Stress-based and Path-based Fault Injection

Timothy K. Tsai
Bell Labs, Lucent Technologies
Murray Hill, NJ 07974 USA
ttsai@research.bell-labs.com

Mei-Chen Hsueh, Hong Zhao, Zbigniew Kalbarczyk, and Ravishankar K. Iyer
Center for Reliable and High Performance Computing
Coordinated Science Laboratory
University of Illinois, Urbana, IL 61801 USA
{hsueh,hzhao,kalbar,iyer}@crhc.uiuc.edu

Abstract

The objective of fault injection is to mimic the existence of faults and to force the exercise of the fault tolerance mechanisms of the target system. To maximize the efficacy of each injection, the locations, timing, and conditions for faults being injected must be carefully chosen. Faults should be injected with a high probability of being accessed. This paper presents two fault injection methodologies–stress-based injection and path-based injection; both are based on resource activity analysis to ensure that injections cause fault tolerance activity and thus the resulting exercise of fault tolerance mechanisms. The difference between these two methods is that stress-based injection validates the system dependability by monitoring the run-time workload activity at the system level to select faults that coincide with the locations and times of greatest workload activity, while path-based injection validates the system from the application perspective by using an analysis of the program flow and resource usage at the application program level to select faults during the program execution. These two injection methodologies focus separately on the system and process viewpoints to facilitate the testing of system dependability. Details of these two injection methodologies are discussed in this paper along with their implementations, experimental results, and advantages and disadvantages.

Keywords: fault injection, stress-based, path-based, workload, program flow, fault tolerance, dependability.

1 Introduction

To achieve high dependability, systems today are often designed with fault tolerance features to first detect errors and then to mask or recover from the effects of those errors. Thus, testing of these features is extremely important in understanding how dependable the systems are with the incorporated fault tolerance mechanisms and in gaining insight into the success and

behavior of error detection and recovery. Fault injection is a means to effectively test and stress these fault tolerance mechanisms so that the system behavior can be studied prior to their actual deployment. The fault injection methods discussed in this paper address the issue of improving the effectiveness and efficiency of dependability testing.

In this paper, fault injection is discussed from a software-implemented fault injection perspective, although the methods for selection of faults are not limited to software-implemented fault injection. In contrast to other techniques where low-level physical faults are injected into hardware, software-implemented techniques emulate the effect of those low-level faults by directly introducing changes to the system state (e.g., contents of memory, registers, etc.) that would have resulted from the low-level faults. This type of injection may perhaps more properly be called *error injection*. However, we will use the more common term *fault injection*.

The objective of fault injection is to mimic the existence of faults and errors and consequently to force the exercise of the fault tolerance components of the target system. To maximize the efficacy of each injection, the locations, timing, and conditions for faults being injected must be carefully chosen. Faults should be injected in locations with a high probability of being activated; conversely, faults must not be placed where activation is very unlikely. For example, faults in memory locations that are not assigned to active processes should be avoided because these locations will not be accessed. Furthermore, non-activated faults incur an undesirable time-cost equivalent to that for activated faults, and thus should be minimized. Random injection, although simple, falls short in this aspect. This paper presents two fault injection methodologies that provide a high level of fault activation activity: stress-based injection and path-based injection. Stress-based injection validates the system as a whole. It monitors the run-time workload activity at the system level to guide fault injection that coincides with the locations and times of greatest workload activity. Path-based injection, on the other hand, analyzes the system from an application standpoint. It uses an analysis of the program flow and resource usage at the application program level to guide fault injection during the program execution to guarantee high fault activation.

By providing a high level of fault activation, these injection strategies increase the efficiency of fault injection. More activated faults can be injected in a given amount of time, which in turn will result in more thorough testing of the target system and potentially greater coverage of the fault space.

Stress-based injection is the process of injecting faults based upon a measurement of the current workload activity. Stress in this sense refers to the amount of activity caused by the workload that could encourage fault tolerance activity. Fault tolerance activity refers to the additional activity caused by the presence of faults that are activated. Such activity

includes corruption of the system state from the error-free case, detection of errors, and recovery actions.

The goal of stress-based injection is to select faults that will increase the level of fault tolerance activity, thereby yielding a more thorough exercise of the system fault tolerance mechanisms. To demonstrate the benefit of stress-based injection, several fault injection experiments are conducted. To quantify the resultant level of fault tolerance activity, two quantities are measured: the number of errors detected per injected fault and the performance degradation. These two quantities are compared for faults selected with a stress-based injection methodology and for faults selected randomly. Results from experiments are provided to demonstrate the effectiveness of stress-based injection in increasing and fault tolerance activity.

Path-based injection is an approach that minimizes non-activated faults through an intelligent selection of fault parameters. Specifically, fault times and locations are chosen based upon a pre-injection analysis of the resource usage of a test program. For example, if memory faults are to be injected, the memory usage of the program is analyzed to determine the set of memory locations that are used and the times when faults would be activated. Thus, not only is the set of activatable faults known, the program inputs that will cause each fault to be activated are also known for each fault. The approach is called "path-based" because the selection of fault parameters is based on knowledge of the test program's execution path and resource usage, which is dependent to a large degree upon the program's control-flow paths.

Path-based injection is especially useful for testing the fault impact for specific applications. One example of a specific application that could benefit from path-based injection is the verification of error detection mechanisms that utilize program execution path information, such as the use of watchdog signatures to detect flow-related errors like control-flow errors. Watchdog error detection is based on the assumption that faults will cause control-flow errors or corrupt data and consequently produce wrong results. For example, anything that prevents the proper updating of the program counter could result in a control-flow error. Even so, the fault injection method used to verify the watchdog signature design is still based on random injection due to its simplicity. Our path-based fault injection utilizes the control-flow graph of the program in conjunction with the "architecture" resources of the hardware system. This allows us to determine the location and timing of faults with greater precision.

Fault injection is an attractive approach to the experimental validation of dependable systems. It is employed to conduct detailed studies of the complex interactions between faults and fault handling mechanisms, e.g., [1], [2], and [3]. In particular, fault injection aims at

(1) exposing deficiencies of fault tolerance mechanisms (i.e., fault removal), e.g., [4], and (2) evaluating coverage of fault tolerance mechanisms (i.e., fault forecasting), e.g., [5]. Over the years, significant progress has been made in fault injection based validation of fault-tolerant systems. Numerous tools were proposed to support fault injection analysis and evaluation of systems. These include simulation-based tools such as FOCUS [6] and MEFISTO [7], hardware-implemented tools such as FIST [8] and MESSALINE [2], software-implemented tools such as FIAT [9], FERRARI [10], DEFINE [11], DOCTOR [12], and Xception [13]. Fault injection was also used to study fault-tolerant communication protocols, e.g., [14]. It is important to emphasize that most fault injection studies address effects of physical hardware faults. Only a few studies consider software faults and use software faults or errors models to drive fault injection experiments, e.g., [15][16].

Studies have established that error rates are influenced by the workload, (e.g., [17]). In [1] a "fault acceleration" is achieved by injecting faults only into a page that is currently in use and by applying a workload that fully utilizes CPU and I/O capacity. Some of the already mentioned software-implemented fault injection tools, like FERRARI and DEFINE, employ software traps and trap handling routines to force injection of faults at a certain point in the control flow of the test program. However, fault activation is not guaranteed because usage of the injected resources is not considered. In [18] the operational profile of register utilization is used to determine times and locations of fault injection in order to minimize non-activated faults.

The existing fault injection tools incorporate a wide range of techniques for low-level fault injection. However, whereas these tools address the question of how to inject faults, stress-based and path-based injection address the higher-level issue of which faults to inject. The set of faults to inject is selected based upon monitoring and analysis of the workload, in terms of the test program's structure and resource usage. Thus, these fault injection methodologies focus on software rather than hardware. In addition, no specific low-level fault injection mechanism is assumed by either injection methodology, which are thus compatible with many different low-level fault injection mechanisms. The software-implemented fault injection approach is used in the experiments presented in this paper due to its ease of implementation and low cost.

The remainder of this paper is as follows: Section 2 presents the detailed fundamentals for the stress-based and path-based injection methodologies. For each methodology, the implementation and experimental results are presented in Sections 3 and 4, respectively. The final Sections 5 and 6 discuss the advantages and disadvantages associated with the two methodologies and provide a summary and conclusions.

