February 1992

UILU-ENG-92-2207
CRHC-92-04

Center for Reliable and High-Performance Computing

Do Not Remove

A HYBRID MONITOR ASSISTED FAULT INJECTION ENVIRONMENT

Luke T. Young and Ravi K. Iyer

Coordinated Science Laboratory
College of Engineering
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION
Unclassified

1b. RESTRICTIVE MARKINGS
None

2a. SECURITY CLASSIFICATION AUTHORITY

2b. DECLASSIFICATION / DOWNGRADING SCHEDULE

3. DISTRIBUTION / AVAILABILITY OF REPORT
Approved for public release;
distribution unlimited

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
UILU-ENG-92-2207 CRHC-92-04

5. MONITORING ORGANIZATION REPORT NUMBER(S)

6a. NAME OF PERFORMING ORGANIZATION
Coordinated Science Lab
University of Illinois

6b. OFFICE SYMBOL *(If applicable)*
N/A

7a. NAME OF MONITORING ORGANIZATION
Office of Naval Research & Tandem Corp

6c. ADDRESS *(City, State, and ZIP Code)*
1101 W. Springfield Ave.
Urbana, IL 61801

7b. ADDRESS *(City, State, and ZIP Code)*
800 N. Quincy St.
Arlington, VA 22217

19333 Vallco Pkwy
Cupertino, CA 95014

8a. NAME OF FUNDING / SPONSORING ORGANIZATION Joint Services
Electronics Program 7a

8b. OFFICE SYMBOL *(If applicable)*

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
N00014-91-J-1116

8c. ADDRESS *(City, State, and ZIP Code)*
800 N. Quincy St.
Arlington, VA 22217 7b

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
A hybrid monitor assisted fault injection environment

12. PERSONAL AUTHOR(S)
YOUNG, L. T. and R. K. Iyer

13a. TYPE OF REPORT
Technical

13b. TIME COVERED
FROM ________ TO ________

14. DATE OF REPORT *(Year, Month, Day)*
1992 March 04

15. PAGE COUNT
31

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
error latency, error propagation, application dependability
hybrid monitor, fault injection, TMR, RISC

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

(continued)

This paper describes a hybrid fault injection environment that can be used to evaluate the dependability of computing systems. It consists of a fault injection system, a hybrid monitor, and a supervisory system to automate the measurements. The hybrid environment combines the versatility of software injection and the accuracy of hardware monitoring. It is useful for obtaining dependability statistics and failure characteristics for a range of system components. It is also well suited for measuring extremely short error latencies, and the introduced overhead is minimal so that error propagation and control flow are not significantly affected by the presence of instrumentation. The hybrid environment can be used to obtain precise measurements of instruction-level activity that would otherwise be impossible to perform with a hardware monitor alone.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION
Unclassified

22a. NAME OF RESPONSIBLE INDIVIDUAL

22b. TELEPHONE *(Include Area Code)*

22c. OFFICE SYMBOL

83 APR edition may be used until exhausted.
All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

The utility of the fault injection environment is demonstrated by applying it to the study of a Tandem *Integrity* S2 system. Faults are injected into CPU registers, cache, and local memory. The effects of faults on individual user applications are studied by obtaining subsystem dependability measurements such as detection and latency statistics for cache and local memory subsystems. Instruction-level *fault sensitivity* and error propagation effects are also measured.

keywords: error latency, error propagation, application dependability, hybrid monitor, fault injection, TMR, RISC.

.19 continued

A Hybrid Monitor Assisted Fault Injection Environment

L.T. Young and R.K. Iyer

Center for Reliable and High Performance Computing
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign, Urbana, IL 61801

ABSTRACT

This paper describes a hybrid fault injection environment that can be used to evaluate the dependability of computing systems. It consists of a fault injection system, a hybrid monitor, and a supervisory system to automate the measurements. The hybrid environment combines the versatility of software injection and the accuracy of hardware monitoring. It is useful for obtaining dependability statistics and failure characteristics for a range of system components. It is also well suited for measuring extremely short error latencies, and the introduced overhead is minimal so that error propagation and control flow are not significantly affected by the presence of instrumentation. The hybrid environment can be used to obtain precise measurements of instruction-level activity that would otherwise be impossible to perform with a hardware monitor alone.

The utility of the fault injection environment is demonstrated by applying it to the study of a Tandem *Integrity* S2 system. Faults are injected into CPU registers, cache, and local memory. The effects of faults on individual user applications are studied by obtaining subsystem dependability measurements such as detection and latency statistics for cache and local memory subsystems. Instruction-level *fault sensitivity* and error propagation effects are also measured.

keywords: error latency, error propagation, application dependability, hybrid monitor, fault injection, TMR, RISC.

I. INTRODUCTION

Designers of highly reliable computer systems need realistic measurements in order to complete evaluations based on simulation or analytical methods. *Fault injection* is well known for its successful use in system validation and in the extraction of dependability statistics such as latency and fault detection ratio. It is difficult, however, to realistically measure the effects of faults without depending on a passive monitor. Likewise, it is difficult to measure the effects of faults on software-defined components (e.g., sections of a given application) without significantly disturbing the system under test.

This paper describes a hybrid fault injection environment, wherein faults are injected via software and the impact is measured by both software and hardware. The environment is useful for evaluating system dependability, and it has the advantage in that it introduces minimal perturbation, and provides a high degree of control over the location of faults to be injected. Faults can be injected into any location that has a physical address, e.g., CPU registers, cache, local memory, mass storage, network controllers, etc.... Faults can also be injected into locations allocated to a single, executing user program or even into the kernel, and propagation can be characterized down to the instruction level. The environment is well suited for measuring extremely short error latencies, and the introduced overhead is minimal so that error propagation and control flow are not significantly affected by the presence of instrumentation.

We illustrate the environment by applying it to the study of the Unix-based, *Tandem Integrity S2* computing system. Detection statistics and precise latency measurements for cache and local memory subsystems are obtained. We also examine instruction-level error propagation effects and measure fault isolation times.

Several key results are presented in this paper. Our findings support the design decision to preserve error correction coding in cache but not in local memory. The faults injected in the local memory subsystem of the S2 causes a CPU divergence/shutdown only 3.6% of the time. But in the cache, CPU divergence/shutdown occurs 95.0% of the time (in only the first minute). Early fault removals are due mostly to overwrites by the application; later removals are due mostly to page deallocation. This information characterizes a natural fault removal processes and could assist in the design of efficient scrubbing techniques. In cache, most detected faults are found in less than 493 microseconds and most injected faults are found in less than 501 microseconds. By comparison, it takes 51.2 seconds to scrub all of cache at the rate local memory uses. This finding supports the design decision to not perform memory scrubbing in cache. We characterize immediate error propagation effects caused by injection faults into the instruction stream of a Mips RISC processor. The impact of faults on instruction code during runtime is significant — we find that single-bit errors propagate additional errors 85.9 percent of the time. We observe that instructions differ substantially in the degree to which they are fault sensitive and affect error propagation. Since latency and detection statistics can be obtained for many applications and opcodes, this tool provides a way to characterize the dependability of an entire instruction set.

II. RELATED RESEARCH

2.1 Latency Studies

The term *error detection latency* is defined as the time that elapses between the activation of an error and its discovery. Similarly, *fault latency* is the time delay between when a fault comes into existence and when it becomes active by producing an error. In computer systems, failure rates can be elevated during a burst of system activity because errors may remain undiscovered until then. For this reason, it is generally believed that long fault and error latencies are undesirable and can have a significant impact on a computing system's reliability.

Most fault latency experiments have taken an emulator-based approach. Studies of CPU fault latency using a gate-level emulation of an avionic miniprocessor are described in [NASA81],[NASA83]. Similar experiments are reported in [Lala83],[McGo83]. A methodology for on-line testing of microprocessors and the distribution of failure detection times for those affecting the M6800 CPU die are reported in [Cour81]. In [Shin86], an indirect technique is used to estimate fault latency at the pins of the chips in the CPU of the Fault Tolerant Multiprocessor (FTMP). Because the exact moment when a fault becomes an error is not known, the technique gives only an upper bound for fault latency.

