To appear in FTCS-26, Sendai, Japan, June 1996

An Approach towards Benchmarking of Fault-Tolerant Commercial Systems

Timothy K. Tsai and Ravishankar K. Iyer
Center for Reliable and High Performance Computing
Coordinated Science Laboratory
University of Illinois, Urbana, IL 61801
{ttsai,iyer}@crhc.uiuc.edu

Doug Jewett
Tandem Computers, Inc.
Austin, TX 77828
dej@mpd.tandem.com

Abstract

This paper presents a benchmark for dependable systems. The benchmark consists of two metrics, number of catastrophic incidents and performance degradation, which are obtained by a tool that (1) generates synthetic workloads that produce a high level of CPU, memory, and I/O activity and (2) injects CPU, memory, and I/O faults according to an injection strategy.

The benchmark has been installed on two TMR-based prototype machines: TMR Prototype A and TMR Prototype B. An implementation for a third prototype, is based on a duplex architecture, is in progress. The results demonstrate the utility of the benchmark in comparing the system-level fault tolerance of these machines and in providing insight into their design. In particular, the benchmark shows that Prototype B suffers fewer catastrophic incidents than Prototype A under the same workload conditions and fault injection method. However, Prototype B also suffers more performance degradation in the presence of faults, which might be an important concern for time-critical applications.

1. Introduction

Fault tolerance is an important issue in many computing applications. For instance, mission critical aerospace or railroad control systems need to ensure the safety of passengers, and banking and telecommunications systems must offer high availability to prevent financial losses. This paper presents an approach towards benchmarking of fault-tolerant commercial systems. A fault tolerance benchmark is a metric that characterizes the fault tolerance of a system, along with the specification of the procedure for obtaining that metric. The benchmarking of fault-tolerant systems is the execution of the procedure for obtaining the benchmark metric. To perform benchmarking, a benchmark tool must be created. This tool is used to obtain the metric by injecting faults throughout a fault-tolerant system and generating a workload that causes those faults to result in fault-tolerant activity. The significance of the measured metric must be viewed with consideration for the the fault models upon which the fault injection is based.

This paper describes a fault tolerance benchmark that has been implemented on several fault-tolerant systems. The benchmark tool consists of a synthetic program that generates CPU, memory, and I/O activity based upon user specifications and injects CPU, memory, and I/O faults according to a specified injection strategy. The program can be configured to execute with different workload specifications (e.g., CPU-intensive workloads, I/O-intensive workloads, etc.) and with different fault injection strategies. The injection strategies include random selection of fault parameters (such as time and location) from distributions (such as exponential or normal) and selection of fault parameters based on workload activity measurements such that faults are injected into components undergoing high activity, thus ensuring a high level of fault activation and propagation (presented as *stress-based injection* in [11]). Two metrics are provided to quantify the system fault tolerance using the proposed benchmark: (1) the number of catastrophic incidents and (2) performance degradation.

The benchmark has been fully installed on two Tandem prototype machines, which will be referred to in this paper as (1) TMR Prototype A and (2) TMR Prototype B. An implementation is in progress for a third Tandem prototype machine (Duplex Prototype C), which is based on the Tandem ServerNet architecture[1]. A brief comparison of the three machines is given in Table 1. Because these machines are prototypes, the results are not necessarily representative of production systems. The benchmark results are given in Table 2 for a mixed (CPU, memory, and I/O activity) workload and a stress-based injection strategy.

Table 2 shows benchmark results based on 50 runs of the benchmark program. The table shows that the same workload and fault injection method produces four catastrophic incidents in Prototype A, compared with none for Prototype B. This indicates that Prototype B, which is a later instantiation of the same architecture, is indeed more fault-tolerant than Prototype A, which is a very early prototype system.

Table 1. Comparison of Target Systems

	TMR Prototype A	TMR Prototype B	TMR Prototype C
CPU	MIPS R3000	MIPS R4400	MIPS R4400
Architecture	TMR CPUs	TMR CPUs	Pair of lock-stepped CPUs
Memory	8 MB local 32 MB global	128 MB local 96 MB global	256 MB
CPU-I/O connection	Replicated bus		High-speed router network
Main fault tolerance	Voting on global memory accesses and interrupts		Self-checking components; ECC memory

Note: These systems are prototypes that are not necessarily representative of production systems.

However, the performance of Prototype B in the presence of faults suffers more than Prototype A, as shown by the performance degradation measure. The performance degradation under faults may be an important consideration for time-critical applications. The third row of Table 2 shows the average number of faults needed to cause a catastrophic incident. These injections were carried out with a series of faults that the machine was not specifically designed to tolerate, and thus a catastrophic incident ultimately resulted. The results indicate that a catastrophic incident occurred for Prototype B with fewer faults than for Prototype A. A detailed examination of these measurements is presented later in this paper. However, the numbers in Table 2 demonstrate the utility of the benchmark in comparing different fault-tolerant machines.

Table 2. Fault-Tolerance Benchmark Results

	Prototypes		
Measure	A	B	C
Catastrophic incidents	4	0	—
Performance Degradation	0.000348	0.008096	—
Avg. injections to catastrophic incident	10.4	4.1	15.8

Related work is discussed in Section 2. Section 3 describes the benchmark procedure. The benchmark program is described in Section 4. Detailed benchmark results are given in Section 5. Section 6 addresses the applicability of the benchmark to different systems. Conclusions, including a discussion of the advantages and disadvantages of the benchmark, are given in Section 7.