2 Fundamentals

As we mentioned earlier, the locations, timing, and conditions for faults being injected must be carefully chosen in order to maximize the efficacy of each injection. Stress-based and path-based are both injection methodologies based on resource activity analysis that increase the error detection and recovery effects of the injected faults. The fundamental difference between these two methods is that stress-based injection takes into account the resource usage by the workload activity at the system level while path-based injection considers the specific resources used by the execution path at the process level. Before we provide details of these two methods, we need to define some terms that are used throughout this paper:

- The *test program* is the program that is executed as the fault is injected. The test program produces a demand on system resources, and this demand is called the workload.
- A *fault* is a low-level anomaly that causes a deviation in the system state. The deviation in the system state is called an *error*. [19] Specifically, *fault injection* is the emulation of faults. However, because the emulation of errors often has the same effect as the emulation of faults and is often easier to implement, the term fault injection is conventionally used to collectively refer to both types of emulation.
- *Fault activation* refers to the activation or access of injected faults. For example, fault injected into a register is activated when that register is read before the register is overwritten. Faults that are not accessed will not have any effect on the system and therefore are similar to faults that are not activated. We used this view of fault activation because the determination that a fault has been accessed is relatively easy to make.
- *Fault activation level* refers to the ratio of the number of activated faults (F_a) to the total number of injected faults (F_i). Therefore,

$$\text{fault activation level} = \frac{F_a}{F_i} \tag{1}$$

2.1 Stress-based

The goal of stress-based injection (SBI) is to maximize the amount of system-wide fault tolerance activity. It is well known that high stress and complex workloads cause greater propagation and detection of errors. This has been shown empirically [20] and analytically [17]. By using knowledge of the workload activity, faults can be injected to maximize the chance of activation. We define stress-based injection as the process of injecting faults based upon a measurement of the current workload activity. Stress in this sense refers to the amount of activity caused by the workload that could encourage fault tolerance activity.

The workload activity is monitored by a workload activity measurement tool that provides two values: (1) the level of workload activity in each system component (e.g., CPU, memory, and disk), which determines the location of injection, and (2) the level of workload activity in the entire system, which determines the time of injection.

Two sets of experiments are conducted to demonstrate the effectiveness of stress-based injection in increasing the level of fault tolerance activity, which is measured in two ways: (1) the number of errors detected per injected fault and (2) the performance degradation due to faults. The more errors propagate, the more the number of error detections is likely to increase. A larger number of error detections causes more recovery activity and hence increases the performance degradation.

To measure performance degradation, the workload program is executed twice, once with fault injection and a second time without faults. Section 3.4.1 describes the detailed procedure for measuring the performance degradation. In addition to performance degradation, the ratio of error detections to fault injections is measured for each run. This ratio represents the effectiveness of error detection. Since it is usually desirable to detect as many errors as possible, the error/fault ratio should be maximized. It should be noted that multiple injected faults might be concurrently present in a system component. When a single error in that component is detected, reintegration of that component may result in correction and removal of all errors in that component. Thus, the error/fault ratio will never be greater than unity, and will usually be less than unity.

2.2 Path-based

Path-based injection is focused on a specific application executing on the target system. The goal of path-based injection is to maximize the level of fault activation based on the application execution characteristics. In order to explain the details, additional definitions are needed. An *input* is the set of data that the test program processes and may include command-line arguments, contents of files, environment variables, and file system state. A *path* is the sequence of test program instructions that are executed based upon a given input. *Resources* are the system state components that may be injected with a fault. The system state consists of the contents of CPU registers and memory locations, as shown in Figure 1. The system state is limited to CPU registers and memory locations in this paper because our fault injections were confined to registers and memory locations. More system state can be considered if faults are injected elsewhere. The time can be expressed in terms of the currently executing instruction in the test program, which is called the *stop address*[1]. The

[1] This program instruction is referred to as the stop address because the program is stopped at that address when the fault injector is activated.

fault location is either the register or memory location.

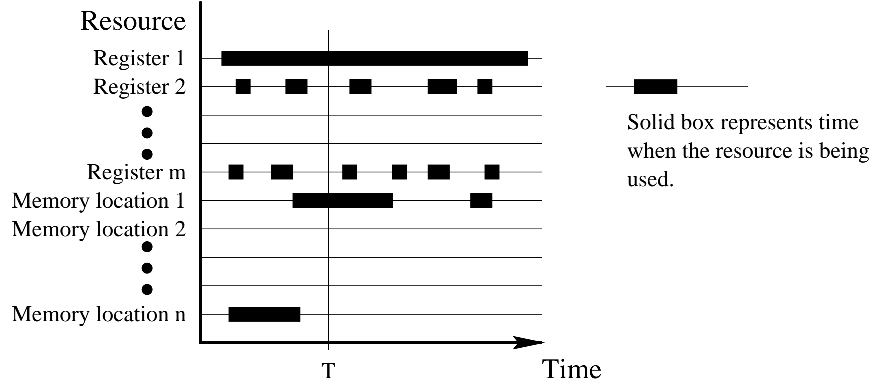

Figure 1: Resources Used by the Test Program Over Time

In path-based injection, a fault is injected into a resource that will be used by the program with a given test program input. A resource is any part of the system state (e.g., memory, registers, etc.) that can be used as input to the program. For a given program, only a few resources are exercised at any single instant of time. Figure 1 conceptually illustrates the resource allocation and usage during the program execution. In this figure, the resource usage for CPU registers corresponds to those times when each register contains a valid value (i.e., a value that is meaningful to the program), or in compiler terminology, when each register is live. Similarly for memory locations, the resource usage corresponds to those times when each memory location contains a valid value – these memory locations will also be called "live." At any particular time (e.g., time T), only a few registers and memory locations contain valid values. For a fault to be activated, the duration of the fault must overlap with the period that the corresponding register or memory location is live. This requirement is true for both permanent and transient faults. For permanent faults, the duration is infinite, and therefore the fault can be activated at any subsequent time when the corresponding register or memory location is live. Selecting the time of injection to be within the period when the register or memory location is live is a simple method to guarantee the overlap of fault latency and resource usage. Only faults injected into live registers and memory locations can be activated. The test program will never access faults injected into the other registers and memory locations, unless first overwritten with an initialization value, in which case the fault is avoided and will never be activated.

Thus, path-based injection guarantees activation of each fault, as long as the fault location is in the set of live resources at the fault time for at least one path. By assembling an input set that results in paths that cover most of the test program code, a large set of interesting activatable faults can be achieved. For instance, if the input set results in paths that cover 100% of the test program code, then all control-flow faults for that test program

are guaranteed to be activatable by path-based injection.

The next two sections describe in detail the two injection methodologies, along with their implementation and results that demonstrate their effectiveness.

3 Stress-based Injection

FTAPE (Fault Tolerance and Performance Evaluator) is a fault injection tool that relies on stress-based injection, by integrating the injection of faults and the generation of the workload necessary to propagate those faults. The tool is composed of three main parts: FI (the fault injector), Measure, and WG (the workload generator). Figure 2 shows how these three parts interact. A detailed description of each part of the tool follows.

The description of the tool is centered on an implementation on a Tandem Integrity S2 fault-tolerant computer, which is described in [21]. The S2 is a TMR-based computer that votes on all global memory accesses, I/O accesses, and interrupts. The main mode of error detection occurs when a voting mismatch is found. If a mismatch of processors or memory is detected, then the content of the remaining good units is copied to the faulty unit. Thus, no restart of the operating system or applications is needed. For the disk system, disk mirroring is used, with one part designated as the primary.

The FTAPE tool is used in several experiments to demonstrate the effectiveness of stress-based injection in increasing fault tolerance activity. The first set of experiments, described in Section 3.4.2, involves injecting matched faults (i.e., faults that are injected into areas of greatest workload stress) and unmatched faults (i.e., faults that are injected into areas of least workload stress). These experiments illustrate the sensitivity of certain workloads to specific faults. The next set of experiments, presented in Section 3.4.3, illustrates the effectiveness of stress-based injections in increasing fault tolerance activity.

3.1 Fault Injector

The method of injection used by FTAPE is software-implemented fault injection, which uses software to emulate the effects of underlying physical faults. For instance, a bit in a memory location can be flipped to emulate the effect of an alpha particle on a memory bit. This method of fault injection is selected because it is more controllable than hardware-based injection and does not require additional injection hardware.

The main goal of fault injection is to exercise the error detection and recovery mechanisms in the target system. The best way to do this is to inject faults throughout the entire system. FTAPE partitions the system into three main areas: *cpu*, *mem*, and *io*. For each area, a different method of fault injection is required. These areas are also targeted by

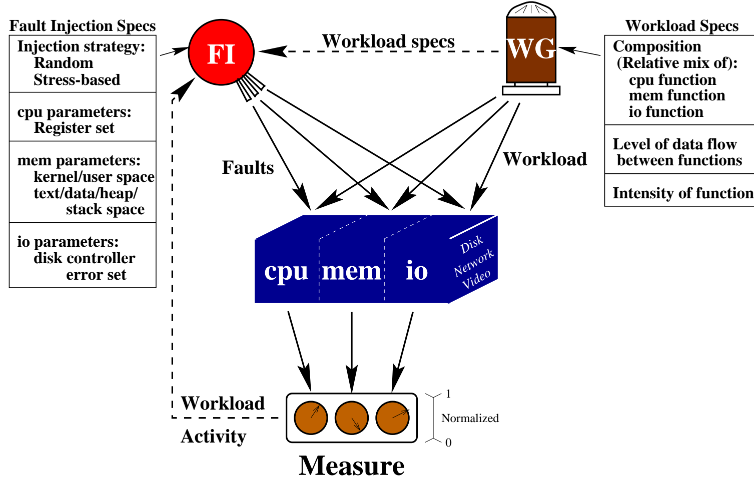

Figure 2: Block Diagram of FTAPE

the WG to increase the chance for the injected faults to be propagated by the workload. CPU and memory faults can be injected directly into the operating system space, although for the experiment described in this paper, CPU and memory faults are only injected into application process space.