Chillarege has developed a methodology for studying error latency characteristics of medium to large computer systems in a full production environment [Chil87]. The technique is applied to the memory subsystem and employs periodic sampling by a hardware monitor. In [Mitr88], this technique was extended to a shared-memory multiprocessing system and used to calculate the risk of encountering multiple latent errors. A failure acceleration method for determining fault detection characteristics is discussed in [Chil89]. Because this study used periodic sampling, the discovery times of only **permanent faults** could be measured. In [Youn91], a hybrid monitor approach to measuring error latency was applied to a TI Explorer II Lisp machine. The method is based on simulation of the error discovery process taken from a continuous trace of software-selected locations.

2.2 Software Fault Injection Studies

A number of studies performed at Carnegie Mellon University have centered around FIAT, an automated environment for injecting faults in a distributed system [Sega88]. The FIAT environment utilizes *software implemented fault injection* (SWIFI) to emulate various hardware faults [Czec91]. The emulation of hardware failure manifestations by automatic instruction substitutions is described in [Youn92]. Another automated fault injection environment, FERRARI, was able to emulate faults in hardware components such as opcode decoding circuitry, program control units, data registers, ALU, and address and data buses [Kana92]. It gave a user control over the location and duration of an injection by inserting trap instructions that remove themselves after recreating the same effects that a given hardware fault would have.

Simulation based approaches have been taken in studying the effects of fault injection. In [Lome86], a simulation environment was used to study error propagation from the gate to chip level. FOCUS, a simulation environment to conduct fault sensitivity analysis of chip-level designs, is described in [Choi89]. Another simulation environment, DEPEND, studies the effects of faults at the system level [Gosw91]. Instruction-level simulations are used to supplement SWIFI in [Czec91]. Such methods are useful at the design stage, but they fail to provide a complete environment for fault propagation. They cannot, for example, include the effects of paging, various interrupts, scheduling, I/O, etc....

This paper describes a hybrid fault injection environment that can be used to evaluate the dependability of computing systems. The environment combines the versatility of software injection and monitoring with the accuracy of hardware monitoring. Traditional SWIFI methods are used to inject faults into CPU registers, cache, and local memory of a test system. The environment consists of a fault injection system, a hybrid monitor, and a supervisory system to automate the measurements. The hybrid monitor is further divided into hardware and software monitors. Details of the hybrid environment provided in the following section.

III. EXPERIMENTAL ENVIRONMENT

3.1 The Hybrid Fault Injection Environment

Figure 1 illustrates the subsystems that make up the hybrid fault injection environment. It consists of a fault injection system, a hybrid monitor system to measure the effects of injected faults, and a

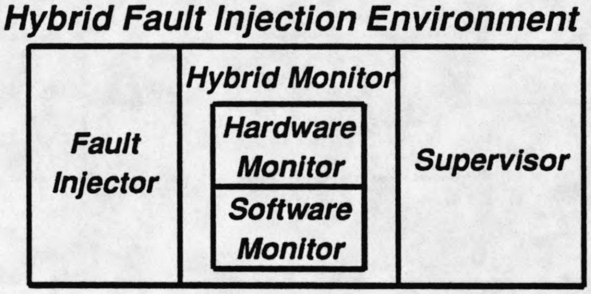


Figure 1: Hybrid Fault Injection Environment

supervisory system to automate the measurements. The hybrid monitor system is further divided into a hardware monitor and a software monitor. Figure 2 illustrates how these systems are physically situated. The *fault injector* and *software monitor* execute on the *test system*, while the *supervisor program* executes on the *control host*. Probes attach the hardware monitor to the address/data backplane of the test system so that the monitor can analyze and record the signals generated. Communication between the supervisor and the hardware monitor takes place over an RS-232 or GPIB connection.

The function of the environment is to perform experiments that repeatedly inject faults and record observations. The environment introduces faults into the test system during the execution of a *target program*, measures the effects of that fault, and returns the test system to conditions present prior to fault injection. These operations form a single *observation loop*. Figure 3 illustrates the control flow of an experiment.

To use the hybrid fault injection environment, one must specify the *target program* to run (with data), the number of times to repeat an observation loop, the number of faults to generate per

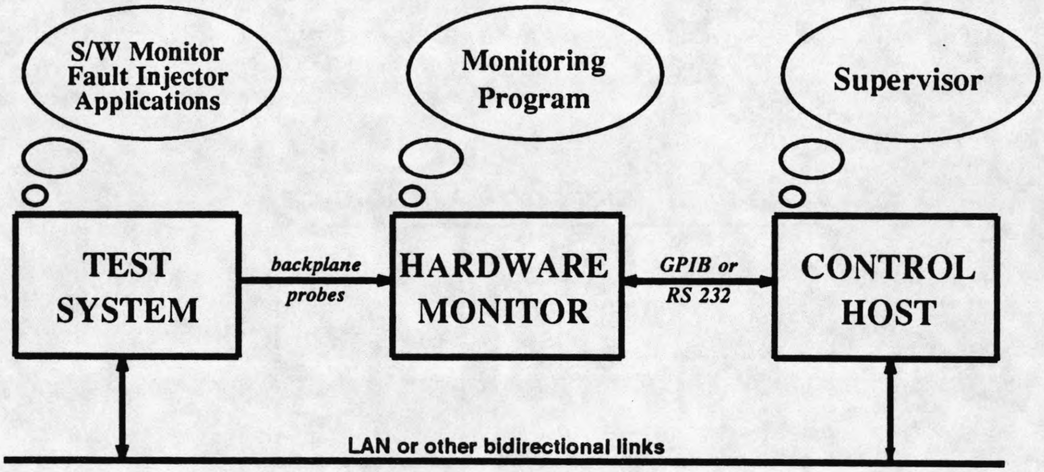


Figure 2: Physical Layout of Hybrid Fault Injection System

observation, and termination conditions (typically a time limit or CPU fail). The *target program* can be any user program desired. After the target program has been started, physical addresses of locations to fault are randomly generated by the supervisor and presented to the hardware monitor. While the hardware monitor is reconfiguring, the software monitor determines which virtual addresses have been allocated to the target program. The software monitor then converts these virtual addresses to physical addresses and determines which physical addresses match those generated by the supervisor. If no match is found, the supervisor must generate another random set of physical addresses and restart the hardware monitor. Thus, the *supervisor* and the *software monitor* work in parallel over the network to generate random locations in the test system to fault.

The matched physical addresses are then passed to the *fault injector*. Although typical experiments constrain fault injections to portions of memory allocated to the target program, the hybrid environment can generate faults within **any** location that has a physical address. e.g., locations may include the kernel, CPU registers, cache, mass storage, network controllers, etc.... The software monitor periodically checks the status of the test system to determine whether termination conditions have been met. When termination conditions such as CPU crash, application completion, or timeout occur, the hardware monitor is stopped and the target program is killed. The supervisor then obtains measurement reports from both the software and hardware monitors, merges the data, and appends a summary of the observation to any previous observations within the experiment. The observation loop is complete at this point.

The remainder of this section examines the component systems of the hybrid fault injection environment in greater detail.

