). These experiments expose the sensitivity of certain workloads to specific faults. The next set of experiments, presented in Section 4.4, illustrates the effectiveness of stress-based injections in increasing fault propagation. Finally, the overhead of the FI and MEASURE tools is measured and given in Section 4.5.

4.1 Description of S2

The Integrity S2[2] is a fault-tolerant computer designed by Tandem Computers, Inc. The core of the S2 is its triple-modular-redundant processors. Each processor includes a CPU, a cache, and an 8MB local memory. Although these three processors perform the same work, they operate independently of each other until they need to access the doubly-replicated global memory. At this point, the duplexed Triple Modular Redundant Controllers (TMRCs) vote on the address and data. If an error is found, the faulty processor is shut down. After that processor passes a power-on self-test (POST), it is reintegrated into the system by copying the states of the two good processors. Voting also occurs on all I/O and interrupts. In addition, the local memory is scrubbed periodically. This architecture ensures that a fault that occurs on one processor will not propagate to other system components without being caught by the TMRC voting process.

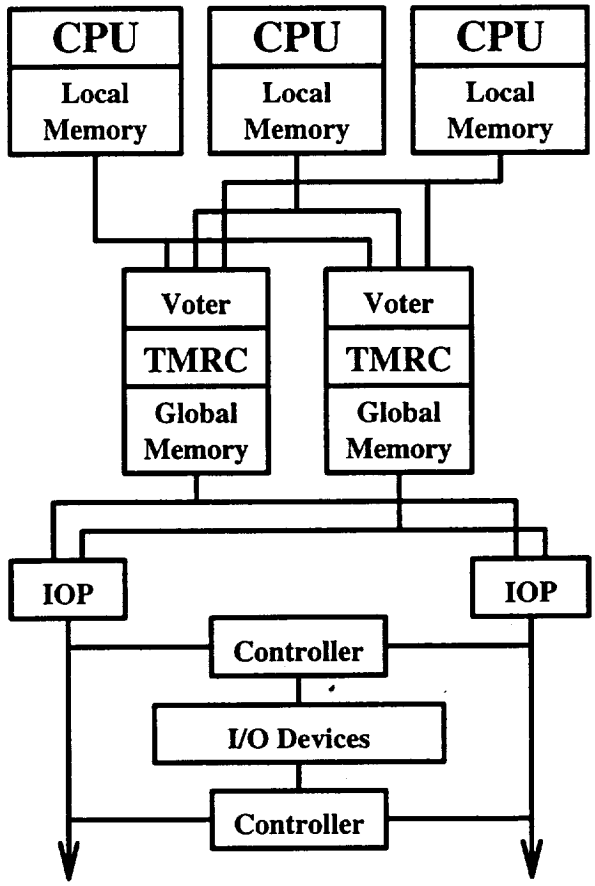

Figure 2: Overview of Tandem Integrity S2 Architecture

4.2 General Experimental Procedure

Each experiment thus is composed of two runs, one with faults and one without. The reason for this duplication is that is allows the calculation of the *performance degradation*, which is the amount of extra time required by the workload due to the detection and correction of faults by the system. If T_f is the workload execution time under fault injection and T_{nf} is the time with no faults, then the performance

degradation is

$$\frac{\text{Performance}}{\text{Degradation}} = \frac{T_f}{T_{nf}} - 1.$$

Performance degradation can be used as a measure of system fault tolerance because it is related to dependability degradation. When a system is in an error state, it is vulnerable to system failure if another fault is activated (assuming single fault tolerance), and thus the dependability of the system is degraded. Dependability degradation is difficult to measure directly. The performance degradation due to faults is easily measured.

Each experiment consists of the following steps:

1. Start the MEASURE tool.
2. Run the workload while injecting faults. Measure the total workload time required.
3. Run the workload a second time, this time without injecting any faults. Again measure the total workload time required.

For the second non-injection run, the FI is still executed, but with null injection masks. In other words, the FI goes through the motions of injecting faults, but instead of flipping a bit (XORing with a 1) and setting a disk error (setting error to nonzero value), the FI doesn't flip a bit (XORs with a 0) and sets a null disk error (sets error to zero value). By so doing, the second run will also invoke the same FI overhead as the first run. This is important when comparing workload execution times.