2. Related work

There are several different approaches to fault injection. A detailed discussion can be found in [4]. One important approach is *software-implemented 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 3. FIAT injects faults into the user process image in memory. FERRARI uses software traps to inject faults. DEFINE is a distributed tool capable of emulating software faults as well as hardware faults. DOCTOR is used to validate dependability mechanisms and has been used on a distributed real-time system. Table 3 only contains tools that have been developed by universities. However, many companies have also developed internal SWIFI tools.

Table 3. Software-Implemented Fault Injection Tools

Name	System Under Test	Fault Types (mem=memory, comm=communications)
FIAT [9]	IBM RT	CPU, mem, comm
FERRARI[6]	Sun	CPU, mem, bus, comm
DEFINE [7]	Sun	CPU, mem, bus, SW, distributed
DOCTOR[3]	HARTS	CPU, mem, comm
FTAPE[11]	Tandem S2	CPU, mem, disk

In contrast to these tools, FTAPE performs fault injection on fault-tolerant machines. In order to inject faults, the normal fault-tolerant design must be bypassed. For instance, a fault injected into a duplicated memory must be placed in only one of the two memories. FTAPE also adds a synthetic workload generator and a workload activity measurement tool that enables the fault injector to inject faults based upon dynamic workload measurements. This type of stress-based injection ensures that a high level of fault-tolerant activity results from the faults injected. The purpose for the program is to develop a fault tolerance benchmark, which measures the ability of a system to tolerate faults, and also measures the effectiveness of the error detection and recovery mechanisms by quantifying the performance degradation due to faults.

Relatively little research has been performed in the area of fault tolerance benchmarks. One area that has received some attention is robustness testing. Siewiorek et al. [10] addressed the development of a benchmark for measuring system robustness. *Crashme*[8] is a program that executes multiple processes consisting of random data masquerading as executable code. Many illegal instruction, segmentation fault, and other exceptions are generated, which should be handled by the operating system. The objective of *crashme* is to test whether a user program executing ille-

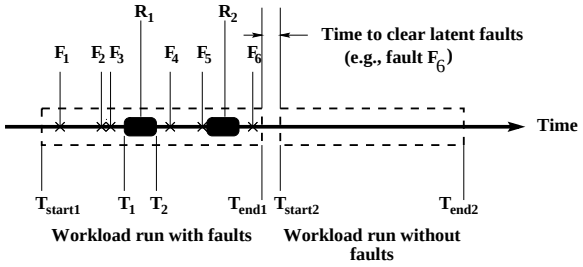
gal instructions is able to crash the operating system. *CMU-Crashme*[8] is a similar type of program, but instead of executing random instruction code, it passes random parameter values to random system calls. Dingman[2] builds on the crashme programs to create a robustness benchmark for a fault-tolerant system. The method of testing involves the execution of system calls with incorrect parameters, which might represent software faults or data corruption. The response of the system (including correct operation and several levels of failure) is then measured. This work is valuable because it provides a method for quantifying the system robustness, i.e., the ability of the system to tolerate unexpected conditions caused by user programs. The method yields numerical results which can be used to compare different systems. As the authors point out, the objective is not per se to measure how well a system responds to faults that the system is expected to tolerate.

This paper presents an approach towards benchmarking of fault-tolerant systems and a specific implementation of benchmark on several systems. While it is true that this benchmark has some deficiencies, it does represent an initial effort that satisfies the definition of a fault tolerance benchmark given in the introduction and should motivate the discussion for the development of improved benchmarks. A discussion of the specific advantages and disadvantages of our benchmark can be found in Section 7.

3. Fault tolerance benchmark

Currently there is no accepted benchmark for fault tolerance. One of the goals of this research is to develop a benchmark for dependable systems. As pointed out in the introduction, our proposed fault tolerance benchmark consists of a synthetic program that generates CPU, memory, and I/O activity based upon user specifications and injects CPU, memory, and I/O faults according to a specified injection strategy. Currently the benchmark has been completely ported to two Tandem prototype systems: (1) TMR Prototype A and (2) Tandem Prototype B. The port to a third Tandem prototype system, Duplex Prototype C, is in progress. The first two are based on the same architecture, and complete results are available for both. The benchmark is being ported to Duplex Prototype C to demonstrate the portability of the benchmark to a widely different architecture (as well as challenges associated with such a port), and the benchmark also produces some results which provide insight into the machine's design.

The general method for the benchmark is illustrated in Figure 1, which shows a single "run." The results for the benchmark are obtained from a series of runs with different workloads and faults. Each run consists of two executions of the same workload. During the first execution of the workload, faults are injected, with parameters based on the "stress-based injection method," which injects faults into



The complete benchmarking procedure consists of multiple executions of the above scenario.

