Febuary 1994

UILU-ENG-94-2205
CRHC-94-04

Center for Reliable and High-Performance Computing

A STUDY OF THE RELATIONSHIP BETWEEN THE PERFORMANCE AND DEPENDABILITY OF A FAULT-TOLERANT COMPUTER

Kumar K. Goswami

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-94-2205 CRHC-94-04

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
NASA , Tandem, and CSC

6c. ADDRESS *(City, State, and ZIP Code)*
1308 W. Main St.
Urbana, IL 61801

7b. ADDRESS *(City, State, and ZIP Code)*
Arlington VA. 22217 Falls Church VA 22220
Moffet Field, CA 95043
Cupertino, CA 95014

8a. NAME OF FUNDING / SPONSORING ORGANIZATION
7a

8b. OFFICE SYMBOL *(If applicable)*

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
N00014-91-J-1116 Tandem
NASA NAG-1-613 GSA CSC 468969

8c. ADDRESS *(City, State, and ZIP Code)*
7b

10. SOURCE OF FUNDING NUMBERS

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

11. TITLE *(Include Security Classification)*
A Study of the Relationship Between the Performance and Dependability of a Fault-Tolerant Computer

12. PERSONAL AUTHOR(S)
TSAI, Timothy K.

13a. TYPE OF REPORT
Technical

13b. TIME COVERED
FROM ________ TO ________

14. DATE OF REPORT *(Year, Month, Day)*
94-02-04

15. PAGE COUNT
40

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
dependability, fault-tolerant, performance, fault injection, latency

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

This thesis studies the relationship by creating a tool (FTAPE) that integrates a high stress workload generator with fault injection and by using the tool to evaluate system performance under error conditions. The workloads are comprised of processes which are formed from atomic components that represent CPU, memory, and I/O activity. The fault injector is software-implemented and is capable of injecting any memory-addressable location, including special registers and caches.

This tool has been used to study a Tandem Integrity S2 Computer. Workloads with varying numbers of processes and varying compositions of CPU, memory, and I/O activity are first characterized in terms of performance. Then faults are injected into these workloads. The results show that as the number of concurrent processes increases, the mean fault latency initially increases due to increased contention for the CPU. However, for even higher numbers of processes(< 3 processes), the mean latency decreases because long latency faults are paged out before they can be activated.

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

A STUDY OF THE RELATIONSHIP BETWEEN THE PERFORMANCE AND DEPENDABILITY OF A FAULT-TOLERANT COMPUTER

BY

TIMOTHY K. TSAI

B.S., Brigham Young University, 1990

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1994

Urbana, Illinois

PRECEDING PAGE BLANK NOT FILMED

ABSTRACT

Many existing tools perform validation of high dependability computers. These tools have focused mainly on creating an environment to support direct fault injection. Less effort has been directed toward investigation of the relationship between performance and dependability. This thesis studies this relationship by creating a tool (FTAPE) that integrates a high stress workload generator with fault injection and by using the tool to evaluate system performance under error conditions.

The workloads are comprised of processes which are formed from atomic components that represent CPU, memory, and I/O activity. The fault injector is software-implemented and is capable of injecting any memory-addressable location, including special registers and caches.

This tool has been used to study a Tandem Integrity S2 computer. Workloads with varying numbers of processes and varying compositions of CPU, memory, and I/O activity are first characterized in terms of performance. Then faults are injected into these workloads. The results show that as the number of concurrent processes increases, the mean fault latency initially increases due to increased contention for the CPU. However, for even higher numbers of processes (> 3 processes), the mean latency decreases because long latency faults are paged out before they can be activated.

ACKNOWLEDGEMENTS

Many people deserve recognition for helping me to finish my M.S. thesis. My professor, Ravi Iyer, has been patient in providing direction and encouragement and giving me enough time to complete my work. Luke Young and Kumar Goswami paved the way for me with their work on the fault injector and DAS. I would also like to thank Wei-lun Kao, Gwan Choi, and Steve VanderLeest for letting me try out new ideas on them. And finally, I must thank my mother for reminding me every time she called that I should work on this thesis.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

The validation of fault-tolerant computer systems is an important issue because the dependability of these computers must be ascertained before use in critical applications. Presently, a common method of validating computer prototypes is through fault injection. Many fault injection studies have been reported in the literature. The studies include radiation-based, hardware-implemented, and software-implemented approaches to inject faults. These efforts concentrate mainly on inventing a better method of performing fault injection.

However, an issue that has not received much attention is the relationship between performance and dependability, which are determined not only by the system but also by the workload. In order to effectively evaluate a fault-tolerant system, the system must be operated with workloads that stress both the functionality and modules in the system under high stress conditions. Faults must be injected under such workloads to evaluate the full potential of the various fault-tolerant mechanisms. If the part of the system that is faulty is not stressed adequately, then the propagation of the fault will appear to be less than it would be under high stress. Contrastingly, if the workload utilizes all parts of the system, then the full potential of the fault will be realized.

This thesis presents a tool that is able to create workloads that can stress the test machine in various ways. A synthetic workload generator is used to obtain high controllability and flexibility. This workload generator is integrated along with a fault injector to produce the Fault-Tolerance and Performance Evaluator (FTAPE), a tool that can study jointly both performance and dependability. The workload is able to send to the fault injector information, such as where in memory the workload is current executing. This information is used to select fault types and locations. The generated workloads consist of processes, which are created by combining atomic components that represent CPU, memory, and I/O activity. Each workload is specified by the number and types of processes, and each process is characterized by the composition and sequence of atomic components.

A software-implemented fault injector is used in this study. *Software-implemented fault injection* (SWIFI) uses software to corrupt memory and registers. SWIFI is cheaper and easier to use than hardware-implemented methods since no additional hardware is needed. Communication is also easier since the injection routines can be reprogrammed to take advantage of dynamic data, such as that sent by the workload. SWIFI can also be used to emulate the effect of hardware faults [1].