To illustrate the relationship between dependability degradation and performance degradation, consider, for example, the mirrored disk system. When an error is detected on one of the mirrored disks, the mirror half is taken off-line. This results in lower dependability because an error in the remaining mirror half will take down the entire disk system (which would result in a total system failure if it contains vital files). When the bad mirror half is off-line, all disk reads must be serviced by the single on-line mirror half. This results in lower degraded performance compared to a disk mirror with two on-line halves, which can send half of the disk reads to each mirror half.

Another result that is interesting is the ratio of error detections to fault injections. Along with the performance degradation the errors/fault ratio show the amount of fault propagation for each experiment.

4.3 Sensitivity of Workloads to Faults

The experiments in this section show that faults which are injected into the areas of greatest workload stress produce the most fault propagation. The experiments are grouped into two categories: (1) inject faults into areas with little workload stress and (2) inject faults into areas with the greatest workload stress. For example, the experiments in the first two rows of Table 1 involve **cpu** injections. The first experiment uses a **cpu**-intensive workload, while the second other uses a mostly **mem** and **io** workload. The injection time is chosen randomly based on an exponential rate with a specified mean of 20 seconds.

The results of the experiments are given in Table 1. Each row represents the average of multiple runs. From Table 1, a few observations can be made. First, the amount of fault propagation (as seen in the errors/fault and performance degradation columns) is higher if faults are exercised by heavy workload activity in that area (as shown in the composition column). This is especially evident for **io**, where the rate of error detection doubles from 0.295 to 0.650, and the performance degradation increases from 0.0703 to 1.1296, when an **io**-intensive workload is used. This effect occurs because the injected **io** faults are being activated more by the increased **io** workload activity. The reason the **io** fault propagation increase is more pronounced than for **cpu** or **mem** is the different type of effect a downed mirror half has compared to a downed CPU or memory. The downed mirror half causes all disk reads to go to the single mirror half, thus almost halving the disk access rate (if disk writes are ignored) until the mirror half is repaired and brought on-line (which takes about 100 seconds). The down CPU or memory only impacts the performance when the CPU or memory must be halted for repair (i.e., copying the data from the good CPU or memory), which requires only one or two seconds. These results verify that fault propagation is affected by the workload.

4.4 Stress-based Injection Results

The experiments in this section use time-based (TSBI) and location-based (LSBI) stress-based injection to determine the time and location for fault injection. The results of these experiments are given in Table 2. Each line in the table represents the average of multiple runs. Four different workloads are injected with faults in the following manner:

1. Faults are injected only if the composite workload stress is higher then the high stress threshold. (Noted as "High" TSBI in the table.)
2. Faults are injected only if the composite workload stress is lower than the low stress threshold. (Noted as "Low" TSBI in the table.)

Table 1: Sensitivity of Workloads to Faults

Run	Injection Location	Composition cpu	mem	io	Faults Injected	Errors Detected	Errors / Fault	Time with Faults (sec)	Time without Faults (sec)	Performance Degradation
a	cpu	4	48	48	61	9	0.148	1588	1544	0.0285
b	cpu	90	5	5	101	26	0.257	2334	2236	0.0438
c	mem	48	4	48	87.3	2.3	0.027	1948	1928	0.0104
d	mem	5	90	5	70.7	2.7	0.038	1558	1537	0.0137
e	io	48	48	4	48.3	12	0.248	2026	1910	0.0607
f	io	5	5	90	36.7	25.7	0.700	3347	1583	1.1143

Table 2: Stress-based Injection Results

Run	TSBI	Composition cpu	mem	io	Faults Injected	Errors Detected	Errors / Fault	Time with Faults (sec)	Time without Faults (sec)	Performance Degradation
g	High	90	5	5	24	4	0.177	2362	2299	0.0273
	Low	90	5	5	170	12	0.073	2375	2301	0.0322
h	High	20	20	60	73	14	0.197	2335	1713	0.3632
	Low	20	20	60	75	2	0.027	1755	1712	0.0251
i	High	5	5	90	38	30	0.788	3573	1589	1.248
	Low	5	5	90	55	2	0.032	1593	1586	0.0046
j	High	33	33	33	68	8	0.118	1959	1861	0.0524
	Low	33	33	33	96	4	0.045	1875	1856	0.0102

It can be seen that the fault propagation (as seen in the errors/fault and performance degradation columns) is lower in all cases when faults are injected during times of low workload activity. As was the case with the experiments in the previous section, the io-intensive workload (with 5/5/90 composition) has the most significant difference in fault propagation.