Figure 1. One "run"

the system areas (e.g., CPU, memory, I/O) experiencing the greatest workload activity. For example, Figure 1 shows that faults F_1, F_2, and F_3 are injected as the workload executes. An error is detected at time T_1, and a recovery action R_1 is initiated and completes at time T_2. After recovery succeeds, faults are again injected until another error detection and recovery. This cycle of faults, detections, and recoveries is repeated until the workload completes. It is possible that a recovery action is initiated before T_{end1} but does not finished until after T_{end1}, but T_{end1} is still used in the calculation of the performance degradation. After this first execution of the workload, latent faults are cleared from the system by forcing all components that have been injected to undergo recovery. Then, the workload is executed a second time, but this time without faults. This dual workload execution allows the measurement of performance degradation, which is calculated as

$$\begin{matrix}\text{Performance}\\ \text{Degradation}\end{matrix} = \frac{1}{n}\left(\frac{T_{\text{end1}} - T_{\text{start1}}}{T_{\text{end2}} - T_{\text{start2}}} - 1\right),$$

where n is the number of faults injected. To compare the two workload execution times, the time overhead of the fault injector must be accounted for both workloads. During the second execution of the workload (i.e., without faults), this is accomplished by executing the fault injector without actually injecting faults (i.e., this can be viewed as injecting a 0-bit fault). If a catastrophic incident occurs, then the run is attempted again to achieve the same number of runs without catastrophic incidents on each system from which the performance degradation can be calculated. Note that a second execution of the same run may not result in exactly the same result because the faults may not be injected at exactly the times relative to the workload and thus fault activation and propagation will be different. This explains why a second execution of the same run does not necessarily recreate a catastrophic incident. A method to repeat catastrophic incidents based on the workload and faults injected is currently under development.

Our proposed benchmarking procedure is comprised of two phases. Phase 1 verifies that the machine tolerates expected faults correctly and also evaluates the impact of the fault-tolerant implementation on performance by injecting

faults that should be tolerated by the system. Phase 2 evaluates the reaction of the system to fault conditions that it may not be designed specifically to handle and thus demonstrates the degree of fault-tolerance beyond what is expected. In general, the difference between the two phases lies in the location of injected faults. For phase 1, faults are injected into only one component of a redundant set (e.g., only one CPU in a TMR set). For phase 2, faults are injected into all components of a redundant set.

It is important to note that the rate of fault injection is much higher than the rate of operational fault occurrence. Thus, the fault-tolerant mechanisms for these systems are stressed much more than with a more realistic fault rate. However, this higher fault rate is applied to all tested machines and is needed to produce results in a reasonable amount of test time.

3.1. Phase 1

For phase 1 injections, three types of results were obtained:

Error/Fault Ratio: The ratio of the number of errors detected to the number of faults injected. The number reflects the extent to which the fault-tolerant detection mechanisms were exercised. A low number indicates that the injections need to be performed more intelligently to obtain more fault-tolerant activity. The recovery due to one error detection may correct more than one error, thus lowering the error/fault ratio. For instance, if multiple faults are injected into one memory in the TMR set, then the first error detection will cause recovery for the entire memory, thus correcting all errors in that memory.

Performance Degradation: The fault injection run is repeated without injecting faults. The time for the run with faults (Twf) and the time without faults (Twof) are measured.

Catastrophic Incidents: The number of catastrophic incidents.

3.2. Phase 2

During phase 2, faults are injected which may not necessarily be tolerated by the fault-tolerant system. The purpose of these injections is to determine how well the system under test can handle faults which it is not specifically designed to tolerate. This is an additional measure which can be used in a benchmark comparison of different dependable systems. Also, the study of the reaction to phase 2 faults may yield insights into the tested machines. For Prototypes A and B, faults are injected into all three CPUs in the TMR configuration. For Prototype C, faults are injected into the multiple ASICs which comprise the high-speed router network connecting the processors and I/O devices.

The next section will provide a detailed description of the benchmark program that is used to perform the above benchmarking procedure.

4. Description of the benchmark program

The benchmark program generates the workload activity and fault injection needed to test a fault-tolerant system. The workload consists of a mixture of CPU, memory, and/or I/O activity based on user specifications, and the faults are CPU, memory, and I/O faults which are injected according to a specified injection strategy. The program is called FTAPE (Fault Tolerance And Performance Evaluator) and is based on the tool described in [11]. A general block diagram of the program is given in Figure 2. The program consists of three main parts: FI (the fault injector), WG (the workload generator), and Measure (the Measure tool is described in [11], but not in this paper). The fault injector and workload generator can be configured with several parameters (as shown in Figure 2) to produce different instantiations of the benchmark program. A detailed description of each part of the program follows.

The FTAPE program has been fully implemented on the Tandem TMR Prototype A and B systems. An implementation for the third system, the Tandem Duplex Prototype C, is in progress.

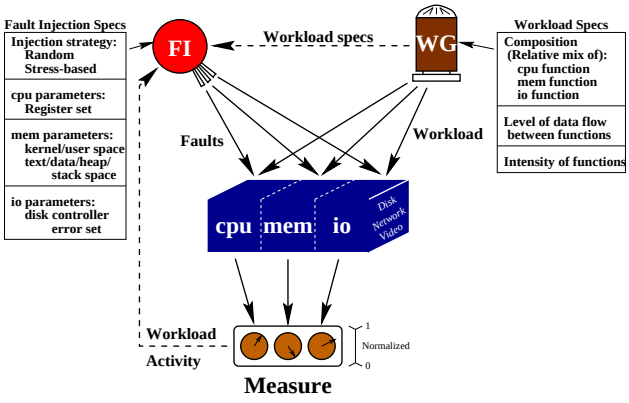


Figure 2. Benchmark Program

4.1. Fault injector

The purpose of the fault injector is to inject faults that will stimulate fault-tolerant activity. These important issues concerning the fault injector are addressed:

1. Faults must be injected throughout the entire system under test.
2. Software-implemented fault injection (SWIFI) is used because it permits a great deal of controllability over fault parameters and allows a wide variety of faults to be injected. Because no additional hardware is required, SWIFI can offer more portability than other fault injection methods, although low-level fault injection mechanisms will still need to be designed for each system.