FTAPE has been implemented and used on a Tandem Integrity S2 computer. Experiments were performed using workloads which (1) varied in CPU, memory, and I/O composition and (2) varied in the number of concurrent processes. The workloads were first characterized through the use of performance measurements to demonstrate their CPU, memory, and I/O nature. Then, the same fault was repeatedly injected into these workloads.

to show that different propagation paths result from different workloads. These experiments are repeated, with the exception that the faults are injected into the entire workload, in order to study the dependability characteristics, such as error detection coverage and fault latency, in relation to the entire workload.

The remainder of this thesis is organized as follows: Chapter 2 has a summary of related work. A detailed description of FTAPE, including the test machine, can be found in Chapter 3. Chapter 4 presents the characterization of the workloads used in the experiments, and Chapter 5 describes the experiments. Conclusions and directions for future work are given in Chapter 6.

2. RELATED WORK

Much work has already been done in the area of fault-tolerant system evaluation. The methods utilized have included simulation, trace-based simulation, physical fault injection, and SWIFI. This thesis uses the SWIFI approach.

Among the simulation-based studies is Iyer [2], in which error propagation from the gate to the chip level was investigated. FOCUS [3], [4] is a tool that conducts fault sensitivity analysis of chip-level designs through the use of circuit-level and logic-level simulation. DEPEND [5], [6] creates a simulation environment to study system-level effects of faults. An instruction-level simulation is used to perform fault injection in [7]. These simulation-based approaches are very flexible, but they do not reflect the effects of system components not included in the simulation model.

Trace-based simulation differs from pure simulation by making use of data traces from actual machines. In [8], the memory of large computer systems was periodically sampled by a hardware monitor. A similar method was used to study a shared-memory multiprocessor in [9]. A hybrid monitor approach was used in [10] to investigate a TI Explorer II Lisp machine. Unfortunately, due to memory limitations, traces are limited in the amount of information they can contain.

Physical fault injection involves physically perturbing a system through some hardware means. In [11], heavy-ion radiation was used as a method of fault injection. Experiments involving FTMP [12], [13], [14] use fault-injection implants between chips and sockets to create pin-level faults. In [15], MESSALINE introduces faults by using active probes which inject current onto a pin, as well as implants between chips and sockets. These physical hardware approaches are either not easily controllable or not very flexible.

An alternative to the physical fault injection approach is the SWIFI approach. Instead of using hardware to inject faults, software is used. More control over injections is gained, but a certain overhead is incurred. FIAT is an automated environment developed at Carnegie Mellon University [7], [16]. Faults were injected into IBM RT workstations, and an analysis of runs with and without injections was performed. Another automated fault-injection approach is FERRARI, which is described in [17]. FERRARI utilizes software traps to inject transient faults, in addition to permanent faults.

The environment used in this study is a hybrid monitor-based environment [18]. A hybrid environment combines the flexibility and utility of SWIFI with the low intrusiveness of a hardware/software monitor. Past SWIFI implementations, such as FIAT and FERRARI, have relied on instrumenting user applications. The fault injection facility on the Tandem Integrity S2 requires no modification of the application. Moreover, previous studies have limited the location of faults to user space because the test machines were not designed to tolerate certain faults.

Research on workload generation has focused on three methods: trace data, benchmarks, and synthetic programs. The method selected in this thesis is the use of synthetic programs.

The main advantage of using trace data to generate workloads is an accurate representation of an actual workload. An example in which trace data of usage in a file system were used to perform experiments on file system cache sizes is found in [19]. The disadvantages of using trace data include large data files and difficulty in using the data on different machines or in modifying the data to alter the workload. These problems are not characteristic of benchmarks.

Benchmarks are relatively short programs or scripts that are designed to model real workloads. They are easily portable to other machines and can be modified by simply changing the input parameters. Some examples of benchmarks are found in [20], which measured the performance of three UNIX systems, and in [21], which compared the Sun NFS with the Andrew file system. The main criticism of benchmarks is that they do not represent real workloads.

Synthetic programs attempt to combine the advantages of trace data and benchmarks by adding parameters that can be adjusted to reflect real workloads. A discussion of synthetic workload design methods can be found in [22]. A few studies using synthetic workloads are found in [23], [24], and [25], which used synthetic workloads to model a file update process.

Much of the previous work concerning synthetic workload generation involves workload characterization. In the literature, workload characterization usually refers to the extraction of parameters describing a representative workload, which can then be used to form probabilistic models. These models can then be solved to evaluate the performance of a computer system under that specific workload. Ferrari [26] was the first to identify *resource patterns*, which were identified by visual inspection and clustering of two-dimensional scatter plots representing resource usage. The clustering process was later improved by using statistical pattern recognition techniques [27], [28].

In this thesis, workload characterization is used to measure the extent to which a workload stresses the system (i.e., to what extent the system resources are utilized by the workload). Although the result of the characterization is not used for modeling, a possible future use is in the derivation of a single measure that could be used for a more direct quantitative comparison of performance and dependability. A single measure incorporating both performance and dependability could be used as a benchmark to compare different fault-tolerant computers.

The major contribution of this work is relating performance to dependability. This is accomplished through the creation and use of a synthetic workload generator in conjunction with a fault injector. Previous studies have limited workloads to small programs which have been chosen with little knowledge of the characteristics of those workloads. With FTAPE, complex workloads, including multiple processes, can be created with a controllable composition of various components. These workloads are characterized in order to obtain performance information, and this information is then related to the results of fault injections into those workloads.

3. DESCRIPTION OF FTAPE

FTAPE consists of two main parts: the fault injector (FI) and the workload generator (WG). An illustration of the interaction between the FI and the WG is given in Figure 3.1. The FI is started up first, and then the WG is initiated by the FI. The FI and WG obtain input from parameters files, and the FI also produces a log file. To observe the propagation of injected faults, a hybrid monitor fault injection environment was used (see Section 3.4). A more detailed description of these main components is given below.