3.2 Hybrid Environment Subsystems

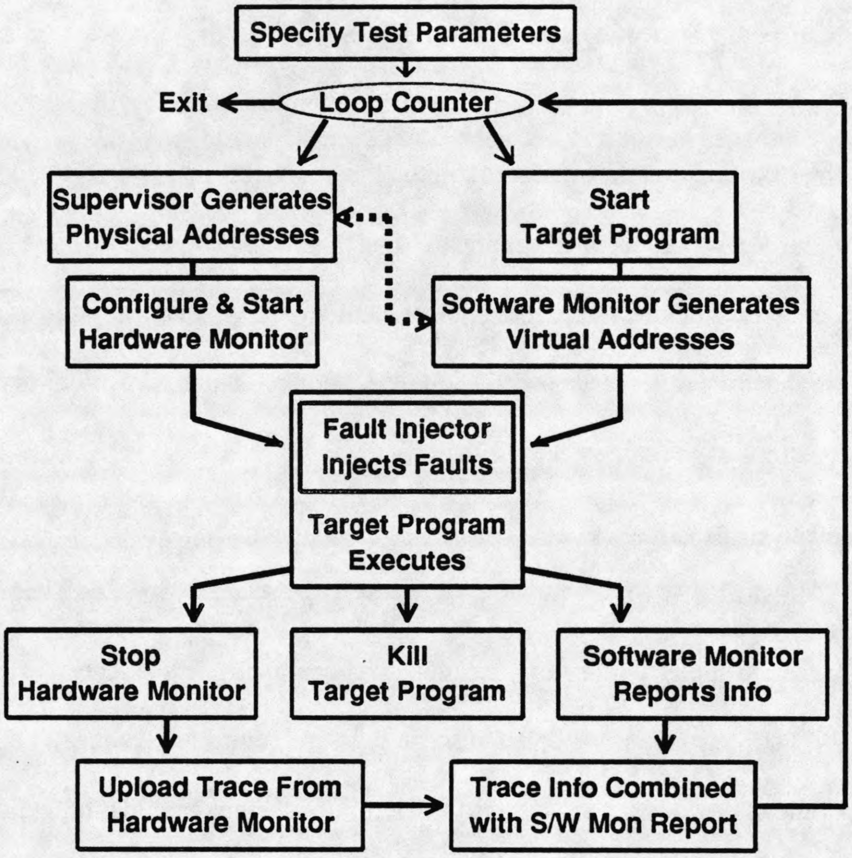

Specify Test Parameters
Exit
Loop Counter
Supervisor Generates Physical Addresses
Start Target Program
Configure & Start Hardware Monitor
Software Monitor Generates Virtual Addresses
Fault Injector Injects Faults
Target Program Executes
Stop Hardware Monitor
Kill Target Program
Software Monitor Reports Info
Upload Trace From Hardware Monitor
Trace Info Combined with S/W Mon Report

Figure 3: Fault Injection Control Flow

The **supervisor** is written in C and runs on the control host, which, in our case, is a DEC MicroVAX II. The supervisor plays roles of communication, synchronization, and data analysis in the hybrid environment. In its current implementation, the supervisor communicates with the software monitor via NFS sockets and with the hardware monitor over an RS-232 connection that uses a software-reconfigured TTY port. At the beginning of an observation loop, it communicates physical addresses to both the software and hardware monitors. At the end of an observation loop, it receives virtual addresses back from the software monitor and acquisition data back from the hardware monitor. For synchronization purposes, the supervisor controls when to start and stop both the target program and the hardware monitor. The uploading of measurements from the hardware and software monitors is also controlled by the supervisor. The analysis role of the supervisor involves taking hardware monitor measurements in the form of a timestamped trace and parsing it according to measurements taken by the software monitor. Further details of the analysis are given in section 3.3.

The **hybrid monitor** consists of both a hardware monitor and a software monitor. The hardware monitor can record and timestamp any activity, addresses, or data present on the address/data backplane of the system under test. In our hybrid environment, the hardware monitor is a Tektronix DAS 9200 — a programmable, digital analysis tool. Transfers of acquisition data and instrument setup data between the hardware monitor and supervisor are supported through a Programmatic Command Language (PCL) [Tek88]. The PCL commands allow the supervisor to reprogram the DAS, start and stop acquisitions, and upload acquisition files. Through its 92A90 data acquisition module, the DAS 9200 can perform general-purpose state analysis for up to 90 channels. The 20 MHz buffer probe accompanying the 92A90 is retargetable and can store up to 32,768 samples, where each sample is time-stamped with a resolution of 20 ns.

The software monitor is also written in C and designed to function within a Unix operating system. It assists the supervisor by determining which virtual addresses can be used for fault injection during an experiment. Virtual-to-physical address translations are performed by accessing the system page table. If no match can be found, the software monitor notifies the supervisor. Otherwise, it provides the fault injector with the matching physical addresses and a generated bit vector that specifies where, within the word, to inject a fault. After a fault has been injected, the software monitor performs periodic, unobtrusive checks of the system status to determine whether termination conditions have been reached. At the end of an observation cycle, the software monitor reports information such as virtual page frame numbers and cause of termination to the supervisor.

The **fault injector** is a psuedo-device driver written in C and was partially implemented through the addition of a small, special-purpose kernel routine. During a fault injection, the content of the physical address is read and then written back to a dummy register with a fixed, physical address. The original value is then XORed with the bit vector provided by the software monitor, and the new value is written back to the original address. By this scheme, every fault injection is immediately preceded by a write to a fixed, physical address. Thus, by programming the hardware monitor to detect and record all write activity to this address, we can obtain a record of the precise moment of fault injection.

3.3 Analysis

One of the roles of the supervisor is to analyze and merge the data it obtains from the hardware and software monitors. Figure 4 shows the information reported by both the software and hardware monitors. For each fault injected, the software monitor reports a virtual address, an XOR bit vector, and

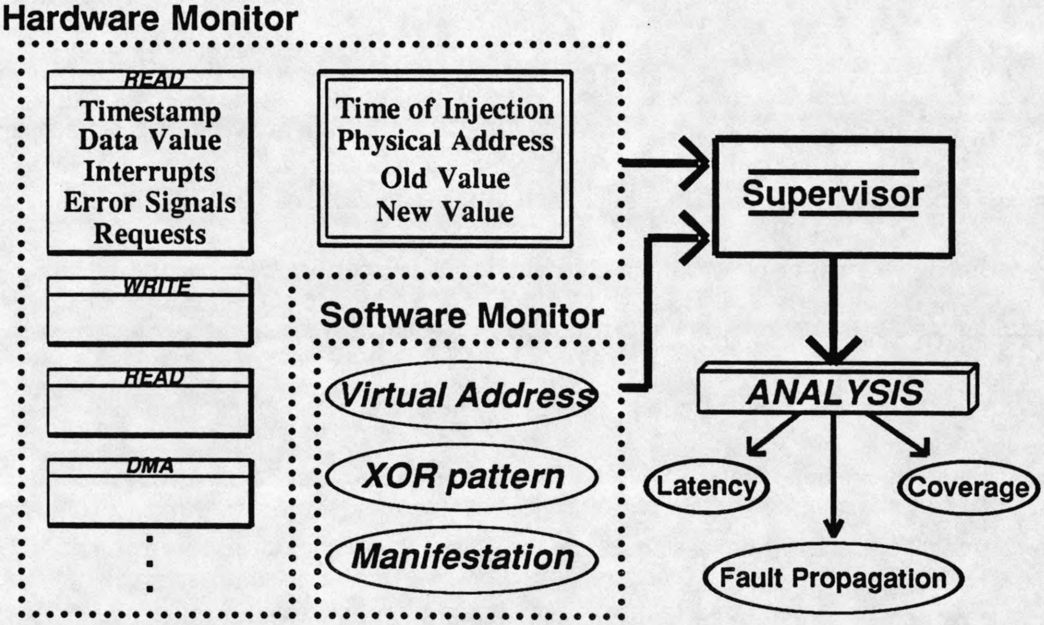


Figure 4: Analysis of Hardware and Software Monitor Reports

the effect of the fault at termination time. For that same fault, the hardware monitor reports the time of fault injection, a physical address, and the contents of that address before and after that fault. This information is followed by a timestamped list of all reads, writes, and DMA accesses to the physical address (complete with data values and signals such as interrupts, bus errors, and interrupt requests).

There are two primary functions that the supervisor performs when merging the separate hardware and software monitor reports. One function is to prune out impertinent information, and the other function is to perform format conversions. The hardware and software monitors use different formats in uploading measurements to the supervisor. The supervisor must translate and merge the information it receives so that hardware-level activity and timing can be connected to software specific information. The result of the merge is a trace file that can be analyzed at many levels.

Analysis of a trace file can yield a number of dependability statistics. Latency measurements can be derived from the difference between two timestamps, where the first timestamp corresponds to the moment of fault injection and the second timestamp corresponds to the moment of fault detection (as indicated by the appropriate read or interrupt signal). The fault detection ratio can be gathered from whether or not the software monitor reports that a failure occurred. Other statistics such as *instruction fault sensitivity* (discussed in section 5.2) can be derived by observation of the virtual address.[1] By examining the application task image, it is possible to determine which instruction is being faulted, and what immediate error propagation effect that fault will have. An example of a typical trace file and its interpretation is provided in section 4.2.

3.4 Target Programs Tested

[1] In a Unix system, the contents of virtual memory are already categorized: Kernel code begins at virtual address 0x80000000, user application code begins at 0x400000 (text), user data begins at 0x10000000 (data), user stack space begins at 0x7FFFF000 (stack) and works down.