3. To inject a fault, the system's implemented fault tolerance must be bypassed. For instance, injections into CPU registers or memory locations which are TMR-replicated must corrupt only a single component in the TMR configuration. Most SWIFI tools have not been implemented on fault-tolerant systems and therefore do not do this.
4. Portability is required, since the fault injector is part of a benchmark program that compares different machines. A large degree of portability is attained by (1) using the SWIFI fault injection method and (2) separating the fault injector into a directly portable, high-level portion and a small system-dependent, low-level portion (see Figure 3).

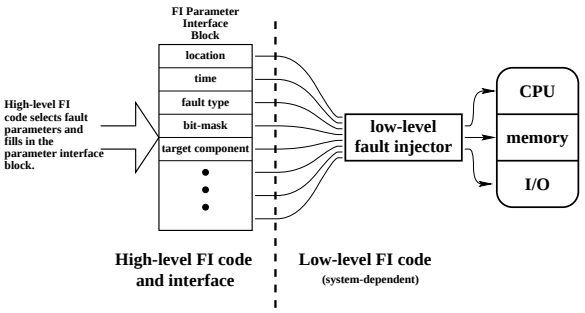


Figure 3. Fault Injector Portability Interface

Since the benchmark is used to evaluate the overall system fault tolerance, faults are injected throughout the entire system under test. FTAPE partitions the system into three main areas: *cpu*, *mem*, and *io*. These same areas are targeted by the workload generator in order to increase the chance for the injected faults to be propagated by the workload. Each area requires a different fault injection method, which are described below.

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 (*saved* registers are those registers whose values must be preserved across procedure calls) general purpose and floating point registers, the program counter, the global pointer, and stack pointer. These registers were chosen because faults in these registers have a higher chance of propagation compared to faults in other registers (e.g., temporary registers). The method of injection involves the use of the memory fault injector. A copy of the CPU register set is obtained and placed in memory. The memory fault injector is then used to corrupt the memory location corresponding to a specific register. The copy of the CPU register set in memory is then loaded back into the CPU, thus placing the corrupted contents into the appropriate CPU register.

Injection method 2: inject_mem: The memory fault models include single/ multiple bit-flip and zero/set faults in memory. Faults are targeted at heavily used parts of memory. Faults are injected by directly modifying the contents of selected memory locations. The method of injection bypasses the normal memory access mechanism. A special device driver is created. This device driver is able to corrupt the contents of a specific memory location based on a given XOR fault mask (i.e., an XOR fault mask of `0x00000001` would flip bit 0 of the memory location). For TMR-replicated memory, only one component in the TMR configuration is corrupted.

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.

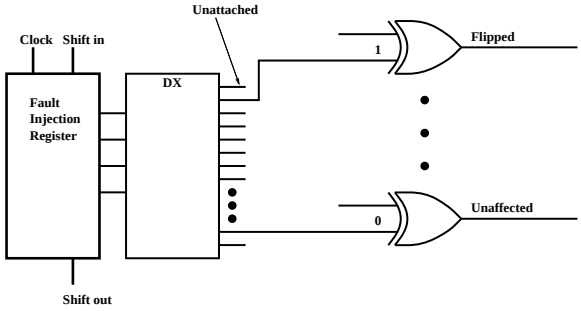


Figure 4. Prototype C Fault Injection

These three fault-injection methods have been implemented on the two TMR Prototype systems. The Tandem Duplex Prototype C is a new system which has been recently released, and for this system, initial work has utilized the built-in fault injection mechanisms (see Figure 4); the software-implemented methods will be included in the future. For the built-in fault injection, faults are injected by means of a set of XOR gates that were inserted into the original design. One input of each XOR gate is connected to a decoder, which then activates one gate. The logical value of the node on which the gate is inserted is then effectively flipped. Thus, activation of an XOR gate is accomplished by setting the appropriate value into the fault injection register which drives the decoder. Faults are injected into the ASICs that comprised the interconnect between the processors and the I/O devices. Three types of ASICs exist. The router ASIC (ASIC R) forms the backbone for the interconnect. ASIC M serves as the interface between the processors and the routers, and ASIC I serves as the interface between the I/O devices and the routers.

Portability: Since one of the main purposes for the FTAPE program is the comparison of different fault-tolerant machines, portability of the program is an important issue. The program software has been written based on the UNIX

operating system. Thus, the software code should port easily to most UNIX-based systems. The code for the implementations on the prototype machines required a total of 672 lines of C-language code out of 9172 to be modified. However, certain functions, such as the low-level fault injection mechanism will have to be written specifically for each machine. These mechanisms are listed below. Once these functions are identified, the fault injector software has been written to facilitate integration of these functions: (1) a virtual to physical memory address translation function, (2) a function returning the virtual addresses for the text and data regions for a specified process, and (3) a low-level fault injector. Also, the three injection methods methods described in this section refer to specific types of CPU registers (e.g., a global pointer) and I/O devices (e.g., SCSI) which might not be defined for all machines. These specifications need to be written in a manner to include all machines, but are sufficient to describe the current implementations.