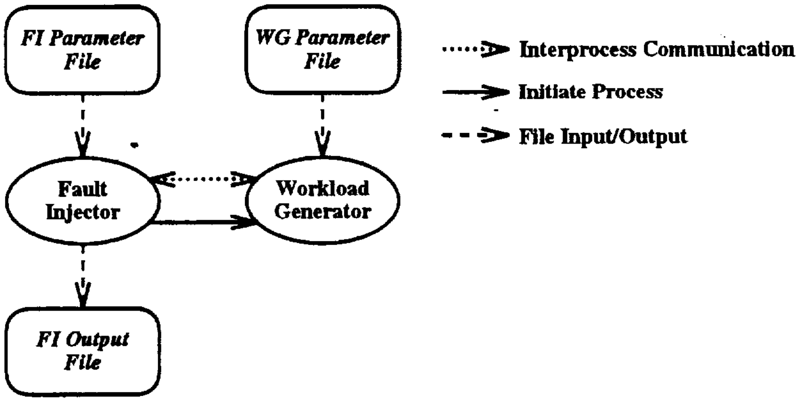


Figure 3.1: Fault Injector and Workload Generator Environment

3.1 Fault Injector

The fault injector is the subprogram that injects faults into the local memory space of the S2. When first started up, it reads in a distribution of interarrival times for fault injections. The distribution is composed of discrete times with associated probabilities. This method allows distributions that cannot be easily represented by a mathematical formula to be specified. After reading in the interarrival times, the FI proceeds to make each injection, which involves XORing the contents of the target local memory word with a user-defined mask. The user can direct the FI to inject faults into the kernel or into a specific user process. Additionally, injections can be targeted at the code, data, or stack segments of specific processes.

The FI also has a single-fault mode. In this mode, a single fault is injected. The FI then waits for a predetermined timeout period to elapse, during which time it checks if the injected processor has crashed.[1] If the processor crashes during the timeout period, the FI brings the processor back on-line. In any case (crash or no crash), the FI makes sure that the processor is fault-free.[2] The process is then repeated.

The low-level fault injection mechanism is implemented as a device driver in the kernel. The parameters for a call to the fault injection routine are the CPU to be injected, the address, and the injection XOR mask. The actual injection consists of XORing the contents of that local memory location with the XOR mask. An injection into the cache can also

[1] Each fault-tolerant component is represented by a file in the file system. The file permissions are used to reflect the current state of each component. Thus, a simple check of the appropriate file permissions will determine if a component has crashed.

[2] If the processor previously crashed and was reintegrated, then it is fault-free. If not, then the processor is intentionally downed and reintegrated.

be simulated by invalidating the appropriate cache line, if the location exists in the cache. The next access involving that cache line will force the corrupted memory contents into the cache. If necessary, the write protection for certain parts of memory, such as the kernel, can be temporarily disabled.

3.2 Workload Generator

The workload generator is an important part of any fault injection study. It has been shown that the probability of a system failure is increased by greater processor activity [29]. Also, the relationship between the probability of a CPU-related error and increased workload activity has been established [30]. Since one of the goals of this thesis is to study the relationship between workload performance and fault propagation, the ability to create workloads with specific characteristics is essential.

Because the characteristics of a real program are more difficult to control, a synthetic workload generator is chosen. Our workload generator creates composite workloads out of components representing concentrated CPU, memory, and I/O activity. An attempt is made to model real workloads by allowing the user to create several types of processes and to execute one or more of those types of processes. In addition, parameters for the workload components can be specified as distributions. Workload component parameters include total execution time, location of memory accesses (cache, local, global), and type and number of I/O accesses. Since the workload generator is intended for use in an integrated fault-injection environment, it should be able to communicate to the fault injector such information as the location, size, and composition of the current workload.

3.2.1 Process components

Three types of *process components* (PCs) have been created: CPU, mem, and IO. A description of each process component follows:

CPU Additions, subtractions, multiplications, and divisions are repeatedly performed. Internal CPU registers are used as much as possible, and few memory accesses are made. The number of operations can be controlled.

mem A large array is constructed in memory, and sequential accesses are made with a specified stride. By controlling the stride, some measure of control over the cache hit rate is available. For example, with a stride of 0, the same array location is constantly accessed, yielding a cache hit rate of practically 100%. A stride of 4 bytes (which is the size of a single S2 cache line) will force a cache hit rate of almost 0%. The number of memory accesses can be controlled.

I/O An I/O-based workload generator developed in [31] is used. Using a synthetically generated file system, the I/O generator initiates I/O requests, which are handled by the UNIX operating system. The I/O requests are made in a logical sequence (e.g., a file must be opened before it can be closed), and an attempt is made to model an actual I/O workload. The number of files which are opened, used for reads and writes, and then closed can be controlled.

3.2.2 Workload composition methods

A workload is created by combining PCs in various ways. An example of the workload model can be see in Figure 3.2. A *sequence* is a logical ordering of PCs. An *instance* of

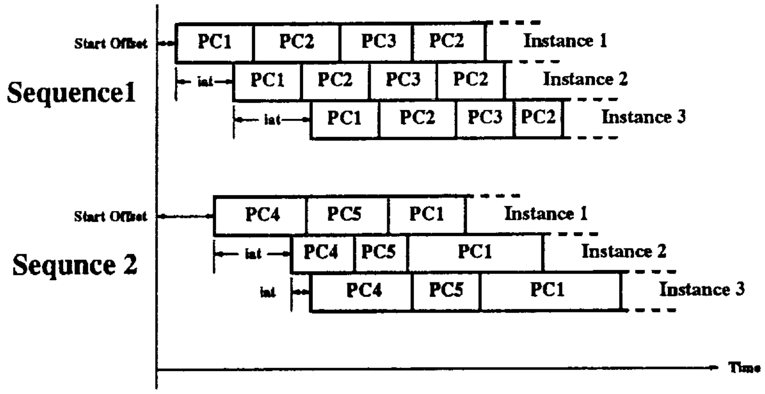


Figure 3.2: Workload Model

a sequence is the executing image of that sequence. Thus, a sequence can be viewed as a program stored on disk, while an instance is a single copy of that program executing in memory. Many instances of a sequence can be started, and each instance can draw the parameters for its PCs from a specified distribution. The interarrival time for the start of the instances of each sequence can also be specified, as well as the *start offset*, which is the time from the start of the WG to the earliest start of an instance for that sequence. By using this sequence/instance, organization, the WG can model processes with a certain composition of CPU, memory, and I/O load, and many instances of these processes can be executed.