For all the experiments described in this paper, two applications were tested under this fault injection environment. They are PRIME and ANAGRAM, and are 3,926 and 7,302 instructions long, respectively. Both were tested under heavy and light multiuser workloads and selected because they were both able to take over half an hour to complete, depending on the workload. PRIME is a CPU and memory intensive program that generates the first half million prime integers. ANAGRAM is a program that finds all three-word anagrams of a string of letters. Since it must access a local dictionary, it must perform extensive I/O in the early stages of execution.

IV. THE TANDEM INTEGRITY S2 SYSTEM

4.1 S2 Architecture

The Tandem *Integrity S2* is a fault-tolerant computing platform for Unix-based applications. Figure 5 depicts the architecture of the S2 system used in these experiments. The principle feature of the S2 is its RISC-based, TMR processing core, making it a highly-available, fault-tolerant system. Each of the three CPU boards contains a traditional CPU with cache and 8 MB of local memory. The three CPUs act as one CPU, performing identical operations and accessing the duplexed Triple Modular Redundant Controllers (TMRCs) through the *Reliable System Bus* (RSB). Each TMRC provides a 32 MB global memory and dual-rail voters. The Reliable I/O Bus (RIOB) interconnects the TMRCs with the I/O Packetizers (IOPs), which handle all system I/O (including mirrored disks, tape drives, ethernet controllers, terminals, etc...). Further information on the S2 architecture may be found in [Jewe91] and [Cutt90].

The voters are the key error detection mechanisms present on the S2. Whenever a CPU attempts to access the duplexed global memory and I/O systems, it must issue requests through the voters and, if necessary, wait for voting to complete and a result to be returned. An error is detected if two CPUs fail

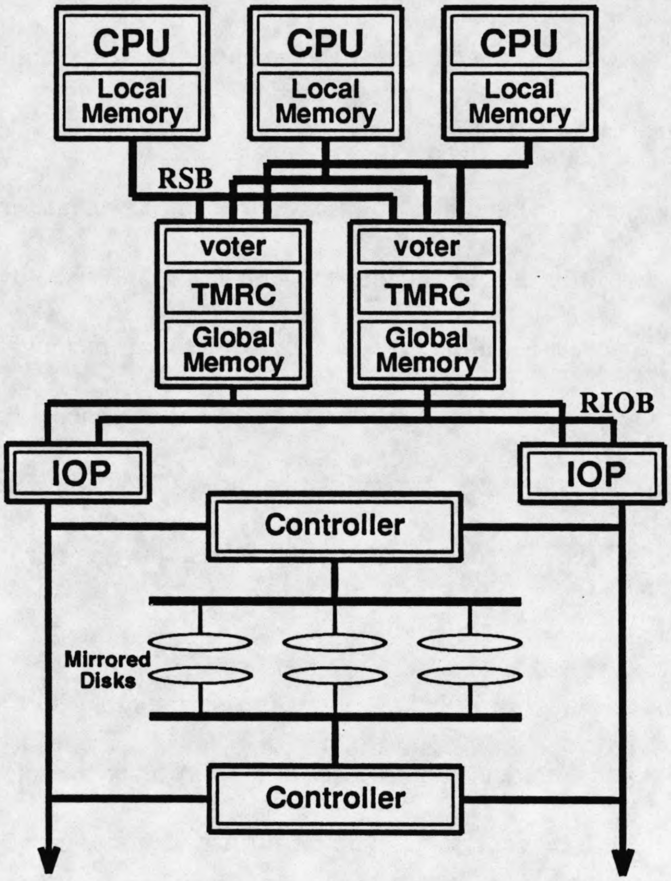


Figure 5. Architecture of the TANDEM *Integrity S2*.

to issue the same request within a fixed time period. Such requests are issued in the form of address, data, control, and interrupt values. For instance, if only two CPUs generate an interrupt, the voters will time out and signal that an error has occurred.

Several facets of the S2 architecture underscore the need to study **fault detection ratios** and **error propagation**. The S2's voters are able to detect and correct all single-bit errors and can detect and correct many multiple-bit errors. The degree to which they are called upon to correct errors can be found through studies of error propagation and fault detection. The nature of local memories also provides incentive for studying detection and error propagation. In local memories, parity checking is not performed, and errors are free to propagate to the CPU and cache without detection. Once inside the CPU, errors can propagate to registers and other local memory locations until the CPU is forced to access the TMRCs.

These system features just discussed also point out the need to examine **error latency** on the S2. Such examinations would reveal the degree to which the amount of time spent by a voter in detecting an error affects overall error latency. Measuring error latency within local memory would tell us how long errors are free to propagate within a single CPU board before being detected. Forced stalls during CPU synchronization causes error detection to be delayed on the faster CPUs while contributing to error propagation within the slowest CPUs. The degree to which this is so may be quantified through studies of error latency on the S2.

Knowing the time associated with isolating a fault is also important to the study of the S2. An intricate chain of interrupts and tests are performed on the S2 in order to isolate faults [Cutt90]. If a fault stems from a voter, the effect is the same as if a fault had stemmed from a CPU. To isolate the source of an error, an exception handler running on all CPUs uses a collection of registers on the CPU and TMRC to determine whether the voter or one of the CPUs contains the fault. It would be useful to know how long fault isolation takes, because additional errors could appear during this chain of tests and prevent successful isolation.

The issues of detection ratio, latency, error propagation, and fault isolation times are addressed by this study. To better illustrate fault injection and data collection, the following example demonstrates an application of the hybrid fault injection environment to the S2 system.

4.2 Example: Fault Injection and Data Collection

```
Phys Addr = 00xxx994

          time        location      data    flags   comments (added)
    ------------------------------------------------
             0 us | (00705994) => 26240001 F 3      before
         17.46 us | (00705994) <= 26244001 F 4      after
                  |
        108.62 us | (007E6994) => 8FBF002C F 3      before
        114.84 us | (007E6994) <= 8FBF0024 F 4      after
                  |
        189.84 us | (00567994) => 0043C821 F 3 ← addu t9,v0,v1 (before)
        196.40 us | (00567994) <= 0043C801 F 4 ← reserved instruction
                  |
        269.54 us | (004C8994) => 1000001A F 3      before
        276.24 us | (004C8994) <= 1000003A F 4      after
                  |
        350.14 us | (0064B994) => 00000000 F 3      before
        356.34 us | (0064B994) <= 00400000 F 4      after
                  |
     56.06224 ms | (00567994) => 0043C801 F 3 ← error activated
     56.38838 ms | (007E07C0)    8017B3D0 B 1 ← error detected
     57.99304 ms | (1FD10030)    B844FF0E 8 1
    103.31650 ms | (1FC00000) => 0BF00082 F 3 ← fault isolated
```

Figure 6. Example of Hardware Monitor Trace

This example is taken from an experiment in which we measured error detection latency and fault isolation times associated with single bit faults in the instruction-stream of a given application. In this example, five single-bit faults were injected into the local memory of a single CPU. The five faults were selected randomly from physical addresses that mapped back to those virtual addresses allocated to the code section of a test application. The test application was ANAGRAM, a program that generates anagrams from a string of letters. The application was running under a moderate workload within a multiuser environment. After the faults were injected, the cache was selectively flushed to ensure that faults would propagate to the CPU and cache upon a cache miss.

Figure 6 illustrates a partial trace file from a single observation generated by the supervisor. The interpretation of this example is as follows: Information recorded includes a timestamp, a memory address, direction of data transfer, data, interrupt flags, and state analysis values. Reads are denoted by

"=>", and writes are denoted by "<=" in this expansion.

The third fault corresponds to a flip of the 5th bit in the instruction *addu t9,v0,v1*, which turned it into a reserved instruction. An attempt to execute the instruction 56.06 msecs later forced a cache miss and caused the error to migrate to the cache. Upon an attempt to execute this illegal instruction, the CPU generated a reserved instruction exception. This is a low-level exception and does not trigger voting. Upon discovery that no instruction interpretation had been implemented, the kernel generated an illegal instruction fault signal which presented a high-level exception to the voters. The generation of this last exception indicates the moment of error detection at the system level. The error detection latency in this example was 326.14 µsecs. Because the other two CPUs were not faulted, they did not report similar activity. After a brief signal timeout, the TMRC performed tests and determined that CPU B (the one that was faulted) needed to be taken offline. The fault isolation time in this example was 46.92812 msecs (± 20 ns).

