Automated Fault-Inject Based Dependability Analysis of Distributed Computer Systems *

David Thomas Stott
Center for Reliable and High-Performance Computing
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1308 West Main Street
Urbana, IL 61801-2307
(217) 244-6974
dstott@crhc.uiuc.edu

Abstract

Recently, there has been interest in developing a dependability benchmarks for computer systems. This will require a way to inject several different types of faults into many different platforms and a way to collect and compare the results. Analyzing complex heterogeneous distributed systems share the same needs. The current approach to building fault injection tool is inappropriate for these goals.

NFTAPE (Networked Fault Tolerance and Performance Evaluator), a tool for designing and executing fault injection campaigns, make these goals possible by using lightweight fault injectors (LWFI). NFTAPE provides (1) an environment for executing fault injection campaigns using a library of fault injection programs (such as LWFI, triggers, monitors, and so on), (2) support services for communicating between test nodes, managing processes, and logging results, and (3) a programming model and API for writing support processes including exist fault injectors.

Unlike older fault injectors, NFTAPE can use any fault injection method (software implemented fault injection, physical fault injection, or simulation). A few new variations on fault injections methods have come out developing NFTAPE so far: driver-based fault injector, target-specific fault injectors and performance faults. Already, NFTAPE has been used to run fault injection campaigns on several platforms (Linux, Solaris, LynxOS, and Windows) using several fault injection methods (debugger-based, driver-based, target-specific, and special-purpose hardware-based).

1 Introduction

Critical computer systems, such as systems controlling medical equipment, power systems, fly-by-wire aircrafts, or business-critical services, are becoming more common. The well-being of the environment, businesses and most importantly human life depend on such systems operating without failure. Although critical systems must operate correctly despite operational faults (such as memory upsets due to sub-atomic particles), how to assess the behavior of these systems in the presents of such faults has received little attention.

Simply waiting to observe fault behavior in the field is inadequate because such events naturally happen too infrequently. Instead, synthetic faults must be introduced (or injected) synthetic faults to emulate the real ones

---

*This paper was originally submitted as part of the requirements for the Preliminary Exam and revised 3/13/00. Copyright 2000, David T. Stott. All rights reserved.
at an accelerated rate to test such systems. Currently, systems are tested by (1) developing a high-level model of possible failure modes (for example, a fault analysis tree), (2) injecting faults into a simulation of the system, or (3) using hardware or software to emulate real faults at an accelerated rate. The first two approaches can not capture implementation issues, and it is usually impractical to develop models with enough detail to accurately model fault propagation. For these reasons, the third approach, fault injection, is generally preferred. Nonetheless, it does have its drawbacks. While there are several tools available for fault injection, and each one is useful at what it does, their abilities are limited. Certain classes of studies we would like to apply fault injection tools, but the available tools are inadequate. For example,

1. **Comparative Studies**: such as, ‘Is my application more reliable on a Unix or Windows platform?’ or ‘Is Sparc-Solaris more reliable than Sparc-x86?’

2. **Dependability Benchmarking**: how can we rank several different systems in terms of dependability?

3. **System-level Evaluation of Large Heterogeneous Systems**: applying traditional fault injection analyses to large systems build on several different platforms.

4. **Special-Purpose Embedded Systems**: some embedded systems are not supported by any fault injection tools and may have memory constrains which prevent a large fault injection tool from being ported to them.

Each of these cases requires injecting several types of faults usually into different architectures. When many types of fault injection are required, it is not practical to use a different tool for each fault type. And, where several platforms are used, the porting effort with several tools is even greater. For most of these experiments, support for distributed systems is also required, but few tools (except for those designed to inject communication faults) support distributed systems. Besides the porting effort, the learning curve for each to configure each and to retrieve logs.

Dependability benchmarking requires a method of evaluating some dependability metric on several different types of systems in order to compare the dependability of the systems. Because all existing tools only operate on a small set of platforms, the cost of porting the tools to each test system could be very high. Furthermore, because no single tool can inject very many different types of faults that will be needed to evaluate different dependability metrics, no single tool is appropriate for measuring all the necessary metrics.

