Measuring Fault Tolerance with the FTAPE Fault Injection Tool

Timothy K. Tsai and Ravishankar K. Iyer
University of Illinois
Center for Reliable and High Performance Computing
Coordinated Science Laboratory
Urbana, Illinois
Email: {ttsai|iyer}@crhc.uiuc.edu
Telephone: (217) 244-1768 Fax: (217) 244-5686

Abstract

This paper describes FTAPE (Fault Tolerance And Performance Evaluator), a tool that can be used to compare fault-tolerant computers. The major parts of the tool include a system-wide fault injector, a workload generator, and a workload activity measurement tool. The workload creates high stress conditions on the machine. Using *stress-based injection*, the fault injector is able to utilize knowledge of the workload activity to ensure a high level of fault propagation. The errors/fault ratio, performance degradation, and number of system crashes are presented as measures of fault tolerance.

Keywords: fault injection, workload generator, fault tolerance measurement, stress-based injection

1 Introduction

A method for measuring the fault tolerance of any computer is desirable. Such a method could be used to compare fault-tolerant computers. For system designers, different fault-tolerant designs could be evaluated, and feedback can be provided in the design process. A fault tolerance measure would be useful to purchasers of new fault-tolerant systems in encapsulating the effectiveness and efficiency of the fault tolerance.

This paper describes FTAPE (Fault Tolerance and Performance Evaluator). The tool combines a fault injector and a workload generator with a workload activity measurement tool in order to inject faults under high stress conditions based on workload activity. It is well known that high stress and complex workloads cause greater propagation of faults and detection of errors[5]. By using knowledge of the workload activity, the fault injector can maximize the chance that faults are activated. The tool also characterizes the fault tolerance of a computer by producing a single measure. This use of FTAPE is analogous to the use of synthetic benchmarks in evaluating CPU performance. Despite the flaws of such measures, these benchmarks are nevertheless useful in comparing different systems. In the area of fault tolerance, no such benchmarks are currently available. Although it may be argued that measures produced by FTAPE are not be the most definitive, they are certainly useful as fault tolerance measures and should motivate the discussion in the area of fault tolerance metrics.

The use of fault injection to measure fault tolerance is needed because the error detection and recovery mechanisms that comprise the fault tolerance of a computer can only be tested when activated by faults and their corresponding manifestations. The fault injector in FTAPE can inject faults throughout the target system, including in CPU, memory, and disk components. This ability to inject faults into different parts of the system is needed due to the distributed nature of the fault tolerance mechanisms.

The workload generator is synthetic and designed to produce workloads that will exercise the CPU, memory, and disk. The amount and intensity of the workload in each area of the system (CPU, memory, and disk) can be controlled and specified as distributions over time.

We define *stress-based injection* as 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. In this paper, fault propagation refers to the activation of faults and the resulting deviation of the system state from the fault-free case. The effect of the fault is propagated due to the activity of the workload. The workload activity measurement tool provides two values: (1) the level of workload activity in each system component (CPU, memory, and disk), which determines the location of injection, and (2) the level of workload activity in the entire system, which determines the time of injection. Results from several experiments are provided to demonstrate the effectiveness of stress-based injection in increasing fault propagation.

Since the tool characterizes 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 detection, while performance degradation represents the efficiency of error recovery. The number of system crashes measures the effectiveness of error recovery.

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 some degree upon the architecture of the measured machine. Plans exist to port the tool to other fault-tolerant machines and to compare the fault tolerance of those machines.

2 Related Work

There are several different approaches to fault injection. A detailed discussion can be found in [6]. Fault injection tools can be classified as simulation-based or prototype-based. Simulation involves the injection of faults into electrical, logical, or functional simulation models. Simulation has the ability to model complex systems with greater accuracy than analytic modeling. However, ensuring that the simulated models are realistic and containing simulation time explosion are challenges. Examples of fault simulators include FOCUS [2] and MEFISTO [7].

For prototype-based injection, faults are injected into existing physical systems. There are several methods for injecting faults into these computers. Hardware-implemented fault injection uses additional hardware instrumentation to introduce faults. FTMP [3] and MESSALINE [1] use active probes and special sockets to alter currents and voltages on chip pins. Radiation [4] can also be used to inject chip-level faults. Although these methods inject actual low-level, physical faults, they require special hardware and accessibility to the target system hardware and have the possibility of damaging the target system.