V. EXPERIMENTAL RESULTS

5.1 Fault Chronology

Figure 7 depicts the life of a fault in the local memory subsystem of the S2. In this example, an error occurs when the CPU obtains incorrect data from the memory. The error is detected at the system level when the voters detect a request mismatch. The *fault latency* is the time that elapses between the development of this fault and its access by the CPU as data. The *error detection latency* is the time between CPU access of this data and system-wide acknowledgment of a problem (via the voters). Within a fault tolerant system such as the S2, there is a delay associated with locating the source of the error. This delay is the *fault isolation time*. If the fault is transient, it will be corrected by performing a "power-on self test" (POST) and reintegrating the subsystem. The time required to perform POST and reintegration is the *fault correction time*.

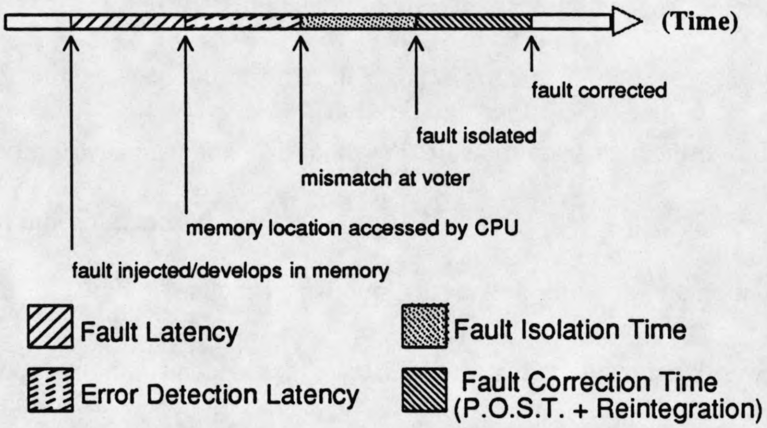

(Time)
fault corrected
fault isolated
mismatch at voter
memory location accessed by CPU
fault injected/develops in memory
Fault Latency
Fault Isolation Time
Error Detection Latency
Fault Correction Time (P.O.S.T. + Reintegration)

Figure 7: Fault Timeline

In the following sections, we present the results of performing several different forms of fault injection. The S2 is used as a case study and the results here may not be representative of Tandem systems as a whole. We begin in section 5.2 by examining issues of fault sensitivity, fault detection percentage, and the error propagation effects due to single bit faults on instructions. After that, we examine latency issues at several levels: In section 5.3, we characterize **fault** latency in the local memory subsystem of the S2. Then we move on to detection at the system level by studying **error detection** latency by fault propagation type in section 5.4. Finally, in section 5.5, we characterize the time taken by the system to perform **fault isolation.**

5.2 Instruction Level Fault Detection

The purpose of this experiment is to quantify the relative extent to which instructions are sensitive to faults. Faults were injected randomly within the code sections (local memory and cache) of executing programs. If a fault resulted in the failure of a CPU, then the identity of the faulted instruction was obtained. From the measurements, the probability that a CPU failure was caused by a fault in a specific instruction was estimated. By comparing the measurements with the probabilities calculated from instruction execution frequencies, the relative *fault sensitivity* of different instructions was evaluated.

Measured values were obtained from 2,400 fault injections, 95 percent of which caused CPU failure. The identity of each faulted instruction was obtained from the disassembler by matching listed virtual addresses with those obtained from the software monitor,[2] The estimated probability that a CPU failure was caused by a fault in a specific instruction was obtained by counting the number of failures attributed to each instruction and dividing by the total number of faults.

Expected values were computed from opcode distributions and instruction execution frequencies. The opcode distributions were found by counting the number of times an opcode appeared within the static image of the program tested. By dividing this count by the total instruction count of the program,

Table I. Expected and Measured Detection Probabilities

Instr	Expected	Measured
addiu	35.5615 %	16.0342 %
sll	18.8399 %	7.3529 %
lw	16.8997 %	18.1214 %
sw	13.8242 %	13.3776 %
bne	4.2130 %	5.6452 %
addu	3.2937 %	9.9146 %
beq	3.0868 %	8.4440 %
lui	1.5014 %	3.5579 %
lbu	1.0479 %	5.3605 %
jal	0.6532 %	4.4118 %
jr	0.4229 %	3.0361 %
andi	0.2689 %	1.6603 %
subu	0.1919 %	1.0911 %
ori	0.1263 %	0.2846 %
bgez	0.0433 %	0.5693 %
sltu	0.0210 %	0.5693 %
slt	0.0045 %	0.5693 %

[2] Instructions having identical opcodes were merged. NOP, LI, B, and MOVE are actually special cases of SLL, ADDIU, BEQ, and ADDU, respectively.

the probability that a specific instruction will be faulted was estimated. Instruction execution frequencies were obtained by counting the appearances of an opcode within an instruction trace generated bt executing a profiled program. These frequencies reflect the conditional probability that a fault in a specific instruction will be detected, given that the instruction has been faulted.[3] Multiplication of this conditional probability by the probability that a specific instruction will be faulted produces the joint probability that a fault will appear in a specific instruction and be detected.

The expected and measured values for the probability that a fault will occur in an instruction and be detected are listed in Table I for several opcodes. The values were normalized to compensate for all the opcodes not shown. Instructions were listed by decreasing order of expected values, and it can be seen that measured values generally decrease as well, but not in every case. This table permits us to evaluate the relative fault sensitivity of any two instructions.

Comparing expected detection probabilities of the load word and store word instructions, we see that LW is expected to be faulted and cause a CPU failure 1.2225 times as often as SW. From the measurements, we see that this happened 1.3546 times as often, which indicates that SW is slightly better able to perform its function in the presence of faults than LW is. By taking the ratio of the measured and expected relative frequencies, (1.3546/1.2225), one might say that LW is 10.8 percent more sensitive to faults than SW is. For ADDIU and LW, we see that ADDIU is expected to be faulted and cause a CPU failure 2.1043 times as often as LW. From the measurements, we see that this happened only 0.8848 times as often, which indicates that LW is 137.8 percent more sensitive to faults than ADDIU is.

These results are of importance in several areas. The findings are relevant to compiler designers wishing to take into account dependability issues because, as this experiment has shown, instructions

[3] This reflection is not necessarily linear, since looping can skew results over short periods of time. In general, however, a fault in a frequently executed instruction is more likely to be discovered than one in a seldomly executed instruction.

differ substantially in the degree to which they are fault sensitive.. Since the function of an instruction may be partially responsible for its fault sensitivity, these findings reveal a need to examine fault detection in terms of the error propagation effects caused by faulted instructions.

In examining popular, fixed-width instruction sets such as those of the Mips RISC set, we found that a single bit fault will immediately impact a given instruction in one of several fashions — we also found that the impact could be determined automatically by a disassembler-like program. Table II lists the eleven manifestations into which all instructions and faulted bits were mapped for this study. The NIL case is that in which a flipped bit is incapable of altering normal operation. An example is discussed below. Cases USV and ILI trigger error detection mechanisms on a CPU, which in turn alert the system that an unacceptable instruction has been encountered. In both cases, no further corruption occurs. Execution of the wrong instruction (WIE) can be caused by an improperly specified opcode. A branch to a wrong address (BWA) can be caused either by an improperly specified address or base register, or by misdirected calculations. Both WIE and BWA cases are capable of causing multiple corruptions. Writes to wrong registers (WWR) and addresses (WWA) can cause double corruptions:

Table II. Error Propagation Types

Id	Description	Multiplier
NIL	Fault has no impact whatsoever on execution	0
USV	(Unassigned Space Violation) Address Exception	0
ILI	ILlegal Instruction	0
WIE	Wrong Instruction Executed	many
BWA	Branch to Wrong Address	many
WWR	Write to Wrong Register	2
WWA	Write to Wrong Address	2
RWR	Read from Wrong Register (reg. corrupted)	1
RWA	Read from Wrong Address (reg. corrupted)	1+
BID	Bad Immediate Data (reg. corrupted)	1
BVM	Wrong register (Bad Value) transferred to Memory	1

the intended location fails to be updated with new data, and a misintended location is corrupted. The process of reading a wrong register or address does not, in itself, cause error propagation. But when such actions cause wrong values to be passed on to operations that update registers (RWR, RWA and BID) or memory (BVM), error propagation results. The case of RWA is a potential exception — if the read of a wrong address causes a page fault at the wrong time, then flow of control is altered, causing more than just data corruption to occur.