In system-level analysis, the goal to be able to test any piece of the system against as many faults as reasonable to:

1. removed design or implementation faults from the system,
2. validate error handling mechanisms,
3. forecast how often critical systems failures will occur,
4. determine the coverage of specific error detection or recovery mechanisms, or
5. to measure the performance degradation due to faults in the system.

It is common to find a few different architectures in such systems and usually these subsystems (sometimes including embedded subsystem) are not supported by existing fault injection tools. For these reasons, the amount of work to have fault injectors for each platform and fault model has the potential of being very costly. Clearly, porting a different tool for each model is not an acceptable solution.

While designing a large computer systems, Jet Propulsion Laboratory (JPL) proposed a set of requirements they needed to analyze the system. The purpose of this project, part of the Remote Exploration and Experimentation (REE), was to develop a computer system which could execute distributed applications for several different
scientific space-borne missions. Because these applications will run in hostile environments (one mission will fly through the high-radiation field around Jupiter’s moon, Europa) on long running missions the system must be designed to tolerate a large number of faults. The design team created a list of requirements (show in Table 1) that the fault injection tool would need to support to allow them to properly test their design. A survey of existing tools show (1) no tool supported all these types of faults, and (2) any tool would need to be ported to at least one of the platforms used in the system or testbed. This motivated finding a new way to build fault injection tools.

<table>
<thead>
<tr>
<th>Table 1. Fault Injection Requirements for JPL’s REE Project</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Provide multiprocessor injection capability</td>
</tr>
<tr>
<td>• Centrally controlled and logged</td>
</tr>
<tr>
<td>• Register Faults</td>
</tr>
<tr>
<td>• Memory Faults (kernel and application addresses)</td>
</tr>
<tr>
<td>• Time-based and Event-based triggers</td>
</tr>
<tr>
<td>• Asynchronous notification of process completion (with exit status)</td>
</tr>
<tr>
<td>• Dynamically trace application memory usage</td>
</tr>
<tr>
<td>• Provide a mechanism for calling functions within application</td>
</tr>
<tr>
<td>• Detect program hangs</td>
</tr>
<tr>
<td>• Parameterized fault rate</td>
</tr>
<tr>
<td>• Application or fault injection may be globally paused</td>
</tr>
<tr>
<td>• API support synchronization to coerce system into specific states</td>
</tr>
<tr>
<td>• Tasks can be added and removed from injection set</td>
</tr>
<tr>
<td>• Monitoring selected memory contents</td>
</tr>
<tr>
<td>• Network fault injection</td>
</tr>
<tr>
<td>• Saving and comparing selected memory ranges</td>
</tr>
<tr>
<td>• Single-bit and multiple-bit upsets</td>
</tr>
</tbody>
</table>

One of the reasons that no tool supported these features is that every tool is monolithic and self contained. They all have functions for triggering faults, injecting faults, logging results, and communicating (for distributed fault injection), but it is not possible to combine features from one tool into another. Even if two tools were ported to the same system, it would be difficult to do this because each tool is monolithic. For example, it is not possible to use the same logging facility between two arbitrary fault injection tools.

A second difficulty in porting several different fault injectors to one system is that the there is a fair amount of code which performs similar functions but needs to be ported, maintain, and configured separately. Some of these functions are logging, communication, fault triggering, and process management (if it is supported). It is more sensible to have a single tool to handle these functions, than to have every fault injection tool do so.

NFTAPE (Networked Fault Injection and Performance Evaluator) is a portable tool for conducting automated dependability campaigns in distributed computer systems. It includes a framework for running experiments using a set of small replaceable modules for fault injection (called lightweight fault injectors), event triggering, system monitoring, error detection, stress generation, etc. NFTAPE also provides basic services for experiments including communication, global control and synchronization, logging, and process startup and an API to simplify the development process for writing specific modules.