Thus, there exist several methods to vary a workload:

1. Execution time for each PC

    - Can be fixed before execution.
    - Can be drawn from a specified distribution at run time.

2. Sequence of PCs

- Can be a fixed sequence of PC types (that is repeated) determined before execution.
- Exact ordering of PCs in each instance can be determined at run time and drawn from a specified distribution.

3. Intensity of each PC
   - For example, relative amount of CPU activity done by the **cpu** PC.
   - May be fixed or drawn from a specified distribution.
4. Number of instances
   - One or more of each sequence may be executed.
   - Must be specified before execution.

3.3 Interaction Between FI and WG

Figure 3.1 shows the interaction between the fault injector and workload generator. The WG is started by the FI. In order for the FI to inject faults into currently active user process space, it must obtain all necessary process IDs (PIDs) from the WG. Once the FI receives the PIDs, it selects one process and determines the required information (virtual location of active pages and virtual-to-physical address translation) for fault injection.

Both the FI and WG use input parameter files to specify necessary variables and distributions. The FI creates an output file that includes time-stamped lines with information such as injection location and mask, error detection, and CPU reintegration.

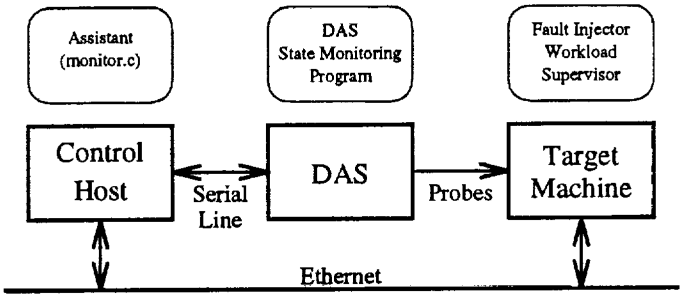


Figure 3.3: Physical Layout of the Hybrid Fault Injection Environment

3.4 The Hybrid Monitor-Based Environment

The hybrid environment uses a hardware monitor to obtain high time resolution and minimize system perturbation due to event detection and logging, as well as a software monitor to assist the fault injector. A detailed description can be found in [18].

The hardware configuration consists of a hardware monitor, a target system, and a control host. The physical layout is given in Figure 3.3.

The target machine is the Tandem S2, which is described in Section 3.5. The hardware monitor is a Tektronix DAS9200 Logic Analyzer (DAS), which has probes attached to the local processor bus on one of the S2's processor boards. Acquired data recorded by the DAS include addresses, data, read/write signals, interrupts, and DMA signals. The control host is a Sun ELC workstation.

The fault injector and workload execute on the S2. In addition, a supervisor program communicates with the fault injector and workload and uses that information to reprogram the DAS. This reprogramming is accomplished by the assistant program on the Sun. The assistant program receives commands from the supervisor program over the local Ethernet. This configuration allows the DAS to dynamically reconfigure itself to adjust to changing conditions on the target machine.

3.5 The Tandem Integrity S2

The Integrity S2 is a fault-tolerant computer designed by Tandem. An in-depth description can be found in [32], and a basic overview of the architecture is given in Figure 3.4. The S2 is UNIX-based (SVR3) and uses the MIPS R2000 microprocessor. The core of the S2 is its triple-modular-redundant processors. Each processor includes a CPU, a cache, and an 8MB local memory. Although these three processors perform the same work, they operate independently of each other until they need to access the doubly replicated global memory. The local processor memory is not parity-protected. Fault detection is performed by the duplexed Triple Modular Redundant Controllers (TMRCs) voters which are activated whenever the global memory is accessed. If an error is found, the faulty processor is shut down, and immediately undergoes a Power-On Self-Test (POST). Upon passing the POST, the processor is reintegrated into the system by copying the states of the two good processors. Voting also occurs on all I/O and interrupts.

The processors are loosely synchronized — they operate independently until TMRC voting occurs. In order to synchronize the processors, at every 2047 instructions each processor is stalled until all other processors arrive at the same synchronization point or a timeout is reached. If the timeout is reached, the appropriate processor is declared faulty and reintegrated.

In addition to these fault detection mechanisms, the local memory is scrubbed periodically. If a fault is discovered, a soft DMA error correction is performed without a POST or

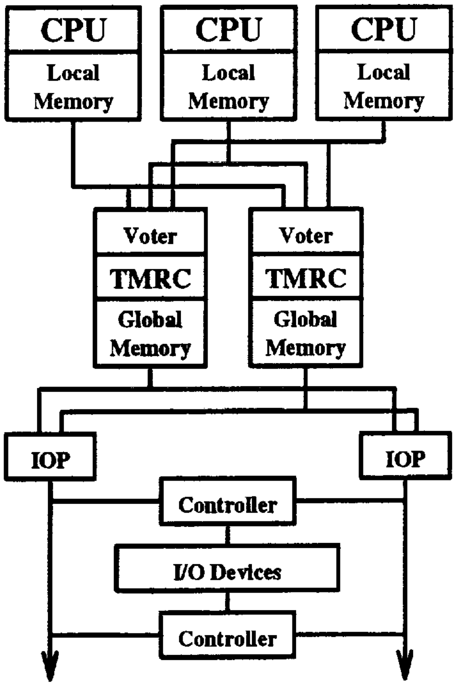


Figure 3.4: Overview of Tandem Integrity S2 Architecture

reintegration. This fault-tolerant architecture ensures that a fault that occurs on one processor will not propagate to other system components without being caught by the TMRC voting process.

4. WORKLOAD CHARACTERIZATION

The workloads used in the experiments in Chapter 5 are characterized in this section. In this thesis, workload characterization refers to using real measurements to obtain the performance characteristics of a workload. In this way, the effect of faults on the performance of a specific workload can be measured.