Another method of injection involving prototypes is *software-based fault injection* (SWIFI). This method uses software to emulate the effect of lower-level hardware faults by altering the contents of memory or registers. No additional hardware is required. Some fault injection tools based on SWIFI are given in Table 1. FIAT injects faults into the user process image in memory. FERRARI uses software traps to inject faults. HYBRID uses hardware and software monitoring of SWIFI faults. DEFINE is a distributed version of FINE [11], which is able to emulate software faults as well as hardware faults. SFI is used to validate dependability mechanisms and has been used on a distributed real-time system. FTAPE differs from these tools by adding a synthetic workload generator and a workload activity measurement tool that enables the fault injector to inject faults based upon dynamic workload measurements. Performance degradation is measured along with other quantities as measures of the overall system fault tolerance.

Table 1: Software-implemented Fault Injection Tools

Name	System Under Test	Fault Types
FIAT [13]	IBM RT	CPU, memory, communications
FERRARI[9]	Sun	CPU, memory, bus, communications
HYBRID [15]	Tandem S2	CPU, memory
DEFINE [10]	Sun	CPU, memory, bus, SW, distributed
SFI[12]	HARTS	CPU, memory, communications
FTAPE	Tandem S2	CPU, memory, disk

3 Description of Tool

FTAPE is a tool that integrates the injection of faults and the generation of 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. A more detailed description of each part of the tool follows.

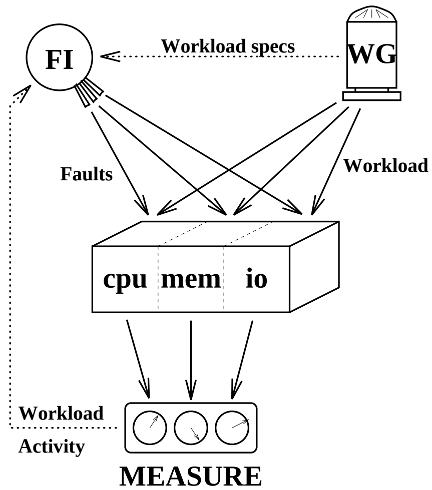

Figure 1: Block Diagram of FTAPE

3.1 Fault Injector

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 selected because it is more controllable than hardware-based injection and does not require additional injection hardware.

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 targeted by the WG in order to increase the 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 is for the implementation of FTAPE on the Tandem Integrity S2:

Injection method 1: inject_cpu The CPU fault models include single/multiple bit-flip and zero/set faults in CPU registers.

Faults are injected into CPU registers, specifically, saved[1] general purpose and floating point registers, the program counter, the global pointer, and

[1] *Saved* registers are those registers who values must be preserved across procedure calls.

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).

Injection method 2: inject_mem The memory fault models include single/multiple bit-flip and zero/set faults in local and global memory.
Faults are targeted at heavily used parts of memory. Faults are injected by directly modifying the contents of selected memory locations.

Injection method 3: inject_io The I/O fault models include valid SCSI and disk errors.
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 activates the error handler in the driver code, and one half of the disk mirror may be disabled.

3.1.2 Fault Selection Method

The time and/or 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:

Selection method 1: location stress-based injection (LSBI) Faults are injected into the area (CPU, memory, or I/O) with the greatest normalized stress.

Selection method 2: time stress-based injection (TSBI) Faults are injected during the time the composite system stress (described later) is greater than a specific threshold.

Selection method 3: 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.

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

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 control of the workload, based on a few parameters. The same areas that are used by the FI (*cpu*, *mem*, and *io*) are targeted for workload activity. The workload is composed of a mixture of the following three functions, each of which exercises one of the three main system areas intensively:

Function 1: 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.

Function 2: 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 (because the cache is direct mapped with a block size of one word).

Function 3: 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.

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 90% of the *cpu* function and 5% each 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 the 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 amount of data flow among functions via the interdependence array can be controlled by resetting parts of the array to default values. This has the effect of providing some measure of control of data flow, and therefore fault propagation, through global variables.

The *intensity* is the amount of activity in each function relative to the maximum possible activity. The intensity of each function is decreased by substituting calls to the `usleep()` function instead of code that would otherwise be executed. Thus, an intensity of 100% would contain no additional `usleep()` calls. The ability to control the intensity 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[2]. Varying the intensity emphasizes the effect of high and low workload activity on the amount of fault propagation.