One solution to these difficulties encountered with the current tool set is to have a standard interface for a fault injector and a fault trigger. This way, fault injector developers can concentrate on the component which
just injects faults and system engineers can learn to use a single tool which communicates with the simpler fault injection modules. This is precisely what NFTAPE does. It provides a framework for coordinating fault injection campaigns on distributed systems. NFTAPE also defines an interface so that different fault injectors or triggers can be selected at run-time. Such fault injectors are called lightweight fault injectors (LWFI) because, unlike older fault injectors which included features like triggering, communication, etc., the only function they perform is fault injection. NFTAPE also includes an API library to facilitate writing new LWFIs and fault triggers.

So far, several experiments have been performed using NFTAPE. NFTAPE has been ported to Solaris, Linux, LynxOS, ChorusOS, AIX and Windows on Sparc, Intel, and PPC architectures. Details of these experiments are given later in the paper:

- **MPI Programs**: compared debugger- and driver-based fault injection distributed MPI programs,
- **Next Generation Space Telescope (NGST)**: assessed reliability concerns for “real” space imaging software program,
- **Chameleon/Voltan “SIFT” Comparison**: Evaluated use of self-checking (in Chameleon ARMORS) as an alternative to full replication (in Voltan) for providing fail-silence in software implemented fault tolerance products (SIFT),
- **Performance Faults with Win2K**: Assessed affect of new fault model (performance faults) on Server SAN driver in Win2K platform,
- **LynxOS Kernel Faults**: Observed effects of single-event upsets in kernel’s address space,
- **ChorusOS (Micro-) Kernel Faults**: Demonstrated ability to inject faults into kernel space of embedded system (running ChorusOS), and
- **Myrinet Hardware Fault Injector**: Demonstrated NFTAPE’s ability to use hardware fault injector using special-purpose hardware to inject faults into a Myrinet link.

The remainder of this paper is arranged as follows. Section 2 describes related research in fault injection. Section 3 describes the NFTAPE architecture. Section 4 describes the concept of Lightweight fault injectors and provides some new terminology relating to them. Section 5 describes some of experiments that have been performed using NFTAPE. Section 6 discusses future work including ideas for dependability benchmarking. Section 7 summarizes the work and provides concluding comments. Samples of a trigger and a LWFI using the NFTAPE API are shown in Appendix B. More detailed results from the experiments are given in Appendices C-F.

## 2 Related Research

As suggested in the Introduction, fault injection has been used as the primary method of assessing dependability measures in fault-tolerant computer systems. Several different fault injection approaches have been used (e.g., pin-level injection, software implemented fault injection, simulation). Previous research has focused on developing individual tools for specific systems usually offering only a single fault injection method.

### 2.1 Pin-level Fault Injectors

The earliest fault injectors use pin-level injection. One example is Lala’s fault injection work at Draper Labs on the Fault Tolerant Multi-Processor (FTMP) [Lala83]. At LAAS, Arlat [Arlat90a, Arlat90b] extended these formalisms for fault injection for dependability assessments using another pin-level fault injection tool, MES-SALINE [Arlat89]. This methodology was used to assess the European rail system and the DELTA-4 architecture [Powell91] which provided fail-silent nodes using hardware interlocks at each node interface.
While pin-level injection was popular Laprie [Laprie85] et. al. presented a set of formalisms and definitions for describing faults in a system. One important contribution from the work at LAAS was a formalism which divided a fault injection experiment in four parts: (\( \mathcal{F} \)) the faults set, (\( \mathcal{A} \)) the set of activations, (\( \mathcal{R} \)) the set of readouts, and (\( \mathcal{M} \)) the derived measurements. For the most part, all fault injection experiments use only these components. Chillarege and Bowen extended these concepts in [Chillarege89] by formalizing failure acceleration and applying it to an IBM 370 mainframe. Avresky [Avresky92] proposed a methodology for using fault injection during the design and testing phases to remove faults from the design.

### 2.2 Software Implemented Fault Injectors (SWIFI)

Pin-level injection has the disadvantage of needing to be customized for each system and often causing permanent damage to the system under test. At Carnegie Mellon University, Siewiorek et. al., suggest using software to simulate hardware induced faults. This idea called Software Implemented Fault Injection, or SWIFI. Siewiorek (et. al.) from Carnegie Mellon University introduced one of the first SWIFI tools, FIAT [Segall88, Barton90], a tool that added functions to test trigger conditions and inject faults at compile time. This way, FIAT could trigger a fault on a condition such as the arrival of a message from a particular node.