The fault injection methods used by the FI are described below. For our implementation, low-level, hardware faults (such as bit-flips in registers and memory) are targeted. Note that fault-tolerant systems have widely varying architectures and therefore require different fault injection techniques. In the following, we describe the implementation of FTAPE on the Tandem Integrity S2:

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

Faults are injected into CPU registers, specifically, saved[2] general purpose and floating point registers, the program counter, the global pointer, and stack pointer. These registers were chosen because errors in these registers have a higher chance of propagation compared to faults in other registers (e.g., temporary registers).

Injection method 2: inject_mem The memory fault models include single/multiple bit-flip and zero/set faults in local and global memory.

Faults are targeted at heavily used parts of memory. Faults are injected by directly modifying the contents of selected memory locations.

Injection method 3: inject_io The I/O fault models include valid SCSI and disk errors.

Faults are injected into a mirrored disk system. The method of injection involves using a test portion of the disk driver code that sets error flags for the next driver request.

[2] *Saved* registers are those registers whose values must be preserved across procedure calls.

Thus, the next request activates the error handler in the driver code, and one half of the disk mirror may be disabled. This type of injection only exercises the ability of the system to handle these exceptional conditions and does not test the ability of the system to handle bit or byte-based disk faults.

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

Selection method 1: location stress-based injection (LSBI) Faults are injected into the area (CPU, memory, or I/O) with the greatest normalized stress. The normalized stress is obtained by remapping the stress measurement for each area to a value between 0 and 1. Section 3.3 provides more details on obtaining the normalized stress. The fault injection time can be chosen randomly (according to a specified inter-arrival distribution) or based on Selection method 2.

Selection method 2: time stress-based injection (TSBI) Faults are injected during the time the composite system stress (described later) is greater than a specific threshold. The area to inject (CPU, memory, or I/O) is chosen randomly.

Selection method 3: randomly The fault time is selected randomly based on a specified distribution (e.g., an exponential inter-arrival distribution with a specified mean of 20 seconds), and the fault location is randomly chosen based on a uniform distribution.

Within each area, the specific fault parameters are chosen randomly. For example, for a CPU fault, a CPU register and a bit within that register would be randomly chosen for injection. The first two selection methods may also be combined so that the area to inject and the time for injection are both chosen based on stress measurements.

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

3.2 Workload Generator

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

Function 1: use_cpu This function is CPU-intensive. It consists of repeated additions, subtractions, multiplications, and divisions for integer and floating point variables. These operations are performed in a loop containing conditional branches. Memory accesses are limited by using CPU registers as much as possible.

Function 2: use_mem This function is memory-intensive. A large memory array is created, and locations in this array are repeatedly read from and written to sequentially. The array is larger than the size of the data cache to ensure that accesses are being made to the physical memory (because the cache is direct mapped with a block size of one word).

Function 3: use_io This function is I/O-intensive. A dummy file system is created on a mirrored disk system. Opens, reads, writes, and closes are repeatedly performed.

In practice, each function is usually specified to last the same amount of time (e.g., one second). Then the composition of each workload process can be specified to contain a specific proportion of each function. For instance, a workload that is CPU-intensive with a small amount of memory and I/O activity can be specified to contain 90% of the *cpu* function and 5% each of the *mem* and *io* functions. Such a workload would be said to have a *composition* of 90/5/5. When the workload process is executed, each function will be randomly chosen according to the corresponding probabilities.

Each function also reads and writes data from a special global *interdependence array* that forces data flow among functions. This is necessary to encourage error propagation among functions. Otherwise, a data fault in one function is usually overwritten if the fault influences only variables local to that function and the system doesn't detect the error before the end of the function. Resetting parts of the array to default values can control the amount of data flow among functions via the interdependence array. This has the effect of providing some measure of control of data flow, and therefore error propagation, through global variables.

The *intensity* is the amount of activity in each function relative to the maximum possible activity. The intensity of each function is decreased by substituting calls to the `usleep()` function instead of code that would otherwise be executed. Thus, an intensity of 100% would contain no additional `usleep()` calls. The ability to control the intensity is useful for studying the impact of the workload activity level on the propagation of errors and the resulting fault tolerance activity. For most of the workloads used in the experiments in Section 3.4, the intensity is varied from 100% to 20% over a period of about nine minutes.[3] Varying the intensity emphasizes the effect of high and low workload activity on the amount of error propagation.

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

The workload generator does not contain any code to detect data errors because the purpose of the fault injection is to determine how well the underlying system detects and tolerates errors. Finally, the workload provides the FI with information needed to determine the location of certain faults, such as which processes are currently executing and what portions of memory are being used.

3.3 Measure

Measure is a tool that monitors the actual workload activity. Although each workload function is designed to be very intensive for one system area, each function must necessarily cause activity in other system areas. For instance, the *io* function must also use the CPU and perform memory reads and writes as well as accessing the disk. Thus, the Measure tool is necessary to measure the actual activity caused by the workload.

Measure returns the level of workload *stress* for each system area as well as for the system as a whole. The stress is the amount of workload activity, especially that which can lead to an increase in fault tolerance activity. As with the FI, the methods needed to obtain the stress measures for each system area are system dependent to some extent. For each system area, the following methods are used to obtain the workload stress:

Method 1: measure_cpu The stress measure is based upon the CPU utilization. On the S2, the `sar` [4] utility returns the CPU utilization.

Method 2: measure_mem The stress measure is based upon the number of reads and writes per second to the memory space used by the workload. Since any software method of obtaining this information would incur an unacceptable amount of overhead, a hardware method is used. A Tektronix DAS 9200 logic analyzer is used to count the number of memory accesses by monitoring the CPU-memory bus. Thus, only actual accesses to the primary memory (and not the cache) are counted. This count is automatically sent to the Measure program every 10 seconds. A detailed description of the setup needed to measure *mem* stress can be found in Young[22].

Method 3: measure_io The stress measure is based on the number of disk blocks accessed per second. On the S2, the `sar` utility returns the number of disk blocks accessed per second.

Each stress measure is normalized to compare the different measures. The normalization is performed by running a set of various workloads[5] and obtaining a distribution of the raw

[4] The `sar` utility is present on most System V Unix systems. It queries measurements that are collected by the operating system.

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

stress measures (i.e., CPU utilization, memory accesses/second, and disk blocks/second). Each raw stress measure was normalized to a value between 0 and 1, inclusively, based on the following formula, where X_{min} is the 5th percentile value and X_{max} is the 95th percentile value in the raw stress distribution:

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

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

3.4 Results

The target machine for these experiments is the Tandem Integrity S2 fault-tolerant computer, which is described in [21]. The general experimental procedure is described in Section 3.4.1. The first set of experiments, described in Section 3.4.2, involves injecting matched faults (i.e., faults that are injected into areas of greatest workload stress) and unmatched faults (i.e., faults that are injected into areas of least workload stress). These experiments expose the sensitivity of certain workloads to specific faults. The next set of experiments, presented in Section 3.4.3, illustrates the effectiveness of stress-based injections in increasing fault tolerance activity.

3.4.1 General Experimental Procedure

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

$$\begin{array}{l}\text{Performance} \\ \text{Degradation}\end{array} = \frac{1}{n}\left(\frac{T_f}{T_{nf}} - 1\right). \tag{2}$$

Performance degradation is a measure of the amount of extra time a system requires to recover from detected errors. Performance degradation can be used as a measure of a system's

fault tolerance, where a lower level of degradation means that the recovery mechanisms are more efficient. To obtain this measure, runs of the experiment that cause system crashes are ignored, since the degradation would be infinite in that case. Instead, the number of system crashes is counted and can be used as another measure of fault tolerance (i.e., how well the system is able to recover from faults).

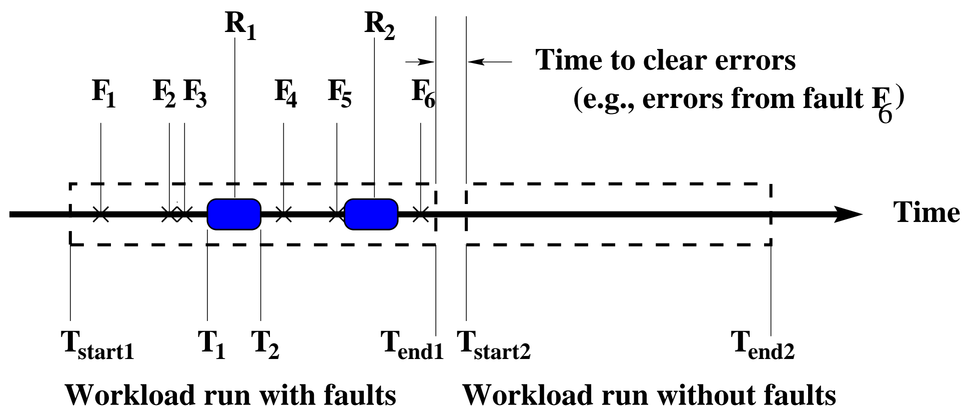

Figure 3: One stress-based injection "run"

Figure 3 illustrates one run of an experiment, which consists of the following steps:

1. Start the Measure tool.
2. Run the workload while injecting faults. Measure the total workload time required (T_f in Equation 2 or $T_{end1} - T_{start1}$ in Figure 3).
3. Run the workload a second time, this time without injecting any faults. Again, measure the total workload time required ($T_n f$ in Equation 2 or $T_{end2} - T_{start2}$ in Figure 3).