Finally, the workload provides the FI with information needed to determine the location of certain faults, such as 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. Although each workload function is designed to be very intensive for one system area, each

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

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 some extent. For each system area, the following methods are used to obtain the workload stress:

Method 1: measure_cpu The stress measure is based upon the CPU utilization. On the S2, the `sar` utility returns the CPU utilization.

Method 2: 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. A detailed description of the setup needed to measure *mem* stress can be found in Young[14].

Method 3: 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.

Each stress measure is normalized in order to compare the different measures. The normalization is performed by running a set of various workloads[3] 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

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

- to illustrate the use of FTAPE in investigating a specific machine as it performs in the presence of faults and
- 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 matched faults (i.e., faults that are injected into areas of greatest workload stress) and unmatched 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.

4.1 Description of S2

The Integrity S2[8] 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. If it 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.

4.2 General Experimental Procedure

Each experiment is composed of two runs, one with faults and one without. The reason for this duplication is that it allows the calculation of the *performance degradation*, which is the ratio of two times: (1) the extra time required by the workload due to the detection and correction of faults by the system and (2) the workload execution time without faults. This ratio is adjusted by the number of faults injected. If T_f is the workload execution time under fault injection, T_{nf} is the time with no faults, and n is the number of faults injected, then the performance degradation is

$$\begin{matrix}\text{Performance}\\ \text{Degradation}\end{matrix} = \frac{1}{n}\left(\frac{T_f}{T_{nf}} - 1\right). \quad (1)$$

Performance degradation is a measure of the amount of extra time a system requires to recover from detected errors. Performance degradation can be used as a measure of a system's fault tolerance, where a lower level of degradation means that the recovery mechanisms are more efficient. In order to obtain this measure, runs of the experiment which cause system crashes are ignored, since

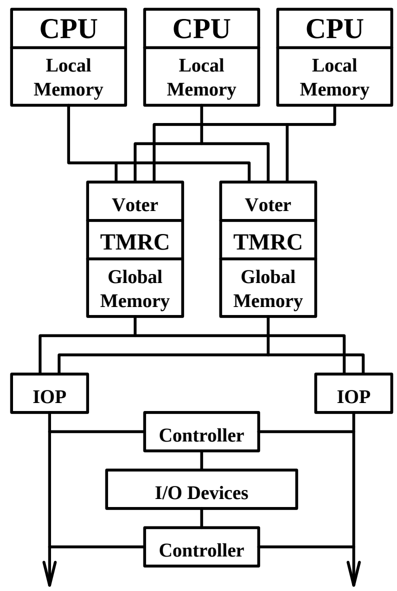

Figure 2: Overview of Tandem Integrity S2 Architecture

the degradation would be infinite in that case. Instead, the number of system crashes is counted and can be used as another measure of fault tolerance (i.e., how well the system is able to recover from faults).

One run of 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 (T_f in Equation 1).
3. Run the workload a second time, this time without injecting any faults. Again measure the total workload time required ($T_n f$ in Equation 1).

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 the error value to a nonzero value), the FI doesn't flip a bit (XORs with a 0) and sets a null disk error (sets the error value to a zero value). By so doing, the second run will also invoke the same FI overhead as the first run, which is needed when comparing workload execution times.

In addition to performance degradation, the ratio of error detections to fault injections is measured for each run. This ratio represents the effectiveness of error detection. Since it is usually desirable to detect as many faults as possible, the errors/fault ratio should be maximized. Although analogous to error detection coverage, it is different because multiple injected faults may be concurrently present in a system component. When a single error in that component is detected, reintegration of that component results in correction and removal of all faults in that component. Thus, the errors/fault ratio is always less than or equal to the error detection coverage.

Performance degradation and the errors/fault ratio can also be used to measure the level of fault propagation on a single machine. Since the detection and recovery mechanisms on a machine remain the same from one run to another, variations in these two measures are caused by the detection of errors caused by injected faults. The more the faults propagate, the more error detections and the errors/fault ratio are likely to increase. A larger number of error detections causes more recovery activity and hence increases the performance degradation.

4.3 Sensitivity of Workloads to Faults

Faults are activated and propagated by workloads. The experiments in this section show that more fault propagation occurs when the locations of faults and high workload activity are the same. The experiments consist of injecting faults into a single system component. Two types of workloads are executed along with those fault injections: (1) a workload with little activity in that component and (2) a workload with its activity mostly concentrated in that component. Thus, for experiment (c) in Table 2, faults are injected only into the disk. The first row represents a non-disk intensive workload, while the second row represents a disk-intensive workload. The injection time was chosen based on an exponential arrival distribution with a mean of 20 seconds.