Soon after FIAT was introduced, Kanawati (et. al.) from University of Texas at Austin created a new SWIFI method which used the Unix `ptrace(2)` system call to set breakpoints in a process and to access its memory and registers. This system call is generally used by debuggers to control the traced process. Their tool, FERRARI [Kanawati92], allowed the user to inject memory and register faults without recompiling the target application running in SunOS.

Dawson, Jahanian, et. al. from the University of Michigan developed Orchestra [Dawson96] and Doctor [Han95] to inject protocol faults into their real-time distributed system. Orchestra tested network protocols by inserting code into the network protocol stack on Mach and Solaris systems to inject faults on certain message arrives. To configure and automate these experiments, the user must write a TCL script. Doctor injected communication and bit-flips to memory and registers in the HARTS real-time OS.

With EFA [Echtle92], a SWIFI tool from University of Dortmund, Germany, they proposed an environment for general fault injection, but it made several assumptions about the target system (e.g., the target must be a distributed application and the fault injector must be triggered by message arrivals). This approach also required the programmer to include fault injection functions into the target program and to enumerate all possible faults that can be injected.

The University of Illinois at Urbana-Champaign created a set of fault injection tools. FINE [Kao93] used traps to allow fault injection into the OS in order to assess the propagation of faults in Unix systems. Kao followed this up with DEFINE [Kao94], an extension to FINE which added distributed support and injected software faults by replacing functions in memory with corrupt machine code compiled from the same source files. This grew into another tool, FTAPE [Tsai96], in which Tsai injected faults and I/O errors into a fault tolerant computer, namely, the Tandem Integrity S2. Using FTape, he found that stress-based triggering lead to a high fault activation rate than path- and time-based triggering.

From the University of Coimbra, Portugal, Carreira presented, Xception [Carreira95, Carreira98], a SWIFI tool which uses the debugging registers of modern CPUs to trigger faults. Xception has been shown to inject register and memory faults on distributed PowerPC system with an extremely low level of intrusion. Another tool from University of Illinois, Loki [Cukier99], can trigger faults when a distributed system is in particular states without explicitly synchronization. Another SWIFI tool, MAFALDA [Rodríguez99] injected faults into a ChorusOS-based system by adding functions to the microkernel; they also noted the importance of having low-level error detection to measure the propagation of the fault.

One advantage of SWIFI is that it is relatively inexpensive to implement because it does not require any special hardware or simulator. It has an advantage over physical fault injectors that you can specify what fault to inject and
inject the same fault more than once. While SWIFI approaches can be intrusive (because the fault injector needs to execute on the CPU), the amount of intrusion is usually low. Although SWIFI can inject many faults that can not be injected in hardware (e.g., corrupting a particular general purpose register), SWIFI faults are limited to those parts of the CPU which can be affected in software. For example, SWIFI can not cause the CPU to put a parity error on the system bus. One study found that one-thirds of the errors induced by random selected device-level faults manifested in ways that could be injected by software [Czeck91].

2.3 Physical Fault Injectors

Another method of injecting faults is to use special-purpose hardware to cause electrical disturbances. At Chalmers, Sweden, Gunneflo and Karlsson [Gunneflo89, Karlsson94] used heavy-ion radiation to inject faults into one of set of replicated CPU boards to evaluate the effect of the radiation faults. Later, in [Karlsson95], Karlsson compared three forms of physical fault injection (heavy ion, pin-level, and EM) in the MARS Architecture. Recently, Samson [Samson98] inserted upsets to transistors in a VLIW chip using a laser system designed for correcting VLIW chips. Of all the fault injection approaches, this method most closely resemble the type of faults that one would expect from a hostile environment such as in space or a control system in a power plant. The hardware approaches are non-intrusive—that is, the system operates exactly the same while faults are being injected (until the fault manifests). But, most hardware approaches are very costly (e.g., the ion injection requires a vacuum chamber). The hardware approaches can also cause permanent damage to the system.