The selection of faults is determined by using one of the fault selection methods described in Section 3.1. For the second (non-injection) run, the FI is still executed, but with null injection masks. In other words, the FI goes through the motions of injecting faults, but instead of flipping a bit (XORing with a 1) and setting a disk error (setting the error value to a nonzero value), the FI does not flip a bit (XORs with a 0) and sets a null disk error (sets the error value to a zero value). By so doing, the second run will also invoke the same FI overhead as the first run, which is needed when comparing workload execution times. Multiple faults are injected for the initial (injection) run in order to increase the likelihood of error detection and recovery. The presence of multiple faults in the system does not affect the measured performance degradation because the performance will only suffer when errors are detected.

In addition to performance degradation, the ratio of error detections to fault injections is measured for each run. This ratio represents the effectiveness of error detection. Since

it is usually desirable to detect as many errors as possible, the error/fault ratio should be maximized. Although analogous to error detection coverage,[6] it is different because multiple injected faults may be concurrently present in a system component. When a single error in that component is detected, reintegration of that component results in correction and removal of all errors in that component, without explicit detection the additional errors in that component. Thus, the error/fault ratio is always less than or equal to the error detection coverage.

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

3.4.2 Sensitivity of Workloads to Faults

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

The results are given in Table 1. Each row represents seven runs. From the table, it can be seen that the error/fault ratio and the performance degradation are higher for the second row of each experiment. This means that the fault tolerance activity is indeed higher when the injection location matches the location of high workload activity. For instance, the error/fault ratio for *io* injections increases from 0.248 to 0.700 when the workload activity becomes disk-intensive. Similarly, the performance degradation increases from 0.001257 to 0.030363.

The increase in the error/fault ratio occurs because the injected faults are accessed by

[6] Error detection coverage is the percentage of total errors that are detected.

Table 1: Sensitivity of Workloads to Faults

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

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

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

3.4.3 Stress-based Injection Results

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

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

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

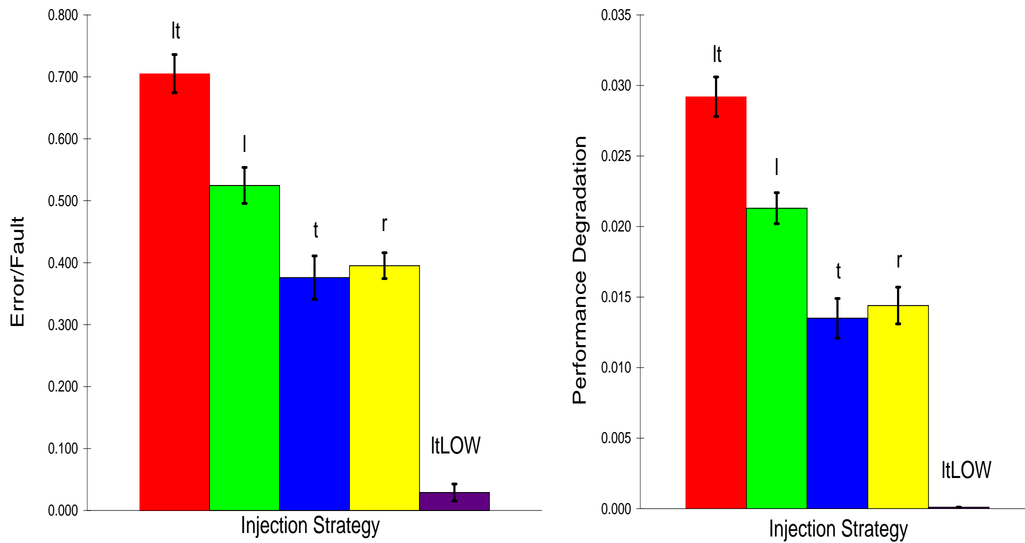

Figure 4: Error/Fault and Performance Degradation (with 95% confidence intervals)

The error/fault ratio and performance degradation for the five injection strategies used with the same workload are given in Figure 4, along with vertical bars that indicate the 95% confidence intervals. The figure shows averages based on 19 runs for each injection strategy. The workload used is a disk-intensive workload. From the figure, it can be seen that the highest level of fault tolerance activity (as measured by the error/fault ratio and performance degradation) is obtained when using both the location-based and time-based injection strategies (labeled in the graph as "lt"). If only the location-based strategy (labeled as "l") is used, then the fault tolerance activity is lower. However, the location-based strategy

still produces more activity than using the time-based or random strategies (labeled as "t" and "r", respectively). Thus, for this disk-intensive workload, injecting faults into the disk produces more fault tolerance activity than choosing the injection location randomly. However, if additionally the faults are injected only when the dynamic workload activity is high, then the effect is even greater.

The measured performance degradation in Figure 4 is small, partly because it is divided by the number of faults injected. However, the measure is significant because it is intended to be used as a relative measure. Thus, the importance of the measure is that the combined location-base and time-based injection strategy produces more performance degradation than the other strategies.

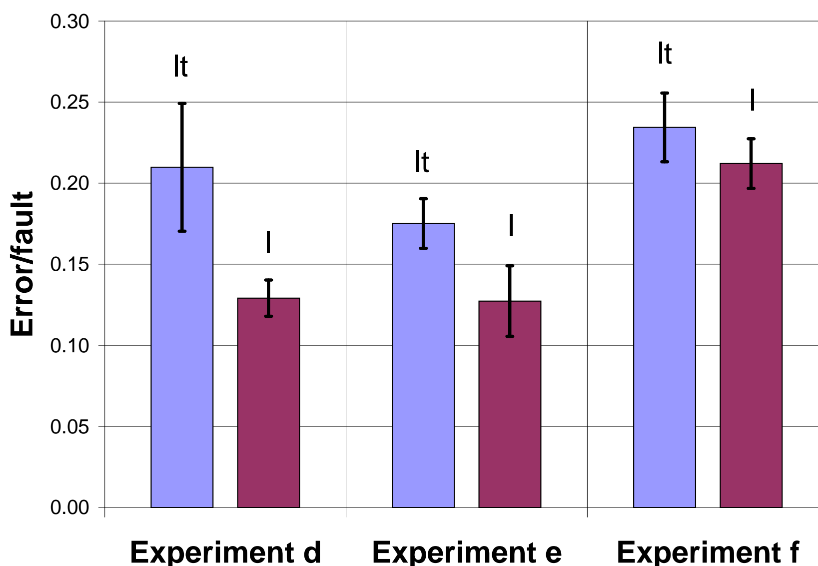

Figure 5: Error/Fault for Several Workloads

This effect can be seen for other workloads. Figure 5 shows the error/fault ratio for several workloads: workload (d) is CPU-intensive, workload (e) is mixed, and workload (f) is somewhat disk-intensive. For each workload, the error/fault ratio is higher when the location-based strategy is combined with the time-based strategy. Again, the combined strategy is labeled as "lt" in the graph, while the location-based strategy alone is labeled as "l".

As shown in Table 1, disk faults have a much higher error/fault ratio and performance degradation compared to CPU and memory faults. To ensure that the results of the experiments are not biased by this, the results were also calculated for the same experiments in Figure 5, but ignoring disk faults. The results are given in Table 2. Again, the error/fault

Table 2: Stress-based Injection Results For CPU and Memory Faults

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

ratio is highest when the location-based and time-based injection strategies (labeled as "lt") are combined for experiments d and e. For experiment f, the error/fault ratio for the lt and random strategies are similar because the workload is disk-intensive but CPU and memory faults are injected. The error/fault ratios for the lt strategies are highlighted in the table. For the error/fault ratio, 95% confidence intervals are given.

System Crash Data The results above do not include experiments that resulted in system crashes. The number of system crashes is given in Table 3 classified by the injection strategy and workload used. Each row represents a different workload, and each column represents the injection strategy used. For example, the "lt" column of row "e" shows that 3 system crashes occurred while the combined location-based and time-based injection strategy was used with workload (e). The table includes data for 212 runs, during which a total of 4 system crashes occurred. All crashes occurred when the location-based and time-based injection strategies were used. This result is consistent with the results of the other experiments in this paper, which show that the combined location-based and time-based strategy seems to produce the most fault tolerance activity.

Table 3: Number of Observed System Crashes

Experiment	Injection strategies lt	l	t	r	ltLOW
d	0	0	0	0	0
e	3	0	0	0	0
f	1	0	0	0	0
Total	4	0	0	0	0

4 Path-based Injection

Recall that path-based injection (PBI) provides a focused view of system dependability for a target application. Two major tasks are needed to implement path-based injection:

1. pre-injection analysis, which associates paths with faults, and

2. injection of the fault, including selection of an appropriate input.

These two tasks are described in the following subsections.