The specification of the workload is done in terms of how much of the composite workload is constituted by a particular type of process component. For example, a workload that consists of 20% **cpu**, 40% **mem**, and 40% **io** PC would be notated as having a 20/40/40 composition. The number of such processes in the workload also has to be specified. The following two types of analyses are used for workload characterization:

1. Gather process statistics that are kept by the operating system. On the S2, which runs SVR3, the `timex` command is used. (See Section 4.1.) Only the target workload and the OS should be running.

2. Do an instruction-level profile of the workload and determine the ratio of load-store instructions to nonload-store instructions. The **cpu** PC should be mostly nonload-store instructions, while the **mem** PC should be mostly load-store instructions.

These two types of analyses are described in Sections 4.1 and 4.2.

4.1 OS Statistics

Most operating systems keep track of some performance statistics. The version of SVR3 implemented on the S2 includes the `timex` command, which monitors performance statistics for one process (including its children). The workloads used in the experiments described later were executed using the `timex` command. The results of the `timex` command for some of these workloads are given in Tables 4.1 and 4.2. The definition of each statistic can be found in Table 4.3. It was found that the results for workload execution times of 20, 60, and 120 minutes were the same. Therefore, the measurements in this section were performed on workloads that ran for 20 minutes each. The overhead due to the `timex` command is minimal since it merely extracts statistics that the operating system normally updates. Also, this data extraction is only performed once the workload has finished.

Each succeeding workload for workloads A through E is less I/O-intensive and more memory-intensive than the previous workload. Table 4.1 shows that this is indeed the case. The CPU composition is kept the same in order to relate any observed change in results to the relative **mem** and **io** composition. The amount of wait time due to blocked I/O (% wio), the disk busy time (% busy), and the number of system calls per second (syscall/s) all decrease because the number of disk requests decreases. The average disk request queue

Table 4.1: Performance Statistics for Selected Workloads

	Workloads				
	A	B	C	D	E
#processes	1	1	1	1	1
% **cpu** PC	20	20	20	20	20
% **mem** PC	0	20	40	60	80
% **io** PC	80	60	40	20	0
% usr	47	65	78	90	99
% sys	34	22	14	6	1
% wio	19	13	8	4	0
% busy	20	13	9	4	0
avque	1.3	1.3	1.6	2.1	7.0
avwait	9.5	12.7	20.3	38.2	133.7
syscall/s	177	158	105	44	10
pswch/s	20	17	15	13	11
runq-sz	1.0	1.0	1.0	1.0	1.0
% runocc	79	84	91	95	100

Table 4.2: Performance Statistics for Selected Workloads

	Workloads				
	F	G	H	I	J
#processes	1	2	3	5	10
% **cpu** PC	20	20	20	20	20
% **mem** PC	40	40	40	40	40
% **io** PC	40	40	40	40	40
% usr	77	80	81	80	82
% sys	15	17	18	18	17
% wio	8	3	1	1	1
% busy	8	8	9	8	7
avque	1.4	1.6	2.6	2.6	2.9
avwait	13.5	21.6	52.5	54.0	63.1
syscall/s	107	103	104	101	93
pswch/s	16	20	20	22	23
runq-sz	1.0	2.0	2.9	4.9	9.7
% runocc	91	97	99	99	99

Table 4.3: Meaning of Performance Statistics

#processes	How many concurrent processes in the workload
% **cpu** PC	% of the total workload represented by the **cpu** PC
% **mem** PC	% of the total workload represented by the **mem** PC
% **io** PC	% of the total workload represented by the **io** PC
% usr	% of CPU time running in user mode
% sys	% of CPU time running in system mode
% wio	% of CPU time waiting for blocked I/O
% busy	% of time disk was busy servicing a request
avque	outstanding number of disk requests while disk is busy
avwait	average time in milliseconds requests wait in queue
syscall/s	system calls per second
pswch/s	process switches per second
runq-sz	average run queue length while occupied
% runocc	% of time run queue was occupied

length and wait time (avque and avwait) are only updated when the disk is busy. Both avque and avwait increase because these numbers are influenced more and more by the intense disk activity needed at the beginning to load the process image.

Each succeeding workload for workloads F through J has more processes than the previous workload. Table 4.2 shows the effects of increased process concurrency. The average wait time due to blocked I/O (% wio) decreases since other processes can use the CPU while a process is blocked for I/O. The average disk request queue length and wait time (avque and avwait) increase because while one process is blocked for I/O, another process will execute until it is blocked for I/O. This process is repeated for additional processes. Thus, a higher level of process concurrency results in more processes being blocked for I/O at the same time. The number of process switches (pswch/s), the average run queue length (runq_sz), and the percentage of time the run queue is occupied (% runocc) all increase as expected.

4.2 Load-Store Instruction Ratio

The three available types of PCs (**cpu**, **mem**, and **io**) can be viewed in terms of the amount of the resultant data flow. Since instruction fetches occur independently of the workload composition, the focus should be on noninstruction data flow. The S2 is based on the MIPS R2000, which is a load-store architecture processor. Data only enter or leave the R2000 when a load-store instruction is executed. Thus, the ratio of load-store instructions to total instructions is a measure of the amount of data flow in and out of the CPU.

Some expectations can be formed about the *load-store instruction ratio* (LSIR) for each type of PC. The **cpu** PC is supposed to contain mostly CPU-intensive activity. Thus, the **cpu** PC should have a very low LSIR, i.e., most instructions should be register-to-register instructions. The **mem** PC should produce a great deal of memory activity and, therefore, should have a high LSIR. It is not obvious what LSIR the **io** PC should have because most of the data flow is performed via DMA. Table 4.4 shows the measured LSIRs for several workloads.

The LSIR is a dynamic count of load-store instructions which are executed. It was obtained by profiling each workload for one hour. Since the profiling uses sampling, the resultant LSIR measure is approximate. The profiling is also only performed for addresses within the workload memory image and does not include kernel routines.