2.4 Simulation-based Fault Injectors

Several simulation-based tools have been used to test fault tolerant designs. In 1990, Goswami introduced DEPEND [Goswami90], a simulation-based environment for modeling systems and faults at the system-level. In [Kalbarczyk99b, Ries97], DEPEND was used to demonstrate a hierarchical method where the behavior at one level can be used at a higher level (e.g., the transistor-level analysis could be used in the at the processor level). Jenn and others introduced MEFISTO [Jenn94] to test faulty circuit behavior by inserting faults into VHDL models. Choi used FOCUS [Choi92] to measure the sensitivity of device to device-level upsets. Clark [Clark93] used REACT to evaluate multiprocessor architectures. Ghosh [Ghosh95], modeled complex real-time, distributed systems using the ADEPT simulator.

Simulation has the advantage that it has the ability to inject any fault model. If source code for a simulation is available, simulation is the least expensive method of fault injection. But, it can be very expensive if no detailed simulator is available. It is useful for quick approximate measures on a system. But, simulations do not always capture the correct behavior of a system and they do not account for implementation errors in the real system.

In [Stott98], we compared simulated fault injection with SWIFI using a low-level simulation of Myrinet network and a SWIFI fault injector on the real system. The results of the comparison was that simulation does a poor job covering how the system behaves in invalid or unspecified states because either the behavior is undefined (e.g., if the simulation says that the network card should read data from an incorrect address in the host system’s memory, the data is undefined in the simulation) or the specifications are ambiguous. But, the simulation was very accurate when the system operated in a valid state.

2.5 Analytic Methods for Fault Analysis

Another method of analysis is analytic modeling [Trivedi82], such as Markov Models and Stochastic Petri Nets. These approaches build state models with probabilistic times in each state and solve them mathematically (or with the help of a tool, e.g., SHARPE [Sahner95]). Though these approaches are good for characterizing systems once metrics (e.g., failure rates) for individual components are know, it is difficult to obtain the metrics for components from these models alone.
2.6 Hybrid Approaches

Only a few studies claim to use more than one type of analysis (physical, SWIFI, or simulation). Young [Young93] combined SWIFI (using FTAPIE) with hardware and software error detection mechanisms. Guthoff [Guthoff95] combined SWIFI with simulation by taking a checkpoint of a target program running on a real system, simulating the effect of a low-level transient and injecting the symptom back into the real system. FlexFi [Benso99] uses a separate Background Debug Mode micro-controller (a COTS processor common on microprocessor boards) to inject faults into a Motorola 68K processor in real time. In all of these studies, it is difficult to classify the tool, as a whole, into one category (SWIFI, hardware fault injection, or simulation), because the trigger and the fault injector use different types. But, none of them can claim that they use more than one method to actually inject faults.

Kanawati (et. al.) [Kanawati95] have the first example of combining hardware and software fault injection and triggering. On a Sparc1 system they used FERRARI to trigger a hardware monitor (via the serial port) which, in turn, activated a pin-level hardware injector to inject faults into the system bus after system returned from the SWIFI fault injector to user mode. The results from this experiment compliment earlier work in which FERRARI injected software faults in a similar system. This study points out the importance of being able to use both hardware and software fault in a validation study—because there are many faults that can be injected using one method (hardware or software), but not the other.

2.7 Limitations of Fault Injection Tools

Despite all the research done for fault injection, there are several limitations with the current tool set. Each of these is addressed by NFTAPIE:

1. No tool supports more than two platforms (Orchestra supports Mach and Solaris, MESSALINE was used two distinct systems; no other tool supports more than one).

2. No tool supports more than one type of fault injection except FERRARI (which can use one specific bus injector and SWIFI approach).

3. No tool supports more than two types of fault models (several tools use communication faults and bit-flips to memory and/or register).

4. No tool supports more than one trigger method (except FTAPIE which used three methods, stress-, time-, and space-based triggers, to compare them).

Because of these limitations, no tool is applicable if the target system has the following properties:

1. multi-platform (heterogeneous) support,

2. several fault model or injection methods, or

3. fault injection under different trigger conditions;

4. using the he same control and configuration process for each type of analysis is also useful.