4.1 Pre-injection Analysis

As mentioned previously, a pre-injection analysis must be performed to associate paths with faults. This task of associating paths with faults is the key to path-based injection. A high fault activation level is only possible because the faults are injected in conjunction with an input that is guaranteed to activate that fault.

To associate an input with a path, path-based fault injection employs techniques similar to software branch testing. Branch testing involves (1) generating a set of paths that will cover every branch in the program and (2) finding a set of test inputs that will execute every path in this set of program paths, e.g., [23], [24], [25].

The implementation was performed on a Tandem Integrity S2 fault-tolerant computer, which is based on the MIPS R2000 microprocessor. The test program is `compress`, the standard UNIX compress utility and is written in the C language. The description of the implementation in this section refers specifically to the S2 and `compress` to present a concrete example. However, the same procedure can be performed using any computer system and any test program.

The following steps are needed to accomplish this task:

1. Derive an input set based upon knowledge of the test program, including command-line options, documentation, and knowledge of the program's high-level language code.

2. For each input in the input set, determine the associated path. The path is represented as a list of test program basic blocks that are executed by the associated input. A basic block is a sequential group of instructions such that every instruction in the basic block is executed if the first instruction is executed.

3. Determine the faults that can be activated by each path.

The analysis of the target software program is performed at the assembly level for several reasons. First, there is no dependence on any single high-level language. Second, compiler optimizations such as the deletion, duplication, or reordering of code do not have to be considered. Third, analysis at the assembly level permits direct access to physical register names without any need to map variables to physical registers.

Deriving an Input Set The first step, derivation of the input set, is performed manually and is not included in the fixed cost, C_F. This cost was not included in C_F because (1) timing a non-automated activity is not simple, (2) the cost is heavily dependent upon the skills of the person creating the input set, and (3) most importantly, the cost only adds to the fixed cost, which might increase F_R in Figure 6 but does not change the fact that the time-cost for path-based injection is lower than that for random injection after F_R faults are activated, where F_R is the minimum number of faults at which path-based injection incurs a lower cost than random injection.

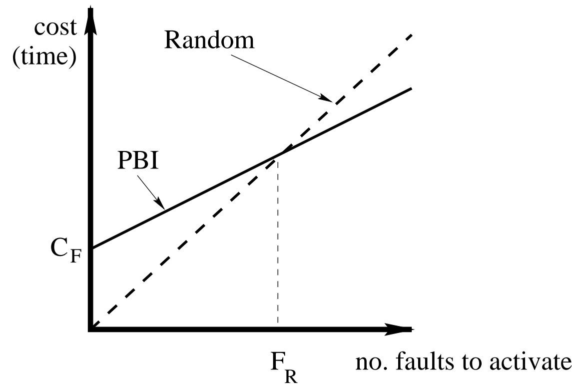

Figure 6: Time cost vs. # faults to activate

Determining the Path Associated with an Input The second and third steps are automated, and their costs constitute the fixed cost, C_F. The second step is the most time-consuming in the pre-injection analysis. This step involves the discovery of the path associated with each input in the input set. This determination of paths can be performed

with a utility such as `pixie` [7], which is available for MIPS processors, or with a custom-built utility that explicitly inserts additional code that is called at the start of each node in the path. The paths are described in terms of a list of basic blocks that are executed due to a given input set.

Determining Faults Associated with a Path The final step in the pre-injection analysis is to determine the faults that can be activated by each path. To simplify this step, we will only consider control-flow faults or faults that directly affect the execution of branches and jumps. For instance, an example of a control-flow fault would be a CPU register that is corrupted and causes a conditional branch to be evaluated incorrectly, thus altering the control-flow of the program. With this in mind, all direct-effect control-flow faults occur when branches or jumps occur. To simplify things further, we will only inject faults into CPU registers, which means that all such control-flow faults occur at conditional branch instructions. Thus, the faults that are activatable by each path occur when the CPU registers used as operands for conditional branches are corrupted.

Although only control-flow faults are treated in this paper, other types of faults can also be addressed. In the case of data-flow faults, data-flow paths, instead of control-flow paths, would need to be enumerated and associated with faults. The remainder of the methodology would be similar.

Once the pre-injection analysis is finished, path-based fault injection can be performed. The next section describes the method used to inject faults.

4.2 Injection of Fault

The purpose of the fault injection implementation described in this section is to provide a testbed for demonstrating the ability of path-based injection to maximize the fault activation level, especially when compared to random injection. The fault injection method used is *software-implemented fault injection*, which uses software routines to modify system state (such as that contained in registers and memory locations) to emulate the effect of lower-level faults. Although software-implemented fault injection can be used to inject a variety of faults, we will only inject faults into CPU registers, since that is sufficient to accomplish our goal of demonstrating the advantages of path-based injection.

The fault injection testbed is shown in Figure 7. The *target machine* is the machine on which the test program is run and faults are injected. A *DAS* logic analyzer is connected to

[7] The `pixie` utility takes as input an executable file and inserts additional code to monitor the execution of all basic blocks. When the instrumented program is executed, an output file is created which contains information about basic block execution and register usage.

the target machine and monitors memory bus activity to detect the activation of injected faults. The DAS needs to be reprogrammed before each fault injection to search for the activation of that specific fault. The reprogramming is controlled by the *control host*, which in turn communicates with the injection software on the target machine to know what fault is to be injected.

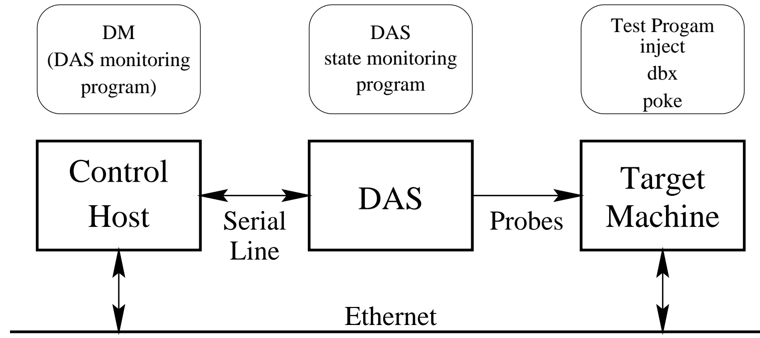

Figure 7: Setup of DAS, Host, and Target Machine

The injection software on the target machine consists of three programs:

1. `inject`,
2. `dbx`, and
3. `poke`.

The flow chart for these three programs is given in Figure 8. `inject` is the main control program. It first initializes the DAS with the needed monitoring software and then reads in the pre-injection analysis information, which shows which inputs and faults should be paired up to ensure activation of the fault. `inject` selects a fault to be injected and an input that will result in the activation of that fault, and then the fault is injected. For random injection, the input is selected randomly, which may not result in the activation of the fault. If no activation occurs, then another input is chosen for the same fault, and the fault is injected again. This process is repeated until either the fault is activated or an arbitrary threshold is reached. In our experiments, the inputs for random injections were not chosen completely randomly; instead, the inputs were chosen randomly from the same set of inputs used for path-based injection. By so doing, the fault activation level is increased beyond that for a completely random selection of inputs. Yet, the results in Section 4.3 will show that even with this assistance, the fault activation level is still much lower for random injections.

`inject` does not perform the actual fault injection. Instead, it calls the `dbx` program. `dbx` is a fairly common UNIX C-language debugger. `dbx` is used to control the timing of the fault injection. This is accomplished by setting a breakpoint at the stop address, which is