Reusing the tracefile data obtained in studying fault sensitivity, we computed first-order error propagation effects from the instruction type and XOR bit vector associated with each injected fault. The fault detections were then tabulated by effect and analyzed. Table III shows the results of this analysis. Roughly five percent of the instruction single-bit errors had no observable impact on program execution. Two of the forms trigger error detection without error propagation, but they are caused by only 9.13% of the instruction single-bit errors. This means that over 90 percent of all detectable single-bit errors in

Table III. Effects of Instruction Single Bit Errors.

Effect of error on instruction execution:	Portion
wrong instruction executed (multiple effects)	21.18%
branch to wrong address (multiple effects)	14.98%
read from wrong address (register corrupted)	13.10%
bad immediate data (register corrupted)	9.83%
write to wrong register (double fault)	9.83%
write to wrong address (double fault)	8.12%
read from wrong register (register corrupted)	6.90%
illegal instruction (triggers detection)	6.16%
no observable impact on program execution	4.97%
(USV) address exception (triggers detection)	2.97%
value of wrong register transferred to memory	1.97%

instructions cause propagating errors. Comparing ILI with WIE, we observe that a single alteration in an instruction's opcode bit is 3.44 times as likely to produce another legal (but wrong) instruction than it is to produce an illegal instruction. We also observe that the two most common error forms (*wrong instruction execution* and *branch to wrong address*) occur 36.2 percent of the time and are also the two with the highest propagation fan-out. This finding means that the expected propagation fan-out can be quite high.

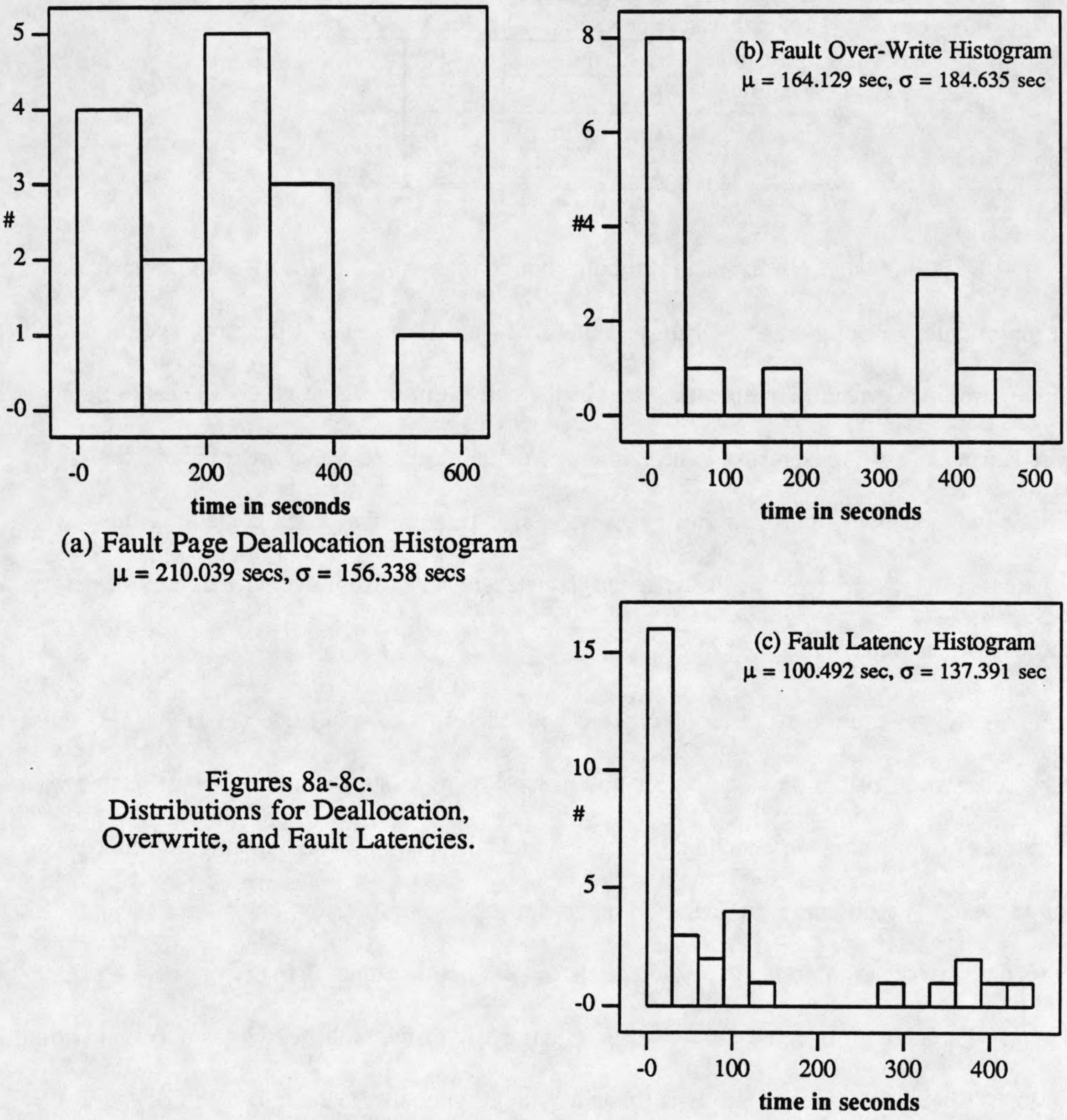
A comparison of fault detection breakdowns by propagation effect between two applications (Table IV) shows that several propagation effects are sensitive to application. An instruction single bit error was more likely to produce wrong instruction execution (WIE) and bad immediate data (BID) propagations within the PRIME application than within ANAGRAM. Also, WWR, RWR, and RWA errors were more likely to result within the ANAGRAM application than in PRIME. For the rest of the error types, the odds were about even.

5.3 Fault Latency

In this experiment, fault locations were selected from all sections of memory allocated to the ANAGRAM test program (data, stack, and text). The ANAGRAM program was executed under a moderately heavy workload, and a total of 750 fault injection observations were performed. Trial runs indicated that the general distribution forms did not change by extending observation times, so we

Table IV. Selected Error Manifestation Probabilities

Description	ANAGRAM	PRIME
wrong instruction executed	18.56%	24.36%
write to wrong register	11.13%	8.31%
read from wrong register	8.23%	5.35%
read from wrong address	14.52%	11.46%
bad immediate data	8.47%	11.46%



(a) Fault Page Deallocation Histogram
μ = 210.039 secs, σ = 156.338 secs

(b) Fault Over-Write Histogram
μ = 164.129 sec, σ = 184.635 sec

(c) Fault Latency Histogram
μ = 100.492 sec, σ = 137.391 sec

Figures 8a-8c.
Distributions for Deallocation,
Overwrite, and Fault Latencies.

restricted observations to ten minutes each. The following results were obtained:

(fate of fault)		count	(%)
Fault never accessed:		687	91.6%
Activated (4.3%)	CPU failure:	27	3.6%
	no effect:	5	0.7%
Undetected (4.1%)	Overwritten:	16	2.1%
	Deallocated:	15	2.0%

These results tell us what can happen to a fault, though many (91.6 percent) were never accessed in the first ten minutes of observation. Of those faults that were accessed (8.4 percent), about one half became active errors via a read (51 percent). Most of the active errors caused CPU failure (84 percent) and the rest had no effect (16 percent). The remainder of the accessed faults were undetected, either because the application overwrote them with new values or because the pages containing them were deallocated first. Fifty-two percent of the accessed, undetected faults were overwritten and 48 percent were within pages that were deallocated.

Figures 8a and 8b show the latency associated with the removal of undetected faults, i.e., those faults that were removed before they could become errors. Figure 8a shows the latency associated with the deallocation of the clean page containing a fault, and figure 8b shows the latency associated with overwriting a fault. We observe that faults within clean pages were deallocated after an average of about 210 seconds, which is more than the average time spent before a fault is overwritten by the application (about 164 seconds). In figure 8b, we observe that many of the fault overwrites occurred within the first minute. These observations tell us is that early fault removals are due mostly to overwrites by the application and that later removals are due mostly to page deallocation. This information characterizes a natural fault removal processes and could assist in the design of efficient scrubbing techniques.