Two examples of campaigns which require all of these properties are dependability benchmarking and system-level analysis of large systems. Dependability benchmarking ranks several different systems in terms of some measure of dependability. This requires evaluating each system under the same conditions (workload, faultload, and trigger conditions). System-level analysis often uses several different systems, often including embedded systems. Such an analysis require injecting several types of faults under different trigger conditions. Studies have
shown that a large number of faults can be injected with either SWIFI or HW faults, but not with both, suggesting that both methods are important.

To date, no other work has been done toward developing a general-purpose fault injection environment that supports an abstract fault injection or fault trigger model like NFTAPE. To illustrate this point, it is not possible to take the trigger from one tool and use it directly in another tool; nor is it possible to use the same tool to inject more than one type of fault into different platforms.

3 NFTAPE Architecture

NFTAPE is a portable tool, build upon Lightweight Fault Injectors (LWFI), for executing fault injection experiments in distributed computer systems. LWFI (described in Section 4) are small modules that inject faults upon receiving a trigger event (from NFTAPE). Some of the components in any fault injection tool depend on the target system, while other components are very portable. NFTAPE separates these so that the portable components can be reused in different experiments and only the target-specific parts have to be rewritten to create a new systems. The most portable components deal with the high-level control of a campaign. In NFTAPE, these components all reside on a node called the Control Host. Each other node in the system is called a Target Node. NFTAPE runs a daemon program called the Process Manager on each Target Node to handle communication and running process on that node. The system dependent components (e.g., most LWFI and triggers) use a standard interface (part of NFTAPE API) to communicate with the Process Manager. Another reason for separating the Control Host from the part where faults are injected is to protect the control of the campaign from the effects of faults (such as a node failure). Figure 1 gives a high-level diagram of NFTAPE.

![NFTAPE Architecture Diagram](image)

Although NFTAPE is designed to support black box processes, some functions (such as fault triggering) are easier to handle when each program uses the same standard interface. NFTAPE provides such an interface and a library (referred to as either the NFTAPE API or the Application API). Because of this interface it is possible to interchange programs at run time (i.e., when specifying a campaign). That is, the Control Host can choose any combination of fault injector and fault trigger(s) for each campaign run.
This section first describes the Control Host in Section 3.1. Then, processes running on the Target Node (the Process Manager, general processes, lightweight fault injector processes, fault trigger processes) are described in Section 3.2. Next, error detection and evaluation metrics is described in Section 3.3. Next, Section 3.2.3 describes NFTAPE’s Application API. Section 3.4 describes an optional device driver interface.

### 3.1 Control Host

The Control Host performs all global control and configuration decisions. This node is generally separated from the target nodes so that the campaign can run in case parts of the target system are failing and so that the experiment is centrally controlled. The Control Host is implemented in Java so that it is portable across platforms; an earlier, simplified version of the Control Host was designed in Perl.

The job of the Control Host is to process a Campaign Strategy script, which is a file that specifies all the parameters for a fault injection experiment. Some of these parameters are which programs to run on the target nodes, when to run them, what parameters to give them, and which nodes to use. The Campaign Editor helps the user create or edit a Campaign Strategy. The Front End GUI is the user interface to NFTAPE.

The purpose of specifying a fault injection campaign strategy is to provide a simple and general way to customize each fault injection experiment. A fault injection campaign is the basic input to NFTAPE. The fault injection campaign consists of two parts: (1) the campaign script, which specifies how the experiments are run and (2) the campaign definition file, which specifies values for all of the parameters needed by NFTAPE to conduct the fault injection experiment.

As the campaign runs, the Status Monitor collects run-time information by reading messages from the target nodes. As the Status Monitor reads data from target nodes, it copies this data to a log file. The log files are separated by the node which generated the message and annotated with a timestamp and information about which process generated the message.