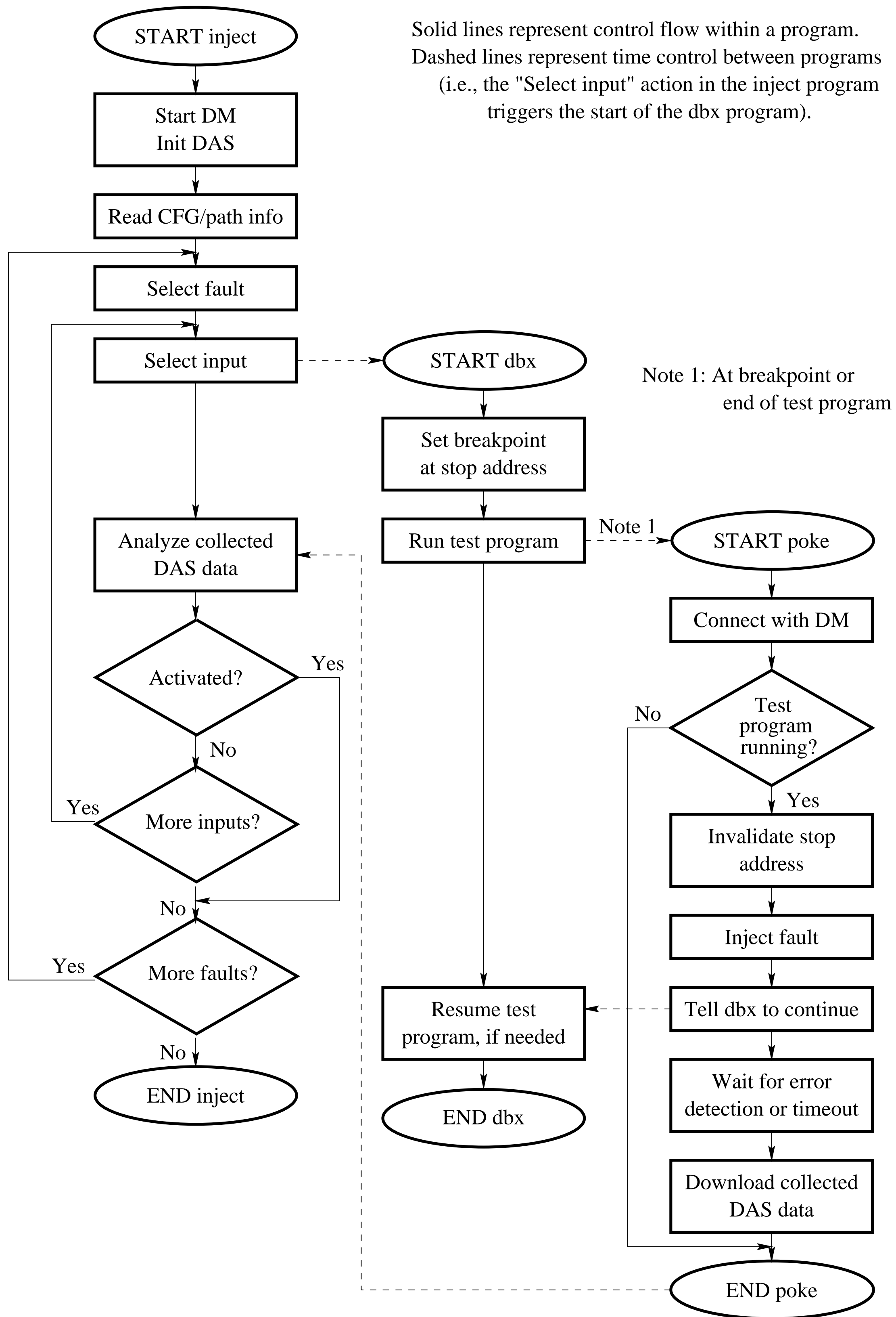

Figure 8: Flow Chart for Injection Instrumentation

the current test program address when the fault should be injected. `dbx` then runs the test program. If the breakpoint is encountered, the `poke` program is executed. `poke` injects the actual fault and is responsible for communicating with the control host machine to reprogram the DAS and to obtain the data collected by the DAS.

The data collected by the DAS is of the form shown in Figure 9. Each line represents an event that the DAS has detected. The leftmost number of each line is the event identifier. The meanings of the event identifiers are given in Table 4. The presence of event identifier 2 indicates that the fault has been activated. This information is sent by `poke` to `inject` to determine if the fault has been activated. Event identifier 7 indicates that that the built-in error detection mechanisms of the system have detected an error. These events correspond to memory addresses in the code that handle the specific events. For example, memory address `0x1FC00000` is the first instruction in the error handling routine of the operating system. Because the injected address is always monitored, any overwriting of the memory location will also be detected. The DAS will continue to monitor events after the fault is overwritten, but these events will be disregarded.

```
State  ADR     DATA    W/R EBI DMA I543~    time
#___________________________________________________________
#ID: 61/0  PID: 24814  Vloc: 0x0040170c  Ploc: 0x003ae70c  Reg#11  mask: 0x00000040
 0  1FCA000C  1960000D  01  111  ... 11111    0.89642522
 1  003AE70C  1960000D  01  111  ... 11111    0.00000630
 2  003AE70C  1960000D  11  111  ... 11111    0.07827434
 2  003AE70C  1960000D  11  111  ... 11111    0.08470108
 7  1FC00000  0BF00082  10  011  ... 10111    0.20634250
```

Figure 9: Sample collected DAS data

Table 4: DAS Event Identifier Meanings

Identifier	Meaning
0	Fault injection routine has been entered
1	Actual fault has been injected
2	Fault has been activated
7	Error has been detected

The results from the experiments conducted with the fault path-based and random injection testbed described above are presented in the next section.

4.3 Results

The path-based injection instrumentation described in the previous section was implemented on a Tandem Integrity S2 computer, which is described in [21]. Since path-based injection is mostly concerned with investigating the properties of the application, the main TMR-based error detection of the S2 was bypassed by injecting faults into all three processor units. This caused the system to act in a manner similar to a uniprocessor system, which meant that errors were detected mainly via operating system exceptions. Even though the S2's fault tolerance capabilities were not needed, it was still used as the experimental platform because the instrumentation needed to measure the results in Section 3.4 had already been developed for previous experiments. Three applications were tested using path-based injection:

`compress` Compress is one of the programs in the SPEC integer benchmark suite. It accepts input from a file or from the standard input and reduces the overall size of the input using adaptive Lempel-Ziv coding. The output can be written to a file or to the standard output.

`dc` `dc` is a stacking (reverse Polish) calculator. Ordinarily it operates on decimal integers, but one may specify an input base, output base, and number of fractional digits to be maintained. If an argument is given as a filename in the command, input is taken from that file until its end, then from the standard input. An end-of-file on standard input or the 'q' operator stops `dc`. The calculation result is shown on the standard out if 'f' is used.

`WPI` The WPI Benchmark Suite is a synthetic database benchmark. The system is composed of a concurrent database server, numerous client processes, a database generation program, a large database file, and analyzer program for the client and server. The database is a mailing list with records holding the address information of a large number of people. The server manages the database and continually services requests made by clients. The services include: get last record, read a record, write a record, and append a new record to the database.

For each of these three programs, an input set was chosen to achieve a high coverage of user code. User code includes all code written by the user. Typically, this includes all code except for standard library routines and extra code inserted by the linker. The focus of the experiments was on user code because (1) user code is exercised more than library routines, (2) access to the source code for user code is greater than for library code, and (3) user code is usually newly created code, whereas library code is reused code. The code coverage for

Table 5: Code Coverage of the Selected Input Set

	# total user code basic blocks	# covered user code basic blocks	% covered user code basic blocks
compress	497	289	58.15%
dc	992	647	65.22%
WPI server	608	447	73.52%

the input sets chosen for the three programs is given in Table 5. Here code coverage refers to the percentage of basic blocks that were executed by at least one input in the input set.

To illustrate the injection of a specific fault, consider the following portion of C-language code from the `compress` program:

```
Line 1: for (fileptr = filelist; *fileptr; fileptr++) {
Line 2:     exit_stat = 0;
Line 3:     if (do_decomp != 0) {                          /* DECOMPRESSION */
Line 4:         /* Check for .Z suffix */
Line 5:         if (strcmp(*fileptr + strlen(*fileptr) - 2, ".Z") != 0) {
Line 6:             /* No .Z: tack one on */
```

This code above checks if all files specified on the command-line ends with a `.Z` and appends a `.Z` if not already present at the end of the filename. This check is only performed if decompression is specified. Line 5 corresponds to the stop address of the fault, or the currently executing instruction when the fault is injected. The conditional branch corresponding to line 5 is

```
beq     s2,zero,0x4006d4
```

which branches to location `0x4006d4` if register `s2` equals zero. The control-flow graph for the six-line portion of code given above is shown in Figure 10, where the boxes are basic blocks and the numbers inside the boxes are the basic block numbers. The `beq` instruction corresponding to line 5 is located in basic block #77. If the contents of register `s2` equal zero, the next basic block is #1402[8]; otherwise, the next basic block is #78. If the path for an input in the input set contains basic block #77, then that input is capable of activating a fault injected into a resource used by basic block #77.

Consider the contrasting effects of path-based and random injection for a fault injected into register `s2` at the `beq` instruction in basic block #77. For path-based injection, an

[8] Although the `compress` program has only 497 basic blocks, many more blocks are added by library code that is linked with the user code. The focus of the fault injection is on the user code, but the library code is also necessarily involved in the control-flow of the program. Thus, this example contains basic blocks from both user code and library code.

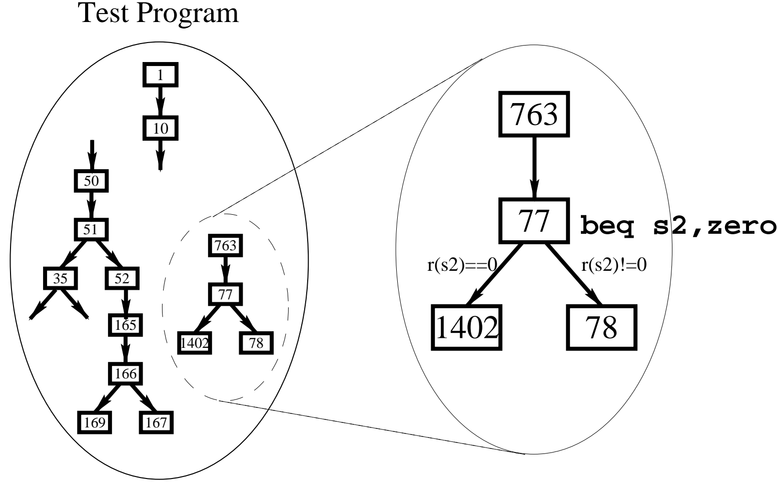

Figure 10: Part of Control-Flow Graph for `compress`

input is selected that executes basic block #77 and thus activates the fault by accessing the corrupted contents of register `s2`. If register `s2` originally contained a `0` and the fault flipped a bit and changed the value (for example, to a `1`), then the control-flow would be incorrectly altered to execute basic block #78 instead of #1402. This incorrect control-flow would add an extra `.Z` to the filename, which would eventually result in an error detection.