The data in Table 4.4 are split into two groups of workloads. The first group (**cpu_wkld**, **mem_wkld**, **io_wkld**) consists entirely of one PC type. The second group (A,. . . ,E) uses a combination of several PC types. The very low LSIR (0.20%) for the **cpu_wkld** workload confirms the expectation that the **cpu** PC consists mostly of register-to-register instructions.

Table 4.4: Measured LSIRs for Some Workloads

	Composition			Load-Store
Name	% cpu	% mem	% io	%
cpu_wkld	100	0	0	0.20%
mem_wkld	0	100	0	49.62%
io_wkld	0	0	100	9.23%
A	20	0	80	5.85%
B	20	20	60	19.09%
C	20	40	40	28.44%
D	20	60	20	34.35%
E	20	80	0	41.08%

The relatively high LSIR (49.62%) for the **mem_wkld** workload shows that the **mem** PC has a much higher mix of instructions which access memory. The LSIR for the **io_wkld** workload (9.23%) is not very high. The low LSIR can be explained by examining the programming structures used to construct the three basic types of PCs. The **cpu** PC consists of many arithmetic operations within a tight loop. Similarly, the **mem** PC performs memory accesses within a tight loop. In contrast, the **io** PC performs system calls within a large `switch()` statement and thus contains many nonload-store instructions to manage the control flow within the `switch()` statement.

5. EXPERIMENTS

The following experiments were performed to show the relationship between performance (represented by the workload) and dependability (represented by the error coverages and fault latencies). In Section 5.1, the same fault (same fault type and location) is injected many times into different workloads to show that the propagation path of a fault can be altered just by changing the workload. Section 5.2 repeats the same experiments, but with faults randomly injected into the entire workload.

All fault injections in this section are single-bit flip faults in the local processor memory. Since the local memory is not parity-protected, single-bit flip faults will not be detected until the faulted location is accessed.

5.1 Propagation Paths

In this section, a single fault is injected into different workloads to demonstrate that the generated workloads affect the propagation paths of the faults. Two different fault scenarios are presented.

5.1.1 Fault 1

In this example, five different workloads with varying amounts of the **mem** and **io** PC are used. Each workload has one process.

Workloads: Workloads A-E. (See Section 4.1.).

Fault: Inject into **mem** PC text space. A reserved instruction exception is generated, which results in an error detection when the interrupt is presented to the voter.

Results: Table 5.1 shows that as the **mem** PC is accessed more often, the mean fault latencies decrease. This is as expected. Since the **mem** PC is not used in workload A, the faults injected into it are not accessed and therefore do not cause any error detections.

Table 5.1: Fault 1 Results

Workload	Composition			Errors	Faults	%	Latency
Name	% cpu	% mem	% io	Detected	Injected	Detected	(sec)
A	20	0	80	0	100	0 %	— — —
B	20	20	60	100	100	100 %	2.40±0.73[1]
C	20	40	40	100	100	100 %	0.97±0.21
D	20	60	20	100	100	100 %	0.49±0.13
E	20	80	0	100	100	100 %	0.20±0.05

5.1.2 Fault 2

A fault may be paged out if memory is being fully utilized. In this case, the page will simply be invalidated if it is in text space. If it is in data space, then a dirty page will be written to global memory, causing an error detection. In this experiment, the effect of increasing process concurrency is investigated.

Workloads: Workloads F-J. (See Section 4.1.).

[1] All intervals in this thesis are 95% confidence intervals.

Fault: Inject into **cpu** PC text space. A reserved instruction exception is generated, which results in an error detection when the interrupt is presented to the voter.

Results: Table 5.2 shows the effects of a workload with multiple concurrent processes. As the level of concurrency increases, the number of faults that are paged out increases, thus decreasing the overall error detection percentage. Also, the mean latency of the faults increases initially because the injected location is accessed less frequently due to contention for the CPU. However, as more processes are added, longer latency faults are corrected by being paged out.

Table 5.2: Fault 2 Results

Workload Name	# Processes	Errors Detected	Faults Injected	% Detected	# Paged Out	Latency (sec)
F	1	150	150	100 %	0	3.46±0.79
G	2	150	150	100 %	0	6.75±1.28
H	3	140	150	93.3 %	10	8.00±1.84
I	5	106	150	70.7 %	44	6.37±1.59
J	10	61	150	40.7 %	89	2.23±0.93

5.2 Multiple Fault Results

The workloads from the previous section are used again. However, instead of injecting the same fault type and location, a random fault type and location are chosen for each injection. The fault types are all single-bit flip faults, and the locations are selected from the *active regions* of the workload text region. The active regions are the portions of memory that the workload uses most often. Injections are confined to the active regions in order to increase the probability of accessing an injected fault. If the fault location is chosen from the entire workload text space, then error detection coverages will be decreased proportionately, and fault latencies would remain the same.