4.5 Overhead

There is an overhead associated with the FI and MEASURE tools. Each requires CPU time, which increases the workload execution time. Also, additional effects such as cache flushing, paging, and I/O activity can further increase the workload execution time. The impact of this overhead on the workload execution time can be measured by comparing the time the workload requires with and without the FI and MEASURE tools executing simultaneously. The results are given in Table 3. Although the overhead caused by the FI and MEASURE programs is significant (5.4-16.8%), this overhead is incurred for all experiments. For the two runs in each type of experiment, the run with faults and the run without faults requires the FI and MEASURE programs to perform the same operations. Thus, although the overhead affects the absolute results, the relative results are much less affected by the overhead. Further study is needed to determine if the impact can be completely ignored.

The overhead also seems to increase with more CPU activity and less I/O activity. This might occur because less idle CPU cycles are available for the FI and Measure programs as the workload blocks less for I/O and uses the CPU more.

5 Conclusions

FTAPE is a tool that can compare the fault tolerance of fault-tolerant computers. Faults are injected injected at times and locations of greatest workload activity in order to encourage fault propagation. Experiments with FTAPE show an increase in fault propagation (as measured by errors/fault and performance degradation) when faults are injected (1) into components (e.g., CPU) that are exercised heavily by the workload and (2) at times of greatest overall workload stress.

In the future, the tool will be ported to other fault-tolerant platforms and used to compare these machines. More representative workloads and fault models will be incorporated into the tool.

6 Acknowledgements

Thanks are due Tandem Computer, Inc. for their help in this work. This research was supported in

Table 3: Overhead Measurements

Run	Composition cpu	mem	io	Time with FI/MEASURE (sec)	Time without FI/MEASURE (sec)	% Overhead
g	90	5	5	2289	1959	16.8%
h	20	20	60	1709	1564	9.3%
i	5	5	90	1594	1512	5.4%
j	33	33	33	1847	1591	16.1%

part by the Advanced Research Projects Agency (ARPA) under contract DABT63-94-C-0045 and by NASA grant NAG 1-613, in cooperation with the Illinois Computer Laboratory for Aerospace Systems and Software (ICLASS). The content of this paper does not necessarily reflect the position or policy of the government and no endoresement should be inferred.

References

[1] J. H. Barton et al., "Fault injection experiments using fiat," *IEEE Transactions on Computers*, vol. 39, pp. 575–582, April 1990.

[2] D. Jewett, "Integrity s2: A fault-tolerant UNIX platform," in *21st International Symposium on Fault-Tolerant Computing*, (Montreal, Canada), pp. 512–519, June 1991.

[3] G. Kanawati, N. Kanawati, and J. Abraham, "Ferrari: A fault and error automatic real-time injector," in *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing*, (Boston, Massachusetts), 1992.

[4] W.-L. Kao, R. Iyer, and D. Tang, "FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior under Faults," in *IEEE Transactions on Software Engineering*, vol. 19, pp. 1105-1118, November 1993.

[5] Z. Segall et al., "Fiat-fault injection-based automated testing environment," in *18th International Symposium on Fault-Tolerant Computing*, pp. 102–107, 1988.

[6] L. Young and R. Iyer, "Error latency measurements in symbolic architectures," in *AIAA Computing in Aerospace 8*, (Baltimore, Maryland), pp. 786–794, October 1992.

[7] L. Young et al., "Hybrid monitor assisted fault injection environment," in *Proceedings of the Third IFIP Working Conference on Dependable Computing for Critical Applications*, (Mondello, Sicily, Italy), pp. 163–174, September 1992.