Table 6: Fault Effects

Fault Effect	compress		dc		WPI	
Wrong output	39	(26.2%)	67	(22.3%)	19	(31.7%)
Segmentation fault	16	(10.7%)	50	(16.6%)	7	(11.7%)
Bus error	1	(0.7%)	8	(2.7%)	0	
Program hang	2	(1.3%)	3	(1.0%)	11	(18.3%)
Bad system call	0		1	(0.3%)	0	
IOT trap	0		1	(0.3%)	0	
Broken pipe	0		0		1	(1.6%)
No effect	91	(61.1%)	171	(56.8%)	22	(36.7%)
Total	149		301		60	

A major motivation for performing fault injection is to examine how a system reacts to fault conditions. We utilized the path-based injection strategy to investigate how the Tandem S2 responds to control-flow faults for several programs. Table 6 lists the final results for all faults injected for the `compress`, `dc`, and WPI programs. The number of occurrences for each fault effect is given along with the percentage of each fault effect out of the total number of injections for that program.

A number of interesting observations can be drawn from Table 6. First, 30%-45% of the undetected faults caused an incorrect output to be produced. The number of undetected faults is the sum of the faults that caused a wrong output or had no effect on the output. The number of undetected wrong outputs is high for all three programs, which suggests that additional error detection mechanisms may be needed.[9] Second, for all three programs, at least 10% of the faults resulted in a segmentation fault. This is desirable because the operating system detects the error, which relieves some of the burden on the application to perform error detection. Third, for the WPI program, 18% of the faults caused the program to hang indefinitely. The WPI is especially susceptible to hangs because it is a client-server database program. Additional watchdog or time-out mechanisms are needed to address this problem. Overall, path-based injection uncovered more problems in the WPI program than the other two programs, which is understandable because (1) the `compress` and `dc` programs are standard UNIX programs that have been tested for a much longer operational period of time and (2) the WPI program must deal with communication and interactions among multiple processes.

5 Discussion

5.1 Applicability

Stress-based and path-based injection are complementary fault injection methodologies that focus on different aspects of the system under test. Stress-based injection determines a fault injection strategy based on measurement of system hardware resources, while path-based injection relies on an analysis of the application software. Thus, stress-based injection tests the effect of faults on the computer system, and path-based injection tests the execution of the application program in the presence of faults.

Stress-based injection determines fault injection parameters by monitoring the system workload activity as the test program executes. The workload activity is measured for the entire system including the processor, memory, and I/O components, and faults are injected throughout the entire system based on those workload activity measurements. Thus, the selected faults are able to test the entire system. Because the focus is on testing an entire system, stress-based injection is geared toward testing during the prototype phase of system development.

Path-based injection relies on a pre-injection analysis of the test program to find appropriate fault parameters. The pre-injection analysis examines the structure and resource

[9] It should be emphasized that these results are for experiments on the fault-tolerant S2, but with the fault tolerance bypassed.

usage of the test program in conjunction with the test input set to determine the set of faults that are guaranteed to be activated by that input set. Thus, in contrast to stress-based injection, path-based injection focuses on the test program and how the system reacts to faults that affect that test program. The effect of faults on the test program must be determined to produce a reliable system. Thus, path-based injection is well suited for use during the development of the test program.

5.2 Fault Activation, Effect, and Detection

In terms of the activation rate for the set of injected faults, path-based injection guarantees activation for all faults. Stress-based injection does not provide this guarantee for individual faults. Instead, activation of faults is promoted by injecting faults at the locations and times that they will likely be activated. Thus, a high level of activation is still obtained, as seen by the amount of resultant fault tolerance activity.

An essential step in fault-injection based testing is the analysis of the fault effects. Were the resulting errors detected, and how was detection accomplished? How effective was the recovery procedure, and in what instances did it fail? Which parts of the system need to be improved? Path-based injection facilitates this analysis because the control-flow and resource usage information for each fault is already available from the pre-injection analysis. This information aids in determining the propagation of the effects of the injected faults, and thus helps to pinpoint the areas in the test program that require improvement. This type of detailed analysis is more difficult with stress-based injection because the information gathered during monitoring of the workload activity is at the system level. The propagation of the fault effects can be related to high-level workload parameters, but an analysis with exact details of the hardware and software require additional run-time monitoring facilities.

Table 7: Random Injections vs. Path-based Injections

	Random	Path-based
Faults injected	623	149
Faults activated	149	149
Injections/activation	4.18	1.00
Fault activation level	**0.24**	**1.00**

Path-based injection does indeed result in a higher fault activation level than random injection. The path-based approach uses knowledge of the relationships between paths and faults to pick a fault that will guarantee activation of that fault. Random injection randomly

associates a fault with a path, and therefore will result in a lower fault activation rate. Table 7 shows that the fault activation level for path-based injection is four times higher than for random injection based upon injections for the `compress` program. For both types of injection, each fault was injected into the program until the fault was activated. If a fault was not activated, that fault was injected once more as the program was executed again.[10] For path-based injection, the path-based injection strategy was used to select program inputs, while program inputs were selected randomly in the random case.

For path-based injection, all injected faults were activated, which is to be expected, since that is the main purpose of path-based injection. With random injection, an average of over four injections were needed to activate each fault. Thus, the fault activation level is less than one-quarter. Note that even this low level of fault activation is achieved with some assistance because the test program inputs for random injection were selected from the path-based injection input set. In reality, the path-based injection input set would not be available, and thus the true fault activation level for random injection would be even lower than that reported in Table 7. In addition, for the random injections in our experiment, fault locations are restricted to the same set of fault locations for path-based injection, which eliminates the injection of faults into resources that are never used. This further inflates the fault activation level for random injection in Table 7.

The error detection rate is also increased by both stress-based and path-based injection. Table 8 shows the error detection rates for the three experiments in Table 2. For all three experiments, stress-based injection resulted in more detection of errors than random injection. Path-based injection also results in more error detection than random injection. Table 7 shows that random injection requires many more injections than path-based injection because many injected faults are not activated. Since non-activated faults will obviously not be detected, the corresponding error detection rate will be lower for random injection.

Table 8: Comparison of Error Detection Rates for Random and Stress-based Injection

Experiment	Random	Stress-based
d	11.7%	17.5%
e	10.5%	16.8%
f	11.5%	11.8%

Two types of important measurements can be obtained from path-based injection: (1) the percentage of faults that resulted in an error detection and (2) the percentage of faults that

[10] Subsequent injections of the same fault may produce different outcomes because the fault location and type are fixed, but the timing relative to the program is not.

resulted in an incorrect output with no detection. Table 9 shows these two measurements for the three test programs in Section 4.3. The percentage of faults that causes an incorrect output with no detection is quite high, 26%-32%. For situations requiring high reliability, additional fault tolerance, either in the form of software or hardware mechanisms should be considered.

Table 9: Error Detection Rates for Path-based Injection

Program	Detection	Undetected wrong output
`compress`	12.4%	26.2%
`dc`	20.9%	22.3%
WPI	31.6%	31.8%

5.3 Cost

Stress-based injection is easier to implement than path-based injection. Because the injection mechanisms for both methodologies are similar, the difference lies in the information gathering mechanisms, i.e., the workload activity monitor for stress-based injection and the pre-injection analysis tool for path-based injection. For stress-based injection, the workload activity monitoring takes advantage of monitoring facilities already present in most operating systems, which greatly alleviates that task of porting. The path-based injection pre-injection analysis tool requires knowledge of the machine instruction set and object code format, which are both very much system-dependent. In terms of the time overhead incurred during the actual fault injection testing process, stress-based injection requires the workload activity monitor to operate on the system under test. The resultant time overhead is slight because the measurement period is on the order of a few seconds, which is much longer than the time allotted to the test program. The time overhead for path-based injection is greater than that for stress-based injection due to the need for a pre-injection analysis. However, this additional time for the pre-injection analysis is well spent because it maximizes the fault activation rate.

The desirability of a high fault activation level can be illustrated with Figure 6. The graph shows the cost in time incurred to activate a certain number of faults that are injected. The costs associated with path-based injection are represented by the solid line, while the dashed line represents the random injection costs. The slope of each line is determined by the cost to activate one fault, which is equivalent to $(\text{fault activation level})^{-1}$. Obviously, the time-cost increases as more faults are to be activated. However, the time-cost for path-based injection

increases less rapidly than that for random injection because path-based injection maximizes the fault activation level and thus minimizes the cost to activate each fault.