The Campaign State Machine (which is still under development) uses this information along with the Campaign Strategy to dynamically determine the global sequencing of actions in an experiment such as which process to run, what action to take when an error is reported, how many runs of an experiment should be executed, etc. Results described in this paper use a simplified version of the state machine. In the current version, it processes every messages from the Process Manager (including every line of output from the processes running on each node). When a message matches a know message format (e.g., the format the Process Manager uses to report that a process has completed, or a known string which an error detection process prints to report an error), the state machine performs some action such as moving on to the next experiment or starting a lightweight fault injector after the trigger and target application begin. The final version of the state machine will be automatically configured based on the Campaign Script. It will still use the log results from each node to update the state of the experiment and determine when to start processes.

Appendix A gives an example of a Campaign Script for a sample campaign. It should be noted, however, that the Control Host is incomplete and the Campaign Script parsing is not yet available.

### 3.2 Target Node

Each Target Node has two main components—the Process Manager and the set of processes running on the node. This section first describes the Process Manager. Next, it describes processes running under the Process Manager (such as LWFI and triggers). Next, it describes the NFTAPE API.

#### 3.2.1 Process Manager

The Process Manager, part of NFTAPE, needs to be ported to each new platform. Because the Process Manager uses common POSIX system calls, the effort to port it to new Unix systems has been very small (some difficulties
were encountered trying to port to different implementations of the POSIX-thread library). Though each port of
the Process Manager uses TCP/IP for communication, the communication functions are abstracted so that they
can be modified to run on systems where TCP/IP is unavailable (such as an embedded control system).

Distributed fault injection based analyses often need to manage several cooperating processes (e.g., workload
generators, monitors or heartbeats, workload applications, target applications, loggers, triggers, acceptance tests,
and lightweight fault injection processes). The Process Manager manages all of these processes. Managing them
entails starting the process, providing parameters, error catching (such as exiting with an error code due to an
invalid filename or arguments), processing output data from the process, and capturing the process termination.
NFTAPE simplifies process management by treating all processes abstractly rather than distinguishing them based
on functionality. In order to provide these services, we impose the restriction that the Process Manager must be a
parent of each of the processes (as is the case for each process the Process Manager creates itself).

To differentiate between different processes, the Control Host assigns a globally unique command id for each
command. This id can be used to direct message to that process. When the Process Manager sends output from the
process to the Control Host, it encapsulates the data with this id so that the Control Host can determine the source
of the data. When the process completes, the Process Manager notifies the Control Host by sending a message
with the exit status. If a timeout value is specified and the process has not completed before the timeout period
expires, then the Process Manager terminates the process and informs the Control Host. This is useful to detect if
a fault caused a process to hang.

Most processes used with NFTAPE can be treated as black box applications. Support programs (monitors,
acceptance test, etc.) generally output information to standard streams. In this case, the Process Manager will log
the process output to the log file on the Control Host where the Status Monitor can perform some action or it can
be processed off-line. Other processes should use the API to interface with NFTAPE so that every component will
use the same interface. For example, a fault injector may need to wait for a trigger and send a message after it
completes a fault injection. This should be done using the NFTAPE’s application API so that every program will
use the same interface.

3.2.2 Processes

Any process can run under the Process Manager; some examples of useful types of processes are lightweight
fault injectors, fault triggers, target application, synthetic workload generators, monitors, and acceptance tests.
When the Process Manager executes these programs, it connects their standard output streams so that they can be
logged by the Control Host. The Process Manager will also notify the Control Host when the process terminates.
To detect hung processes, processes can be started with a timeout value; if the execution time exceeds the timeout
value, the Process Manager will terminate the process.

Lightweight fault injectors are small programs which just inject faults. Unlike traditional fault injectors, they do
not include error detection, triggering, etc. Since they are simpler than traditional fault injectors, LWFI are smaller
and easier to write. NFTAPE has a skeleton for a LWFI which makes an API call to receive NFTAPE triggers and
registers a fault injection function (which the developer supplies) to inject the fault when the trigger arrives. This
way, a programmer only needs to write the function that injects the fault to write a fault injector.

An existing fault injector can be used with NFTAPE without modification. This is done by adding a wrapper
program that waits for a trigger event using NFTAPE’s API and then triggers a fault in the existing fault injector
using whatever method it expects.