The results are given in Table 2. Each row represents seven runs. From the table, it can be seen that the errors/fault ratio and the performance degradation are higher for the second row of each experiment. This means that the fault propagation is indeed higher when the injection location matches the location of high workload activity. For instance, the errors/fault ratio for *io* injections increases from 0.248 to 0.700 when the workload activity becomes disk-intensive. Similarly the performance degradation increases from 0.001257 to 0.030363.

The increase in the errors/fault ratio occurs because the injected faults are accessed by the workload more frequently when the workload activity is concentrated in the injection area. Furthermore, the high workload activity causes the accessed fault to produce additional errors. For instance, a CPU fault may be a corrupted register. That register may be a pointer to a memory location. Each time that corrupted register is referenced by the workload, an additional memory error is created (i.e., fault propagation). This fault propagation effect is increased when the workload causes the register to be used more often.

4.4 Stress-based Injection Results

Stress-based injection is a method of selecting the time and location for injected faults with the goal of producing the greatest fault propagation possible. Injected faults must be activated and propagated in order to adequately exercise the error detection and correction mechanisms on a fault-tolerant system. Thus, by using stress-based injection, the likelihood that the fault tolerance of a system is tested can be increased.

To show that stress-based injection increases fault propagation, experiments were performed using using five different stress-based injection strategies:

Table 2: Sensitivity of Workloads to Faults

Exp	Injection Location	Composition cpu	mem	io
a	*cpu*	4	48	48
	cpu	90	5	5
b	*mem*	48	4	48
	mem	5	90	5
c	*io*	48	48	4
	io	5	5	90

Exp	Inj. Loc.	Errors Detected	Faults Injected	$\frac{\text{Errors}}{\text{Fault}}$	Execution Time with Faults (sec)	Execution Time without Faults (sec)	Performance Degradation
a	*cpu*	9	61	0.148	1588	1544	0.000467
	cpu	26	101	0.257	2334	2236	0.000434
b	*mem*	2	87	0.027	1948	1928	0.000119
	mem	3	71	0.038	1558	1537	0.000193
c	*io*	12	48	0.248	2026	1910	0.001257
	io	26	37	0.700	3347	1583	0.030363

Strategy	Description
lt	Use both location-based stress injection (LSBI) and time-based stress injection (TSBI).
l	Use LSBI.
t	Use TSBI.
random	Randomly select injection times from an exponential distribution and injection locations from a uniform distribution.
ltLOW	Use both LSBI and TSBI. However, select injection times when the composite stress is below a specific threshold, and select the injection location (CPU, memory, I/O) with the lowest measured stress.

The errors/fault ratio and performance degradation for the five injection strategies used with the same workload are given in Figure 3. The figure shows averages based on 19 runs for each injection strategy. The workload used is a disk-intensive workload. From the figure, it can be seen that the highest level of fault propagation (as measured by the errors/fault ratio and performance degradation) is obtained when using both the location-based and time-based injection strategies (labeled in the graph as "lt"). If only the location-based strategy (labeled as "l") is used, then the propagation is lower. However, the location-based strategy still produces more propagation than using the time-based or random strategies (labeled as "t" and "random", respectively). Thus, for this disk-intensive workload, injecting faults into the disk produces more fault propagation than choosing the injection location randomly. However, if additionally the faults are injected only when the dynamic workload activity is high, then even more propagation occurs.

The measured performance degradation in Figure 3 is small, partly because it is divided by the number of faults injected. However, the measure is significant because it is intended to be used as a relative measure. Thus, the importance of the measure is that the combined location-base and time-based injection strategy produces more performance degradation than the other strategies.