Since path-based injection requires a pre-injection analysis, a one-time cost is incurred. This cost is represented by C_F in Figure 6. C_F is dependent on the number of paths to be analyzed. However, once the number of paths to be analyzed is determined, C_F is fixed and thus is referred to as the *fixed cost*. This fixed cost initially increases the time-cost of the path-based injection approach. However, as more faults are activated, the fixed cost is amortized and eventually at some number of faults, F_R, the total time-cost of path-based injection is lower than that for random injections. This fact is true regardless of the fixed cost of the pre-injection analysis. In our experiment, F_R turns out to be a fairly low number for at least one fault injection testbed. This is shown in Section 4.3.

Note that although the two lines in Figure 6 are straight lines, the actual time-cost for each activated fault will vary. Especially for the random injection approach, the time-cost for each activated fault is very dependent upon the number of injections needed to activate that fault, which may be as low as a single injection or as high as the surrender threshold. Since the time-cost per activation can never be lower for random injection, the two lines in Figure 6 are still accurate in a relative sense.

Table 10: Measured Values for Cost Graph (see Figure 6)

Meaning	Value
Fixed time-cost of pre-injection analysis (C_F)	1357s
Startup time-cost of injections	9s
PBI time-cost per-activated fault	51s
Random time-cost per-activated fault	213s
Min. # faults to justify PBI (F_R)	8.4

In addition to verifying that path-based injection does indeed result in a higher fault activation level, the experiments performed also allow the fixed cost (C_F in Figure 6) and the per-activated fault time-cost for path-based and random injection to be measured. With these measurements, the minimum number of faults needed to justify path-based injection (F_R) can be calculated. These values for these parameters are given in Table 10. The fixed cost of the pre-injection analysis (C_F) is 1357 seconds. This value does not include the time required to manually determine an appropriate input set, but that time was small relative to the measured C_F[11] The average time-cost associated with each fault before activation

[11]For this example, the time to determine the input set was approximately five minutes, because the test programs were relatively small and well known. Larger, more complex programs will obviously require more

occurred was found to be 51 seconds for path-based injection and 213 seconds for random injection. Based on these measurements, F_R is calculated to be 8.4 faults. F_R is the minimum number of faults that have to be injected before path-based injection incurs a lower time-cost than random injection. Thus, for this particular test environment, path-based injection is justified from a time-cost perspective if at least 9 faults are to be activated.

6 Conclusion

This paper presents two methodologies for selecting fault parameters that result in a high level of fault tolerance activity. Stress-based injection uses a run-time workload activity monitor to select faults that coincide with the locations and times of greatest workload activity. Path-based injection depends on a pre-injection analysis of the software program control flow and resource usage to find faults that are guaranteed to be activated for a given input set. The experiments that have been conducted with implementation of both methodologies show that the level of fault tolerance activity is indeed increased compared to random fault selection.

Both stress-based and path-based injection guide the process of fault injection by selecting faults that test the dependability of a system better than a random selection of faults. For both injection methodologies, experiments have been performed to demonstrate the advantage over random injection. Stress-based injection has been shown to produce more error detection and more performance degradation than random injection, and path-based injection has been shown to produce a higher level of fault activation than random injection. These benefits are direct results of the respective injection methodologies.

The important characteristics of the stress-based and path-based methodologies are summarized in Table 11. The main differences are due to the difference between the methods of analyzing the workload activity.

Acknowledgments

This research was supported in part by the National Science Foundation grant 96-01631 and NASA grant NAG 1-613, in cooperation with the Illinois Computer Laboratory for Aerospace Systems and Software (ICLASS). The content of this paper does not necessarily reflect the position or policy of these agencies, and no endorsement should be inferred.

time and thus increase C_F, which will in turn cause F_R to increase. However, for these larger programs, a higher F_R is acceptable because it is to be expected that more faults are injected to cover the increased code size.

Table 11: Summary of Stress-based and Path-based Characteristics

Characteristic	Stress-based	Path-based
Analysis	Run-time	Pre-run-time
Analysis tool	Workload monitor	Program analyzer for control and data flow
Pre-injection analysis	No	Yes
Implementation complexity	Low	High
Knowledge required	Architecture level	Architecture level
Fault activation rate	High	Very high
Error detection level	High	High
Applicability	System prototype phase	Software development phase

References

[1] Ram Chillarege and Nicholas S. Bowen. Understanding large system failures – a fault injection experiment. In *Proceedings 19th International Symposium on Fault-Tolerant Computing (FTCS-19)*, pages 356–363, June 1989.

[2] Jean Arlat, Martine Aguera, Louis Amat, Yves Crouzet, Jean-Charles Fabre, Jean-Claude Laprie, Eliane Martins, and David Powell. Fault injection for dependability validation-a methodology and some applications. *IEEE Transactions on Software Engineering*, 16(2):166–182, February 1990.

[3] Ravi Iyer and Dong Tang. Experimental analysis of computer system dependability. In Dhiraj K. Pradhan, editor, *Fault-Tolerant Computer System Design*, chapter 5. Prentice Hall PTR, 1996.

[4] Dimitri R. Avresky, Jean Arlat, Jean-Claude Laprie, and Yves Crouzet. Fault injection for the formal testing of fault tolerance. In *Proceedings 22nd International Symposium on Fault-Tolerant Computing (FTCS-22)*, pages 345–354, Boston, Massachusetts, June 1992.

[5] Jean Arlat, Alain Costes, Yves Crouzet, Jean-Claude Laprie, and David Powell. Fault injection and dependability evaluation of fault-tolerant systems. *IEEE Transactions on Computers*, 42(8):913–923, August 1993.

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

[7] Eric Jenn, Jean Arlat, Marcus Rimen, Joakim Ohlsson, and Johan Karlsson. Fault injection into VHDL models: The MEFISTO tool. In *24th International Symposium on Fault-Tolerant Computing (FTCS-24)*, 1994.

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

[9] James H. Barton et al. Fault injection experiments using FIAT. *IEEE Transactions on Computers*, 39(4):575–582, April 1990.

[10] Ghani A. Kanawati, Nasser A. Kanawati, and Jacob A. Abraham. FERRARI: A tool for the validation of system dependability properties. In *Proceedings 22nd International Symposium on Fault-Tolerant Computing (FTCS-22)*, pages 336–344, Boston, Massachusetts, July 1992.

[11] Wei-Lun Kao and Ravi Iyer. DEFINE: A distributed fault injection and monitoring environment. In Dhiraj K. Pradhan and Dimitri R. Avresky, editors, *Fault-Tolerant Parallel and Distributed Systems*, pages 252–259. IEEE CS Press, Los Alamitos, California, USA, 1995.

[12] S. Han, K. G. Shin, and H. A. Rosenberg. DOCTOR: An integrateD sOftware fault injeCTiOn environment for distributed Real-time systems. In *International Computer Performance and Dependability Symposium*, pages 204–213, April 1995.

[13] J. Carreira, H. Madeira, and J. G. Silva. Xception: Software fault injection and monitoring in processor functional units. In *Proceedings 5th International Working Conference on Dependable Computing for Critical Applications (DCCA-5)*, pages 135–149, Urbana, IL, September 1995.

[14] K. Echtle and Y. Chen. Evaluation of deterministic fault injection for fault tolerant protocol testing. In *Proceedings 21st International Symposium on Fault-Tolerant Computing (FTCS-21)*, pages 418–425, Montreal, Canada, June 1991.

[15] J. Christmansson and P. Santhaman. Error injection aimed at fault removal in fault tolerance mechanisms. In *Proceedings 7th International Symposium on Software Reliability Engineering (ISSRE'96)*, pages 175–184, White Plains, New York, October 1996.

[16] Wei-Lun Kao. Experimental study of software dependability. Technical Report CRHC-94-16, Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1994. Ph.D. thesis.

[17] X. Castillo and Daniel P. Siewiorek. Workload, performance, and reliability of digital computer systems. In *Proceedings 11th International Symposium on Fault-Tolerant Computing (FTCS-11)*, pages 84–89, Portland, Maine, June 1981.

[18] J. Guthoff and V. Sieh. Combining software-implemented and simulation-based fault injection into a single fault injection method. In *Proceedings 25th International Symposium on Fault-Tolerant Computing (FTCS-25)*, pages 196–206, Pasadena, California, June 1995.

[19] Jean-Claude Laprie. Dependable computing and fault tolerance: Concepts and terminology. In *Proceedings 15th International Symposium on Fault-Tolerant Computing (FTCS-15)*, pages 2–11, June 1985.

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

[21] Doug Jewett. Integrity S2: A fault-tolerant Unix platform. In *Proceedings 21st International Symposium on Fault-Tolerant Computing (FTCS-21)*, pages 512–519, Montreal, Canada, June 1991.

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

[23] Antonia Bertiolino and Martina Marr'e. Automatic generation of path covers based on the control flow analysis of computer programs. *IEEE Transactions on Software Engineering*, 20(12):885–899, December 1992.

[24] David Hedley and Michael A. Hennel. The cause and effects infeasible paths in computer programs. In *Proceedings 8th International Conference on Software Engineering*, pages 259–266, London, UK, August 1985.

[25] Elaine Weyuker. Translatability and decidability questions for restricted classes of program schemas. *SIAM Journal on Computing*, 8(4):587–598, 1979.