5.2.1 Experiment 1: varying composition

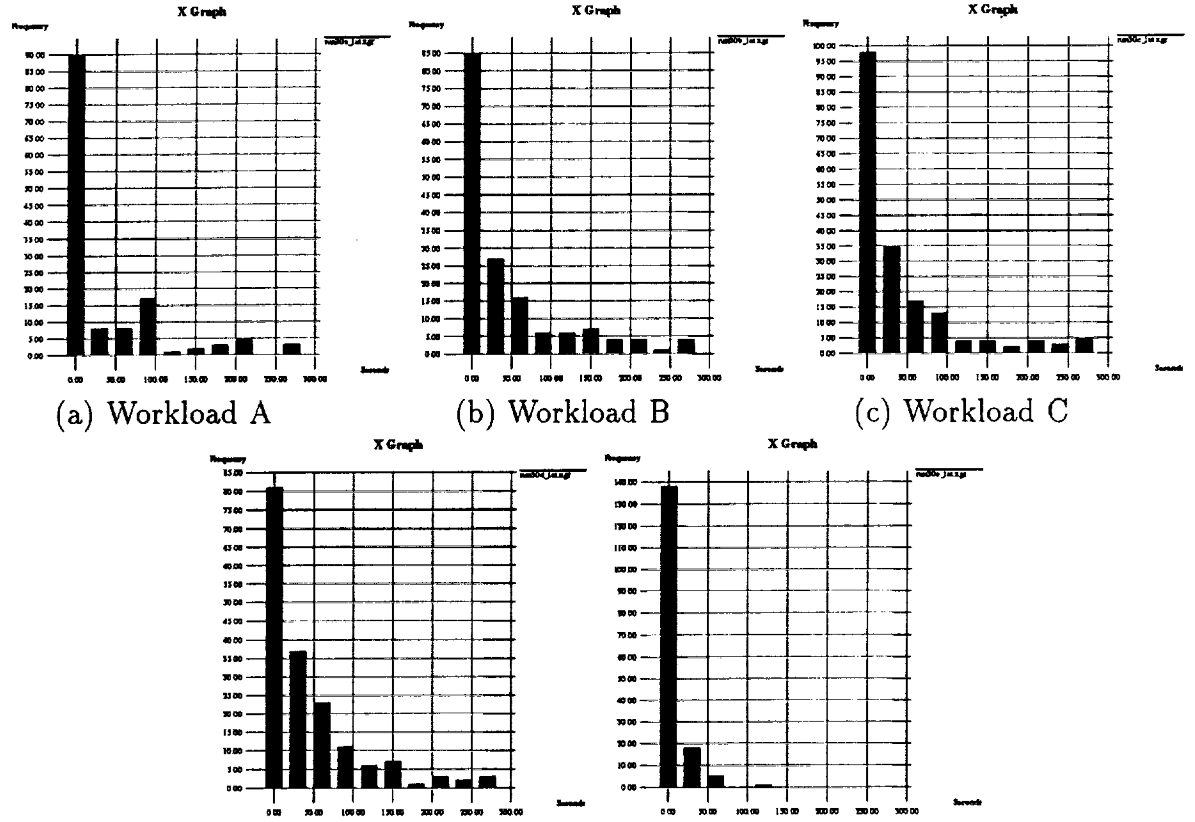
In the first case, five workloads consisting of a gradually increasing amount of **mem** and a decreasing amount of **io** are compared. A constant amount of **cpu** is used for all workloads.[2] The detection ratios and fault latency distributions are given in Table 5.3 and Figures 5.1(a)–5.1(e), respectively.

Table 5.3: Detection Ratios

Workload	Composition			Errors	Faults	%
Name	% cpu	% mem	% io	Detected	Injected	Detected
A	20	0	80	137	352	38.9
B	20	20	60	160	346	46.2
C	20	40	40	185	347	53.3
D	20	60	20	174	353	49.2
E	20	80	0	162	350	46.3

Table 5.3 shows that the highest detection ratio occurs when the workload is most evenly balanced among the three types of PCs. Since all components are evenly accessed in a balanced workload, the detection ratio is high. When the workload is unbalanced, certain parts are less frequently accessed, which decreases the detection ratio. The range of the detection ratios is not large, since fault injection is confined to the active regions. This means that when the **mem** or **io** PC is not used, no faults will be injected into those areas. If the same fault injection region had been used for all five workloads, then the detection

[2] If all three types of PCs had been varied, then it would have been unclear which PC variation had caused the change in results.



(a) Workload A

(b) Workload B

(c) Workload C

(d) Workload D

(e) Workload E

Figure 5.1: Fault Latency Distributions

ratios for the unbalanced workloads (workloads A and E) would have been dramatically smaller.

In Figures 5.1(a)–5.1(e), the fault latency distributions are given for the five workloads. Examination of the distributions shows that the presence of the **io** PC produces a long tail. The distribution for workload E contains no such tail. Furthermore, as the contribution of the **mem** PC increases, the distributions shift to the left.

This phenomenon can be explained in two ways. First, since disk accesses are always slower than memory accesses, the latencies for I/O-bound processes should be longer. Another reason for this effect is the control flow structure of each PC. The **cpu** and **mem** PCs are both tight loops, in which the instructions in the loops are executed many times. The

io PC is different. It consists of a large **switch** statement that is executed inside a loop. Most instructions in the **io** PC are located within the **switch** statement, which means that those instructions are not executed each time through the loop. Thus, the time from when the fault is injected to when it is accessed is longer if the **io** PC is used. The first access time is the time between the time of fault injection and the first time the faulted location is accessed by a read, write, or page out. The average first access time for all workloads is less than 2 seconds.

5.2.2 Experiment 2: concurrent processes

In this experiment with multiple faults, the number of processes was varied from 1 to 10. Each process was composed of 20% **cpu** PC, 40% **mem** PC, and 40% **io** PC. Table 5.4 shows the error detection ratios, mean fault latencies and page-out rates for each workload. There are some differences with single faults between the effects seen here and in Section 5.1. In this experiment, the detection ratios are all lower since faults are injected into the entire workload, which includes some locations that are seldom accessed. The mean latencies are similar to those observed in Section 5.1.

The similarity to results observed with a single fault (Section 5.1) is that as the level of process concurrency increases, the error detection ratio decreases, and the mean fault latency increases. This is an expected result, since the time given to each process by the CPU decreases as the number of processes increases. This results in a longer time to first access and hence larger fault latency. Also, more faults are paged out as the number of processes increases, thus decreasing the error detection ratio.

Table 5.4: Results for Concurrent Processes

Workload	#	Errors	Faults	%	Paged Out		Mean Fault
Name	Processes	Detected	Injected	Detected	#	%	Latency (sec)
F	1	108	229	47.2	0	0.0%	3.79±2.99
G	2	114	234	48.7	0	0.0%	6.02±3.85
H	3	103	230	44.8	59	25.7%	7.85±1.86
	4	82	193	42.5	101	52.3%	9.60±3.95
I	5	90	231	39.0	128	55.4%	7.99±5.43
	6	85	197	43.2	106	53.8%	5.72±1.72
	7	53	191	27.8	132	69.1%	4.93±1.73
	8	50	194	25.8	138	71.1%	3.23±1.23
	9	62	192	32.9	122	63.5%	5.04±1.93
J	10	46	205	22.4	151	73.7%	4.74±1.83

6. CONCLUSIONS

This thesis presented a study of the relationship between the performance and dependability of fault-tolerant computers. To perform this study, a synthetic workload generator/fault injector tool (FTAPE) was developed. FTAPE allowed the user under controlled conditions to stress the test machine in terms of both workloads and fault injections. The hybrid monitor-based environment was used on the Tandem Integrity S2 computer to perform the actual experiments.