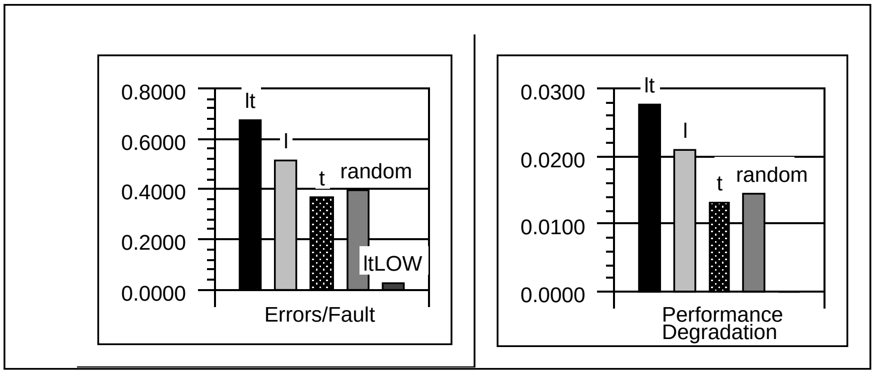

Figure 3: Errors/Fault and Performance Degradation

This same effect can be seen for other workloads. Figure 4 shows the errors/fault ratio for several workloads. For each workload, the errors/fault ratio is higher when the location-based strategy is combined with the time-based strategy. Again, the combined strategy is labeled as "lt" in the graph, while the location-based strategy alone is labeled as "l".

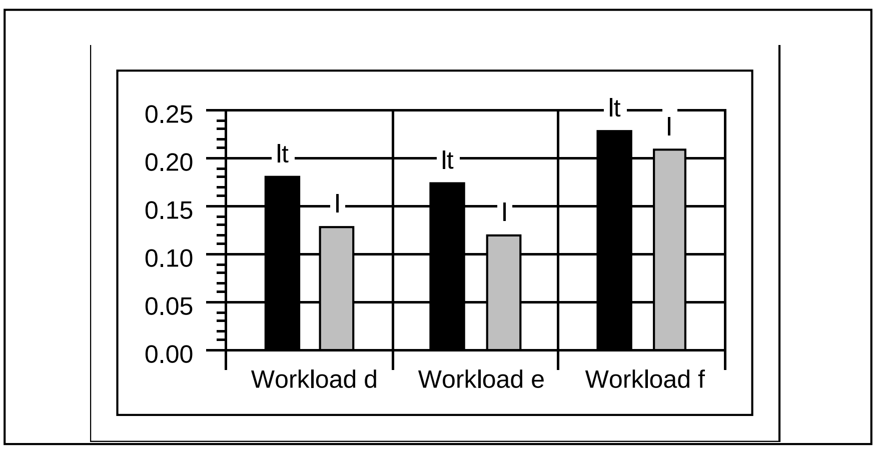

Figure 4: Errors/Fault for Several Workloads

As shown in Table 2, disk faults have a much higher errors/fault ratio and performance degradation compared to CPU and memory faults. To ensure that the results of the experiments are not biased by this, the results were also calculated for the same experiments in Figure 4 ignoring disk faults. The results are given in Table 3. Again, the errors/fault ratio is highest when the location-

Table 3: Stress-based Injection Results For CPU and Memory Fault

Exp	Injection Method	Composition			# Runs	Errors Detected	Faults Injected	$\frac{\text{Errors}}{\text{Fault}}$
		cpu	*mem*	*io*				
d	lt	90	5	5	19	4	22	*0.1749*±0.0362
	l	90	5	5	18	13	104	0.1206±0.0147
	t	90	5	5	19	2	17	0.1184±0.0353
	random	90	5	5	19	3	23	0.1170±0.0302
	ltLOW	90	5	5	6	12	169	0.0740±0.0161
e	lt	33	33	33	12	11	66	*0.1679*±0.0261
	l	33	33	33	17	7	71	0.1007±0.0169
	t	33	33	33	18	3	29	0.1075±0.0264
	random	33	33	33	16	3	28	0.1053±0.0282
	ltLOW	33	33	33	5	4	94	0.0403±0.0177
f	lt	20	20	60	19	7	60	*0.1178*±0.0187
	l	20	20	60	10	4	52	0.0874±0.0220
	t	20	20	60	9	3	33	0.1003±0.0298
	random	20	20	60	19	4	32	0.1151±0.0254
	ltLOW	20	20	60	6	2	76	0.0263±0.0147

based and time-based injection strategies (labeled as "lt") are combined. The errors/fault ratios for the lt strategies are highlighted in the table. For the errors/faults ratio, 95% confidence intervals are given.