Figure 3 shows the fault injector code, which has been separated into a high-level part and a system-dependent low-level part. The high-level code, which is directly portable to most UNIX systems, selects fault parameters (such as fault time and location) and places those values into the FI parameter interface block. The low-level fault injector, which is implemented by a device driver dedicated to fault injection, uses the values in the interface block to inject the actual fault into the appropriate system component. The system under test is viewed as being comprised of logical components (CPU, memory, and I/O). By viewing the system as being comprised of logical components, the same fault sets can be used for different systems. This separate view of logical and physical components is supported by the separation of the fault injector into high and low-level as shown in Figure 3. The high-level fault injector code injects faults into logical components, while the system-dependent, low-level code determines how injections into logical components maps to injections into physical components.

The device driver injects faults in such a way as to circumvent the normal fault-tolerance. For example, memory faults in a TMR system are written to a single memory as opposed to all three. The functions other than the low-level fault injector should be available for most UNIX-based systems and simply need to be identified. These functions may be available either as an option in the `/proc` file system or as a privileged function in the kernel, which can be accessed through a device driver.

4.2. 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: `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: `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. An attempt is made to force accesses to the physical memory, for instance, by making the size of the array larger than the size of the data cache.

Function 3: `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.

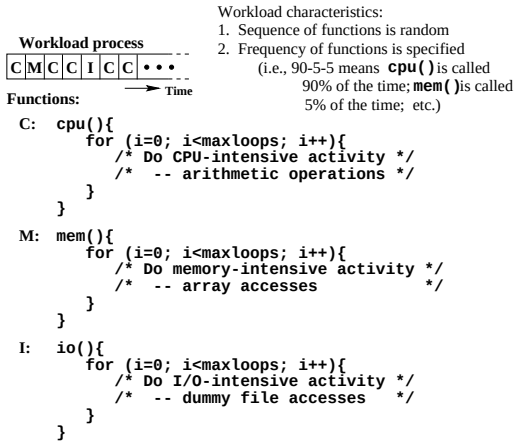


Figure 5. Assembling a Workload Process from Functions

4.3. Phase 1

The three types of workload functions are assembled into a workload process. Figure 5 illustrates how this is done. The sequence of workload functions is randomly chosen. The frequency of each function type is specified before the workload starts. For instance, if a CPU-intensive workload is desired, the workload is specified to be 90-5-5, which means that `cpu()` is executing 90% of the time, `mem()` 5% of the time, and `io()` 5% of the time. The `maxloops` variable in Figure 5 is chosen from a normal distribution with a mean of one second. Since the order of function in the workload process is chosen randomly, a specification of a different random seed will produce a different workload. However, the relative frequencies of the workload functions will remain the same. For example, a 90-5-5 workload will remain CPU-intensive regardless of the random seed specified.

Table 4. Phase 1 Results

	TMR Prototype A							
Workload	Runs	Faults	Errors	E/F	Twf(sec)	Twof(sec)	PD/F	Catastrophic Incidents
33-33-33	12	66.42	11.5	0.174	1925	1882	.000494±.000266	3
5-5-90	19	41.00	27.6	0.673	3411	1596	.028003±.001853	1
90-5-5	19	22.47	4.0	0.180	2341	2305	.001012±.000361	0

	TMR Prototype B							
Workload	Runs	Faults	Errors	E/F	Twf	Twof	PD/F	Catastrophic Incidents
33-33-33	12	10.78	7.3	0.680	2377	2194	.006648±.001851	0
5-5-90	19	4.12	3.3	0.788	2224	2130	.011394±.002108	0
90-5-5	19	6.09	3.7	0.612	2224	2138	.006686±.001865	0

Note: All PD/F intervals are 95% confidence intervals for means.

Portability: The workload generator code is easily ported to any UNIX-based system. Some communication with the fault injector is based on UNIX file systems and pipes. In order to help minimize variations among different systems, a random number generator function is included. No additional functions need to be identified or written to port the workload generator.

5. Fault tolerance benchmark results

The benchmark procedure described in Section 3 was performed for the two TMR Prototype systems. The phase 1 and phase 2 results are presented below. Because the FTAPE tool has not been fully implemented on Prototype C, only phase 2 results are presented for that machine.

Phase 1 consists of injections that should be tolerated by the system; these injections are performed in order to verify that the machine tolerated expected faults correctly and to evaluate the impact of the fault-tolerant implementation on performance. The phase 1 results were obtained for only the two TMR Prototype systems. The results for Duplex Prototype C are not given because the full port of FTAPE to the that machine is still ongoing. Table 4 contains these results. For each machine, the results are categorized by the workload used during the injections (5-5-90 is disk intensive, 90-5-5 is CPU intensive, and 33-33-33 is mixed). The number of faults and errors are an average of the total number of faults injected and errors detected for each workload type. The error/fault ratio is given in the "E/F" column, the performance degradation is given in the "PD/F" columns, and the number of catastrophic incidents is given in the "Catastrophic Incidents" column. Twf is the time in seconds for the workload program to execute with faults; Twof is the same measurement without faults. The Twf and Twof values for Prototype B in Table 4 are larger than those for Prototype A because the total amount of workload activity was increased to account for the faster processor speed. The workload composition (relative mix of CPU, memory, and I/O activity) remained the same.