Workload characterization was first performed to extract the performance characteristics of a particular workload. Then faults were injected into those workloads to obtain the dependability measures (error detection coverage and fault latency) for those workloads.

Comparing the effects of different workloads on the propagation path of a single fault showed that the workload could be generated in a controllable manner to affect the injected fault. Injections into the entire workload showed that balanced workloads have slightly higher error detection ratios. It was also shown that the effect of increasing the number of concurrent processes has two effects: (1) the mean fault latency initially increased due to competition for the CPU and later decreased due to the paging out of long latency faults, and (2) the error detection coverage decreased because some faults were paged out, and thus were corrected before propagating.

There are many directions for future work. Perhaps a single composite measure of the workload performance can be used. The workload characterization can also be performed after each fault injection in order to better understand the effect of the fault on the performance of the workload. The workload generator and the PCs it uses may be improved. The fault injector can also be extended to inject into global memory, multiple processors, and the I/O system. A long-term goal is to perform this study on another fault-tolerant architecture, not only to measure the performance of the machines, but also to make a comparative study.

REFERENCES

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

[2] D. Lomelino and R. Iyer, "Error propagation in a digital avionic processor: A simulation-based study," Tech. Rep. NASA CR-176501, University of Illinois, Urbana, Illinois, 1986.

[3] G. S. Choi, R. K. Iyer, and V. A. Carreno, "Simulated fault injection: A methodology to evaluate fault tolerant microprocessor architectures," *IEEE Transactions on Reliability-Special Issue on Experimental Evaluation*, vol. 39, pp. 486–491, October 1990.

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

[5] K. Goswami and R. Iyer, "Depend: A design environment for prediction and evaluation of system dependability," in *9th Digital Avionics System Conference*, October 1990.

[6] K. Goswami and R. Iyer, "A simulation-based study of a triple modular redundant system using depend," in *5th International FTRS Conference*, (Nuremberg, Germany), September 1991.

[7] E. W. Czeck, On the Prediction of Fault Behavior Based on Workload, Ph.D., dissertation, Carnegie Mellon University, April 1991.

[8] R. Chillarege and R. K. Iyer, "Measurement-based analysis of error latency," *IEEE Transactions on Computers*, vol. 36, pp. 529–537, May 1987.

[9] S. G. Mitra and R. K. Iyer, "Measurement-based analysis of multiple latent errors and near-coincident fault discovery in a shared memory multiprocessor," in *Proceedings 1988 International Conference on Parallel Processing*, (St. Charles, Illinois), pp. 404–409, August 1988.

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

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

[12] J. Lala, "Fault detection, isolation, and reconfiguration in ftmp: Methods and experimental results," in *Proceedings of the 5th AIAA/IEEE Digital Avionics Systems Conference (DASC)*, pp. 21.3.1–21.3.9, 1983.

[13] K. G. Shin and Y. H. Lee, "Measurement and application of fault latency," *IEEE Transactions on Computers*, vol. 35, pp. 370–375, April 1986.

[14] G. B. Finelli, "Characterization of fault recovery through fault injection on ftmp," *IEEE Transactions on Reliability*, vol. 36, pp. 164–170, June 1987.

[15] J. Arlat et al., "Fault injection for dependability validation-a methodology and some applications," *IEEE Transactions on Software Engineering*, vol. 16, pp. 166–182, February 1990.

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

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

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

[19] J. Ousterhout et al., "A trace-driven analysis of the unix 4.2 bsd file system," in *Proceedings of the 10th ACM Symposium on Operating System Principles*, pp. 15–24, 1985.

[20] G. Serazzi, *Workload Characterization of Computer Systems and Computer Cetworks.* Amsterdam, Netherlands: Elsevier Science Publishing, 1986.

[21] J. H. Howard et al., "Scale and performance in a distributed file system," *ACM Transactions on Computer Systems*, vol. 6, pp. 51–81, February 1988.

[22] D. Ferrari, "On the foundations of artificial workload design," in *Proceedings of the 1984 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, (Cambridge, Massachusetts), pp. 8–14, August 1984.

[23] W. Buchholz, "A synthetic job for measuring system performance," *IBM Systems Journal*, vol. 8, no. 4, pp. 309–318, 1969.

[24] D. C. Wood and E. H. Forman, "Throughput measurement using a synthetic job stream," in *AFIPS Conference Proceedings FJCC*, vol. 39, pp. 51–56, 1971.

[25] K. Sreenivasan and A. J. Kleinman, "On the construction of a representative synthetic workload," *CACM*, vol. 17, pp. 127–133, March 1974.

[26] D. Ferrari, "Workload characterization and selection in computer performance measurement," *Computer*, vol. 5, July/August 1972.

[27] H. P. Artis, "Workload characterization using sas proc fastclus," in *The International Workshop on Workload Characterization of Computer Systems and Computer Networks*, (Pavia, Italy), pp. 21–32, October 1985.

[28] A. K. Agrawala, J. M. Mohr, and R. M. Bryant, "An approach to the workload characterization problem," *Computer*, vol. 9, pp. 18–32, June 1976.

[29] E. W. Czeck, "Observations on the effects of fault manifestation as a function of workload," *IEEE Transactions on Computers*, vol. 41, pp. 559–566, May 1992.

[30] R. Iyer and D. Rossetti, "A measurement-based model for workload dependence of cpu errors," *IEEE Transactions on Computers*, vol. 35, pp. 511–519, June 1986.

[31] W.-L. Kao, "An user-oriented synthetic workload generator," in *12th International Conference on Distributed Computing Systems*, May 1992.

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