System Crash Data

The results above do not include experiments which resulted in system crashes. The number of system crashes is given in Table 4 classified by the injection strategy and workload used. Each row represents a different workload, and each column represents the injection strategy used. For example, the "lt" column of row "e" shows that 3 system crashes occurred while the combined location-based and time-based injection strategy was used with workload (e). The table includes data for 272 runs, during which a total of 5 system crashes occurred. All crashes occurred when the location-based and time-based injection strategies were used. This result is consistent with the results of the other experiments in this paper, which show that the combined location-based and time-based strategy seems to produce the most fault propagation.

Repeatability

5 Conclusions

FTAPE is a tool that can be used to compare the fault tolerance of computer systems. Stress-based injection is used to to inject faults at the times and locations of greatest workload activity. This encourages fault propagation, which is necessary to ensure that the fault-tolerant mechanisms are adequately exercised. Experiments on the Tandem Integrity S2 show that fault propagation (as measured by error/fault, performance degradation, and system crashes) is highest when faults are injected (1) into components (e.g., CPU) that are exercised

Table 4: Number of Observed System Crashes

Experiment	Injection strategies				
	lt	l	t	random	ltLOW
d	0	0	0	0	0
e	3	0	0	0	0
f	1	0	0	0	0
g	1	0	0	0	0
Total	5	0	0	0	0

heavily by the workload and (2) at times of greatest overall workload stress.

6 Acknowledgments

Thanks are due Tandem Computer, Inc., and especially Luke Young, for their help in this work. This research was supported in part by the Advanced Research Projects Agency (ARPA) 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). The content of this paper does not necessarily reflect the position or policy of these agencies, and no endorsement should be inferred.

References

[1] J. Arlat et al. Fault injection for dependability validation-a methodology and some applications. *IEEE Transactions on Software Engineering*, 16(2):166–182, February 1990.

[2] G. S. Choi, R. K. Iyer, and V. Carreno. Focus: An experimental environment for fault sensitivity analysis. *IEEE Transactions on Computers*, 41(12):1515–1526, December 1992.

[3] G. B. Finelli. Characterization of fault recovery through fault injection on FTMP. *IEEE Transactions on Reliability*, 36(2):164–170, June 1987.

[4] U. Gunneflo, J. Karlsson, and J. Rorrin. Evaluation of error detection schemes using fault injection by heavy-ion radiation. In *Proceedings 19th International Symposium on Fault-Tolerant Computing*, pages 340–347, Chicago, Illinois, June 1989.

[5] R. Iyer, D. Rossetti, and M. Hsueh. Measurement and modeling of computing reliability as affected by system activity. *ACM Transactions on Computer Systems*, 4:214–237, August 1986.

[6] R. Iyer and D. Tang. Experimental analysis of computer system dependability. Technical Report CRHC-93-15, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1993.

[7] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson. Fault injection into VHDL models: The MEFISTO tool. In *24st International Symposium on Fault-Tolerant Computing*, 1994.

[8] D. Jewett. Integrity S2: A fault-tolerant UNIX platform. In *21st International Symposium on Fault-Tolerant Computing*, pages 512–519, Montreal, Canada, June 1991.

[9] G. Kanawati, N. Kanawati, and J. Abraham. Ferrari: A fault and error automatic real-time injector. In *Proc. 22nd International Symposium on Fault-Tolerant Computing*, Boston, Massachusetts, 1992.

[10] W.-L. Kao and R. K. Iyer. Define: A distributed fault injection and monitoring environment. In *Proceedings of IEEE Workshop on Fault-tolerant Parallel and Distributed Systems*, June 1994.

[11] W.-L. Kao, R. K. Iyer, and D. Tang. FINE: A fault injection and monitoring environment for tracing the unix system behavior under faults. *IEEE Transactions on Software Engineering*, 19:1105–1118, November 1993.

[12] H. Rosenberg and K. Shin. Software fault injection and its application in distributed environment. In *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing*, France, June 1993.

[13] Z. Segall, D. Vrsalovie, et al. FIAT-fault injection-based automated testing environment. In *18th International Symposium on Fault-Tolerant Computing*, pages 102–107, 1988.

[14] L. Young and R. Iyer. Error latency measurements in symbolic architectures. In *AIAA Computing in Aerospace 8*, pages 786–794, Baltimore, Maryland, October 1992.

[15] L. Young, R. Iyer, K. Goswami, and C. Alonso. A hybrid monitor assisted fault injection environment. In *Proceedings of the Third IFIP Working Conference on Dependable Computing for Critical Applications*, pages 163–174, Mondello, Sicily, Italy, September 1992.