Table 4 contains some important results. Most important is the benchmark result that 4 catastrophic incidents occurred for Prototype A and none for Prototype B. This indicates the high degree of fault tolerance for Prototype B compared to Prototype A, which is to be expected, since Prototype A is a very early prototype and the Prototype B is a newer design which contains corrections to the problems of Prototype A. In particular, improvements were made to the CPU and TMRC (see [5]) which affect fault containment and isolation. The mean catastrophic incident probability for Prototype A is 0.08 with a 95% confidence interval of [0.004,0.156]. If the mean catastrophic incident rates for Prototypes A and B are α_1 and α_2 respectively, then a hypothesis test for $H_0 : \alpha_1 = \alpha_2$ at a 0.01 level of significance shows that $z = 7.071$ and H_0 is rejected. Thus, more catastrophic incidents occur for Prototype A at a statistically significant level.

The performance degradation numbers are generally higher for Prototype B, mainly because it contains more memory and hence requires more time for recovery. (Because the CPUs are in a TMR configuration, recovery entails copying the data in the memories of the good CPUs to the bad CPU. The more memory that needs to be copied, the longer recovery is.) The exception is the performance degradation for the 5-5-90 (or I/O-intensive) workload, which is much greater for Prototype A than for Prototype B. The difference here is the extent to which recovery of a disk mirror affects the workload execution. Prototype A uses an older disk file system, which incurs a greater loss of disk bandwidth during recovery. This effect can be easily seen when the 5-5-90 workload is executed while mirror recovery is constantly ongoing. Table 5 compares the execution times for an I/O-intensive workload with simultaneous mirror recovery and without, for Prototypes A and B. The table shows that Prototype A suffers a much higher performance hit when mirror recovery occurs. Statistical hypothesis testing shows that the mean performance degradation numbers for each type of workload in Table 4 are statistically different when comparing Prototypes A and B. Indeed, the hypothesis that the mean performance degradation for a particular workload is the same for both machines can be easily rejected even at a 0.005 level of significance.

Table 5. Demonstration of Disk Bandwidth During Mirror Recovery

	Time with mirror recovery	Time without mirror recovery	Ratio
Prototype A	5539 secs	1796 secs	3.08
Prototype B	2310 secs	1878 secs	1.23

5.1. Phase 2

Phase 2 consists of injections that may not necessarily be tolerated by the system; these injections are performed in order to evaluate the reaction of the system to fault conditions that it may not be designed specifically to handle and thus demonstrates the degree of fault-tolerance beyond what is expected. Table 6 gives some phase 2 results for the three systems tested. The table contains two types of measures: (1) the average number of faults and time in seconds to the initial error detection and to a catastrophic incident, and (2) the type of catastrophic incident. For all phase 2 injections, a catastrophic incident is defined as the condition which prevents successful completion of the workload and is usually manifested as a system panic or hang. The "Others" category in the last row of Table 6 includes a premature workload process termination and an automatic system reboot. The fault injections for Prototypes A and B consist of bit-flips in CPU registers and memory. For Prototype C, faults are injected using the fault injection mechanism built into the three ASICs described near the end of Section 4.1.

Table 6. Phase 2 Results

Measure	Prototype Machines A	B	C
Avg. injections to initial detection	6.6	3.0	2.1
Avg. injections to catastrophic incident	10.4	4.1	15.8
Avg. time to catastrophic incident (sec)	85.8	31.9	467.5
Time from initial detection to catastrophic incident (sec)	18.5	28.0	428.7

Catastrophic incident manifestation	A	B	C
Panics	7	7	3
Hangs	1	3	7
Others	2	0	0

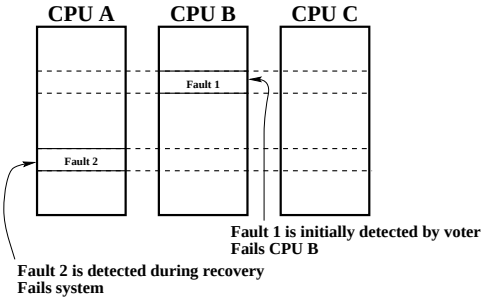


Figure 6. Phase 2 Catastrophic Incident Scenario

Of the first type of measures, two are especially significant: the average time to catastrophic incident and the time from initial detection to catastrophic incident. It is interesting to note that all three machines were able to continue correct operation past the first few fault injections even though the injections were intended to cause catastrophic incidents. Also, all three machines detect their first error within an average of one minute. However, Prototype C is able to continue operation for a much longer period of time after the initial error detection. Part of the explanation for this effect is the different type of fault that is injected for Prototype C. However, a more important answer lies in the type of recovery attempted by TMR Prototypes A and B. Figure 6 depicts a situation where two phase-2 faults have been injected into the memories of CPU A and CPU B. The fault in the memory of CPU B is discovered first by the memory voter, which then fails CPU B. After CPU B passes a diagnostic hardware check, recovery begins. The memories of CPUs A and C are first compared and copied to the memory of CPU B on a block-by-block basis. When the block containing Fault 2 is reached, the mismatch between CPUs A and C is detected, and the system fails.