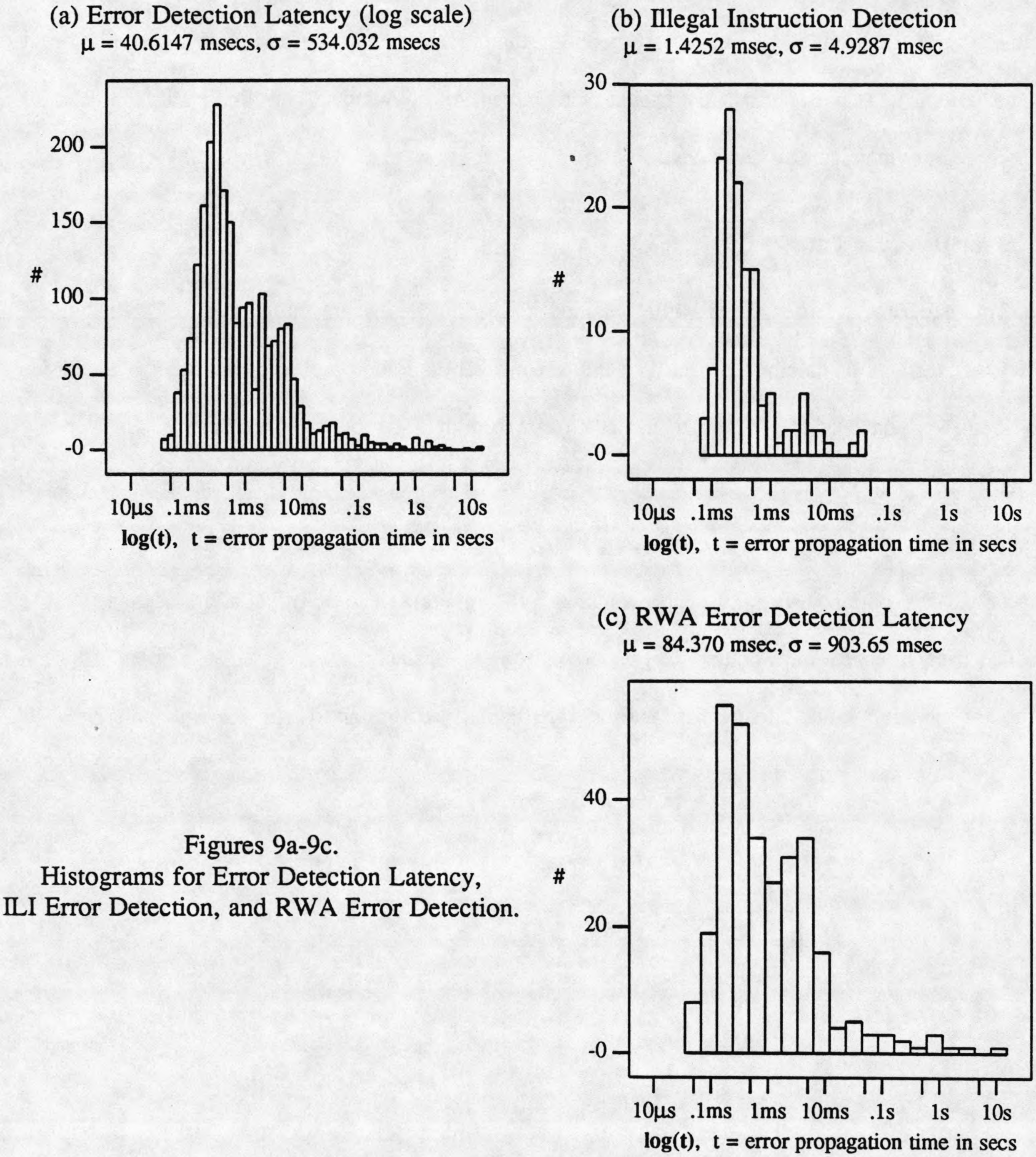
In figure 8c, we again see a sharp decay and long-tailed distribution for detected faults. In this example, faults were detected after an average of about 100 seconds, but half were detected in the first 35 seconds. This distribution is very similar to the distribution of latency associated with fault overwrites (figure 8b), and is characteristic of distributions found in past latency studies [Youn91]. The similarity in distributions implies that the probability that a local memory location will be read before it will be written does not significantly change with increasing dormancy. This finding can be used to support memory management decisions.

5.4 Error Detection Latency

This experiment reuses the tracefile obtained in section 5.2, where faults were injected into the code (text) sections of memory allocated to the test programs. Each program was tested to determine how long it would take the system to detect an error through its voters. The cache was selectively flushed after a fault injection to ensure that faults would propagate to the CPU and cache upon a cache miss. The error detection latency was measured for each of 2,176 faults and the identity of each faulted instruction was obtained from the disassembler by comparing listed virtual addresses with those obtained from the software monitor. Corresponding error propagation type was then computed by the method of section 5.2, and latency distributions were tabulated according to error propagation type.

Table V. Error Detection Latency by Type

Type	Median	Mean	Std Dev
USV	252.2 μs	464.0 μs	734.0 μs
ILI	280.1 μs	1.425 ms	4.929 ms
WIE	381.0 μs	30.08 ms	380.8 ms
BWA	543.5 μs	21.35 ms	162.1 ms
WWR	654.9 μs	85.82 ms	1.069 s
WWA	609.0 μs	40.67 ms	281.2 ms
RWR	481.8 μs	47.55 ms	344.8 ms
RWA	596.8 μs	84.37 ms	903.6 ms
BID	690.6 μs	20.43 ms	127.4 ms
BVM	1.179 ms	43.06 ms	231.3 ms
all	492.9 μs	40.61 ms	534.0 ms



(a) Error Detection Latency (log scale)
μ = 40.6147 msecs, σ = 534.032 msecs

(b) Illegal Instruction Detection
μ = 1.4252 msec, σ = 4.9287 msec

(c) RWA Error Detection Latency
μ = 84.370 msec, σ = 903.65 msec

Figures 9a-9c.
Histograms for Error Detection Latency,
ILI Error Detection, and RWA Error Detection.

Figure 9a illustrates the overall error detection latency distribution. Latencies range from under 50µs to over 10 seconds. The mean error detection latency was 40.61 msecs, but several distinct peaks can be seen in the distribution and may be due to distinct error detection mechanisms.

Table V lists error detection latency means and medians for each of the error types. The first two, USV and ILI cause immediate detection at the CPU level and short error detection latencies at the system level. These system-level latencies are considerably longer than the CPU-level error detection latencies found in past simulation-based studies[Czec91], but part of the system-level latency comes from the time taken during voting. As will be seen later, these latencies are usually negligible when compared to fault isolation times. In examining each of the error types, we observe that while the means are on the order of tens of milliseconds, most errors are detected in under 493 μs. This corresponds to about 5,900 instruction cycles and includes at least one round of exception handling.

Figures 9b and 9c illustrate distributions of error detection latency for error types ILI and RWA, respectively. For illegal instructions, the number of cycles required to cause the CPU to issue an interrupt is small and fairly constant. Therefore, what we see in figure 9b is primarily the latency imposed by the voters in detecting the error of a lone CPU-generated interrupt. As expected, the main peak seen in an Illegal Instruction error is narrower than what is seen in the distributions for errors that propagate.

By contrast, the error detection latency distribution for errors that corrupt registers by reading from a wrong memory location (RWA) is quite wide. In figure 9c, we observe that the error detection latency introduced by the voter plus the latency due to error propagation can be as large as 10 seconds. This finding shows that propagating errors can remain undetected for millions of instructions and highlights the need to better characterize error propagation.