Table 7. Phase 2 Results for Duplex Prototype C (with no attempted recovery)

Measure	ASICs All 3	M	R	I
Avg. injections to initial detection	2.1	6.0	1.0	5.5
Avg. injections to catastrophic incident	15.8	15.1	6.9	6.9
Avg. time to catastrophic incident (sec)	467.5	304.8	199.2	139.9
Avg. time from initial detection to catastrophic incident (sec)	428.7	138.8	194.6	44.4

Catastrophic incident manifestation	All 3	M	R	I
Panics	3	9	5	5
Hangs	7	0	5	5
Others	0	1	0	0

However, there is one sequence of events that avoids the described catastrophic incident scenario and upon which an alternative, albeit more complex, hardware recovery implementation might be developed. A memory scrubber is implemented for both TMR Prototypes A and B. When the memory scrubber discovers an error in a CPU, it performs a memory block copy, but only for the affected memory block. If another error is present in a different memory block in another CPU, that error is not corrected until the scrubber discovers it later. Unfortunately, the memory scrubber discovers far fewer errors than the memory voting mechanism, because the memory scrubber is specifically configured to slowly traverse memory in order to minimize its impact on performance through memory contention. However, this type of strategy could be used to improve the recovery

of CPUs downed by the memory voter. Referring again to Figure 6, when Fault 2 is encountered during the recovery process and a mismatch between CPUs A and C is detected, the copy of that memory block for CPU B can be used to determine that CPU A is in error. CPU A can then be corrected at that time, and the recovery for CPU B can continue. This recovery process only fails when the memory for all three CPUs disagree at exactly the same memory location, which should be extremely unlikely. One drawback to this recovery method is the reliance on data in a failed component for recovery.

Table 7 shows the results of injections into different ASICs in Prototype C. For these experiments, no recovery was attempted, thus causing the system to fail in every case. The fully operational system will, of course, be capable of recovery from errors. As previously mentioned in Section 4.1, these ASICs comprise the high-speed interconnect between the processors and I/O devices. ASIC R is a router chip that provides the backbone for the interconnect, ASIC M interfaces the processors to the router ASICs, and ASIC I interfaces the I/O devices to the router ASICs. These results provide insight into the designs of the different types of ASICs and in particular highlight specific areas that should receive more attention. For example, ASIC I fails shortly after detecting the first error whereas ASIC M is able to forestall a catastrophic incident for a longer period of time. Interestingly, when faults are injected into all three types of ASICs, the mean time to catastrophic incident is longer than for faults into one specific type of ASIC. However, when such an incident finally occurs, more hangs than panics result, indicating that such incidents are more severe.

As a final note, one of the phase 2 injections into Prototype C caused a system hang after a single fault had been injected. This discovered a design defect, which illustrates the utility of the fault injection method in providing feedback to designers.

6. Application of benchmark to different architectures

An important consideration is the applicability of the proposed benchmark to systems that are very different. A benchmark consists of a metric and the procedure needed to obtain that metric. The benchmark described in this paper specifies the procedure for the obtaining the metric in the form of a benchmark program. The benchmark program was initially implemented on Prototype A. Porting the benchmark to Prototype B was easily performed because both systems are based on the same architecture. Thus, the physical components, such as the TMR CPUs, that are targeted by the workload generator and fault injector are present on both machines. Although systems may differ in the design of their physical components, all systems possess certain vital logical components, such as a processor, memory, and I/O subsystem. The faults and workloads generated by the benchmark program target these logical components. For example, faults are injected into the logical processor, whether the actual design is TMR or lock-stepped duplication. In both the TMR and lock-stepped designs, a single fault would be injected into one out of either two or three duplicated components.

When systems with different architectures are being tested, the benchmarking procedure needs to be specified in terms of fault models and workloads targeted at logical components. By viewing the system as being comprised of logical components, the same workload and fault sets can be used for different systems. This separate view of logical and physical components is supported by the separation of the fault injector into high and low-level as shown in Figure 3. The high-level fault injector code injects faults into logical components, while the system-dependent, low-level code determines how injections into logical components map to injections into physical components. Prototype C is based on a duplex architecture that is very different from the TMR architecture. All of the high-level benchmark code has been ported to Prototype C, as well as a portion of the low-level code, which has allowed some results to be obtained.

One concern regarding the injection of faults into logical components is fairness. Is the injection of a single fault into one TMR CPU comparable to the injection of a single fault into one CPU of a gracefully degrading 3-CPU processor? Similar questions also arise for performance benchmarks. For instance, is the execution of a large program in a small-cache system comparable to that for a system with a larger cache? These questions can be answered in the positive because one design is actually better able to handle the specific fault or program size mentioned. Fairness results by properly selecting a set of faults and workloads that consider the desirability of each design. In certain cases, the consideration is obvious. For example, a larger cache is almost always better for performance than a smaller cache. Other cases are more difficult. For example, two-CPU faults injected into the gracefully degrading 3-CPU processor result in fewer catastrophic incidents (because only a single processor is required for operation) but more performance degradation (because each additional failed CPU degrades the overall performance). Thus, care has to be taken in the selection of workload and fault inputs for the benchmark program.