5.5 Fault Isolation

As defined in section 5.1, *fault isolation time* is the delay associated with locating the source of an error. Such a source may be a faulty CPU board, memory board, voter, bus, etc... In this experiment, faults were injected only into the local memory of CPU B. Consequently, fault isolation identified CPU B as the component to be shutdown every time. As can be seen in Figure 10, the amount of time

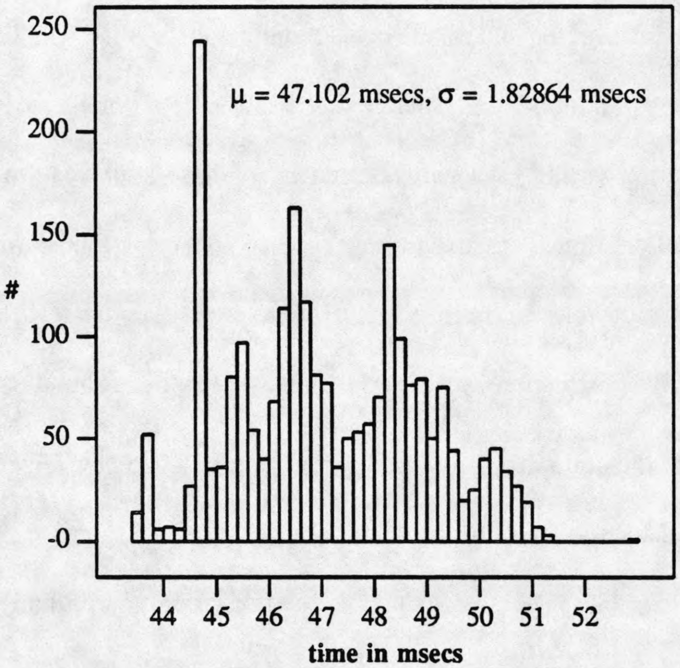


Figure 10. Histogram of Fault Isolation Times

required to achieve fault isolation was roughly constant (47 msecs), with a standard deviation of only 3.88%. Times were fairly independent of workload level and the application tested. The fault isolation time corresponds to approximately 560,000 instructions.

Bearing in mind from the previous section that undetected error propagation times can well exceed 47 msecs (3.45 percent of the error detection times exceeded 47 msecs), the potential for any propagating errors to affect the fault isolation process is of much concern. This finding makes evident the need to investigate the effects of multiple errors.

V. CONCLUSIONS

This paper describes a hybrid fault injection environment, wherein faults are injected via software and the impact is measured by both software and hardware. The environment is useful for evaluating system dependability, and it has the advantage in that it introduces minimal perturbation, and provides a high degree of control over the location of faults to be injected. The injection system is not limited to just user application space. It can be used to inject faults in the Kernel, in CPU registers, cache, local memory, mass storage, network controllers, and any other subsystem that is mapped into physical address space. The fault injection environment was applied to the fault tolerant, Unix-based Tandem Integrity S2.

Using our hybrid monitor and fault injection environment, we obtained several key results: The design decision to preserve error correction coding in the cache but not in local memory was supported by comparing fault detection ratios of the cache and local memory subsystems. A natural fault removal processes that can assist in the design of efficient scrubbing techniques was characterized. We also characterized immediate error propagation effects caused by injecting faults into the instruction stream of a Mips RISC processor. The impact of faults on instruction code during runtime is significant — we found that single-bit errors propagate additional errors 85.9 percent of the time and that individual instructions differ substantially in the degree to which they are fault sensitive. It was also found that error propagation times can well exceed the fault isolation time.

Empirical measurements of all the dependability statistics discussed can further serve as input data for any simulation-based study of the long-term effects of faults on this system.

VI. ACKNOWLEDGMENTS

Special thanks go to Carlos Alonso (Tandem Computers) for writing the customized kernel section that enabled us to perform fault injections on the S2. Without his expertise and familiarity with the S2, our experiments would not have been possible. The authors also wish to thank Howard Alt (Sun Microsystems) for his help in the virtual to physical address conversion routines, Rob Reinauer (Tandem) for supplying workloads, and Doug Jewett (Tandem) for supplying profile-parsing tools and many useful suggestions.

This research was supported in part by Tandem Computers, Inc. and in part by the Department of the Navy, Office of the Chief of Naval Research under Grant N00014-91-J-1116. The content of this paper does not necessarily reflect the position or policy of the government and no official endorsement should be inferred.

REFERENCES

[Chil87]

R. Chillarege and R. K. Iyer, "Measurement-Based Analysis of Error Latency," *IEEE Trans. Computers*, vol. C-36, no. 5., May 1987, pp. 529-537.

[Chil89]

R. Chillarege and N.S. Bowen, "Understanding large system failures — A fault injection experiment," *19th International Symposium on Fault-Tolerant Computing*, June 1989, pp. 355-363.

[Choi89]

G.S. Choi, R.K. Iyer and V. Careno, ''FOCUS: An Experimental Environment for Validation of Fault Tolerant Systems: A case study of a Jet Engine Controller'', *IEEE International Conference on Computer Design*, Cambridge, MA, October, 1989, pp. 561-564.

[Cour81]

B. Courtois, "A Methodology for On-line Testing of Microprocessors," *Proc. 11th International Symposium on Fault-Tolerant Computing*, Portland, Maine, 1981, pp. 272-274.

[Cutt90]

R.W. Cutts, N.A. Mehta, and D.E. Jewett, ''Multiple Processor System Having Shared Memory With Private-Write Capability'', United States Patent No. 4,965,717, Oct. 23, 1990.

[Czec91]

E. Czeck, ''On the Prediction of Fault Behavior based on Workload'', Ph.D. dissertation, Electrical and Computer Engineering Department, Carnegie Mellon University, Pittsburgh, PA, April 19, 1991.

[Gosw91]

K. Goswami and R. Iyer, ''A Simulation-Based Study of a Triple Modular Redundant System using DEPEND'', 5th International FTRS conference, Nurnberg, Germany, Sept. 25-27th 1991.

[Jewe91]

D. Jewett, ''Integrity S2: A Fault-Tolerant Unix Platform'', 21st International Symposium on Fault-Tolerant Computing, Montreal, June 25-27, 1991, pp. 512-519.

[Kana92]

G. Kanawati, N. Kanawati, and J. Abraham, ''FERRARI: A Fault and ERRor Automatic Real-time Injector'', CERC Technical Report UT-CERC-TR-JAA92-01, Computing Engineering Research Center, University of Texas, Austin, TX, 1992.

[Lala83]

J. H. Lala, "Fault detection, isolation and reconfiguration in FTMP: Methods and experimental results," *Proc. 5th Avionics Systems Conference*, Seattle, WA, Nov. 1983, pp. 21.3.1-21.3.9.

[Lome86]
D. Lomelino and R. Iyer, ''Error propagation in a digital avionic processor: A simulation-based study'', *NASA CR-176501*, University of Illinois, 1986.

[McGo83]
J. G. McGough, F. L. Swern and S. Bavuso, "New results in fault latency modeling," *Proc. IEEE EASCON Conf.*, Washington, D.C., Aug. 1983, pp. 882-889.

[Mitr88]
S.G. Mitra and R.K. Iyer, "Measurement-based Analysis of Multiple Latent Errors and Near-coincident Fault Discovery in a Shared Memory multiprocessor," *Proceedings, 1988 International Conference on Parallel Processing*, St. Charles, IL, August 15-19, 1988, pp. 404-409.

[NASA81]
"Measurement of fault latency in a digital avionic miniprocessor," NASA Contractor Report 3462, 1981.

[NASA83]
J. G. McGough and F. L. Swern, "Measurement of fault latency in a digital avionic miniprocessor part II," NASA Contractor Report 3651, 1983.

[Sega88]
Z. Segall, D. Vrsalovic, et al., ''FIAT — Fault Injection Based Automated Testing Environment'', 18th International Symposium on Fault-Tolerant Computing, 1988, pp. 102-107.

[Shin86]
K. G. Shin and Y. H. Lee, "Measurement and Application of Fault Latency," *IEEE Trans. Computers*, vol. C-35, no. 4., April 1986, pp. 370-375.

[Tek88]
DAS 9200 92A60/90 User's Manual (8-/16-/32-Bit Microprocessor Support Modules), Tektronix, Inc., Beaverton, OR, May 1988.

[Youn91]
L. Young and R. Iyer, ''Error Latency Measurements in Symbolic Architectures'', *AIAA Computing in Aerospace 8*, Baltimore, Maryland, October 22-24, 1991, pp. 786-794.

[Youn92]
C. Yount and D. Siewiorek, ''Automatic Generation of Instruction-Level Error Manifestations of Hardware Failures'', (pending technical report), Center for Dependable Systems, Dept. of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, 1992.