7. Conclusions

This paper presents an approach for fault tolerance benchmarking. The benchmark consists of two metrics: number of catastrophic incidents and performance degradation. These metrics are obtained by executing the FTAPE fault injection tool. The benchmark has been fully implemented on two Tandem fault-tolerant prototype machines: (1) TMR Prototype A and (2) TMR Prototype B. An implementation for a

third Tandem prototype machine, which is based on a duplex architecture, is under development. The experiments performed on these prototype machines do not necessarily characterize the operation of production machines, but the experiments do demonstrate the utility of this benchmark in comparing the high-level dependability of different systems. The results show that Prototype B is more fault-tolerant than Prototype A, in that it suffers fewer catastrophic incidents under the same workload conditions and fault injection method. However, Prototype B also suffers more performance degradation in the presence of faults, which might be an important concern for time-critical applications.

The area of fault tolerance benchmarks is just emerging. This work represents an effort to propose and demonstrate a fault tolerance benchmark and, to the authors' knowledge, is the first to develop a benchmark that represents the overall fault tolerance of a system.

Several significant ideas are embodied in our benchmark: The system under test is viewed as being composed of logical components (such as CPU, memory, disk, etc.), which not only provides targets for fault injection, but also aids portability by defining fault injection targets in term of components that all systems possess. The fault injector and workload generator are both modular, which allows the benchmark to be easily reconfigured if the underlying fault models to be tested are redefined. Also, the use of a performance degradation measurement is unique.

Because our benchmarking approach is the first effort for fault-tolerant computing, some deficiencies exist: The set of fault models used for injection is not complete. To be more useful, studies should be conducted to determine a realistic set of fault models and workloads. Also, the fairness of the fault models and workloads in discriminating among different machines needs to be studied further. While attention has been paid to portability, the benchmark has only been ported to two machines of the same architectural family, and thus, increased portability for widely differently hardware and software platforms needs to be addressed. The outcome of each benchmark run is not repeatable, although the average of a series of runs is repeatable. Finally, the use of SWIFI techniques for fault injection imposes some limitations on the types of faults that can be included in the fault model set.

8. Acknowledgments

Thanks are due to Bob Horst and Luke Young of Tandem Computers, for many useful coments and suggestions, and also to Sandy Hsueh for her inputs during the course of 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] W. E. Baker, R. W. Horst, D. P. Sonnier, and W. J. Watson. A Flexible ServerNet-Based Fault-Tolerant Architecture. In *Proc. 25th Int. Symp. on Fault-Tolerant Computing (FTCS-25)*, pages 2–11, Pasadena, CA, USA, June 1995.

[2] C. P. Dingman, J. Marshall, and D. P. Siewiorek. Measuring Robustness of a Fault Tolerant Aerospace System. In *Proc. 25th Int. Symp. on Fault-Tolerant Computing (FTCS-25)*, pages 522–527, Pasadena, CA, USA, June 1995.

[3] S. Han, K. G. Shin, and H. A. Rosenberg. DOCTOR: An Integrated Software Fault Injection Environment for Distributed Real-Time Systems. In *Proc. Int. Computer Performance and Dependability Symp.*, pages 204–213, Erlangen, Germany, April 1995.

[4] 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.

[5] D. Jewett. Integrity S2: A Fault-tolerant Unix Platform. In *Proc. 21st Int. Symp. on Fault-Tolerant Computing (FTCS-21)*, pages 512–519, Montreal, Canada, June 1991.

[6] G. Kanawati, N. Kanawati, and J. Abraham. Ferrari: A Fault and Error Automatic Real-Time Injector. In *Proc. 22nd Int. Symp. on Fault-Tolerant Computing (FTCS-22)*, Boston, MA, USA, June 1992.

[7] W.-L. Kao and R. K. Iyer. DEFINE: A Distributed Fault Injection and Monitoring Environment. In *Fault-Tolerant Parallel and Distributed Systems*, (D. Pradhan and D. R. Avresky, Eds.), pp. 252–259, Los Alamitos, California, USA: IEEE CS Press, 1995.

[8] A. Mukherjee and D. Siewiorek. Measuring Software Dependability by Robustness Benchmarking. Technical Report CMU-CS-94-148, Carnegie-Mellon University, 1994.

[9] Z. Segall, D. Vrsalovie, et al. FIAT-Fault Injection-Based Automated Testing Environment. In *Proc. 18th Int. Symp. on Fault-Tolerant Computing (FTCS-18)*, pages 102–107, Tokyo, Japan, June 1988.

[10] D. P. Siewiorek, J. J. Hudak, B.-H. Suh, and Z. Segall. Development of a Benchmark to Measure System Robustness. In *Proc. 23rd Int. Symp. on Fault-Tolerant Computing (FTCS-23)*, pages 88–97, Toulouse, France, June 1993.

[11] T. K. Tsai and R. K. Iyer. Measuring Fault Tolerance with the FTAPE Fault Injection Tool. In *Proc. of Performance Tools '95/MMB '95*, pages 26–40, Heidelberg, Germany, September 1995.