The purpose of a fault trigger is to tell an LWFI when to inject a fault. Two simple examples of triggers
are timers and breakpoints. A more complicated example is to trigger a fault when a distributed system is in a
particular state. For example, the trigger event might be while the system has more than two unacknowledged
messages in transit. To inject faults when an application is in a particular state, the application can be modified
to produce its own trigger; this is similar to the idea of an Application-Specific Fault Injector (as described in
Section 4.1. On newer CPUs, performance counters can be used to trigger faults (e.g., Xception [Carreira95]).

Using more than one simple trigger, it is possible create interesting compound triggers. For example, suppose one trigger detects when the system begins recovery and another detects when a particular message type is sent, the two triggers can be cascaded to form a trigger that fires when the message type is sent during recovery. Because each trigger uses the same interface to produce a trigger event and each trigger can use the same interface that the LWFI uses to consume trigger. Figure 2 demonstrates this property.

![Figure 2. Cascade of Triggers](image)

In (A), the fault injector receives a trigger event every time the periodic timer fires. In (B), the stress monitor only forwards trigger events from the periodic timer trigger when the system stress exceeds a given threshold.

### 3.2.3 NFTAPE Application API

The purposes for NFTAPE API is threefold:

- to provide a common interface for processes used in an experiments,
- to handle communication between process in the experiment, and
- to provide a programming aide to simplify writing components.

It should be noted, however, that NFTAPE can usually use black box processes that do not use the API.

In the current version of the API, the API code filters every line of input to the program using standard input for application programs. The Process Manager uses the same API functions, but it passes all of its streams through the API; these streams are the socket to the Control Host and the pipes connected to each Target Process’s standard output and standard error streams. If the line of input matches the format of an API function, then the
API processes the message. In the applications that use the API, NFT APE data is the only data on the input stream. But, for the Process Manager, each program’s output shares the same streams as the API messages. Thus, any data that is not used by the API must be output from the program (which it encapsulates and passes to the Control Host to be logged).

Appendix B has an example trigger program and LWFI that use the NFT APE API. Each sample is fewer than one hundred lines of code. In these examples, the primary use of the API is to pass event information (trigger events and their acknowledgments). The API will also send an event to each program when the run has completed so that they can exit.

Some details of how the API should be processed can not be determined until the implementation of the Control Host is complete. For this reason, the way the API passes messages between components may change in the future. One of these implementation details is if the message routing should be performed by the Process Manager, or if it is sufficient to have the Control Host perform all such decisions.

### 3.3 Error Detection and Evaluation Metrics

NFT APE can evaluate a system using a wide range of dependability metrics including availability, reliability, coverage, error latency, etc. Rather than providing only one generic metric for evaluating the effects of all fault injections, NFT APE offers the flexibility of evaluating experimental data using whatever evaluation methods are most appropriate for the given experiment. An important issue in deriving dependability measures is how to determine whether an injected fault produces an error. Though NFT APE includes means for determining presence of errors it usually relies on the user to provide some methods of identifying errors and selecting dependability metrics. One exception to this is that the Process Manager provides some error detection for process termination. When any process terminates, the Process Manager records the exit status (for most operating systems, this includes any uncaught signal such as a segmentation fault). The Process Manager can also run applications with a timeout and detects a hang if the process does not complete within the timeout period.

Let us consider two examples to illustrate mechanisms that can be used for error detection. The first example is for an experiment that measures the network availability. In such a case, the user would need some means to measure the status of the network such as a heartbeat monitor which may also collect data about the latency in sending a message. NFT APE would configure the monitor and collect data from it, but would not interpret the data; the interpretation would be done off-line. In the second example, NFT APE is used to determine the coverage of a fault tolerant software algorithm (such as algorithm-based fault tolerant matrix multiply). In this case, the error is determined by an assertion check internal to the application or an acceptance test that runs on the program’s output. The application API can be used for informing NFT APE about an error. If the API is not used, then the program’s output could be searched for the assertion check’s error message off-line. If an acceptance test that runs on the programs output exists, then NFT APE can be configured to automatically run the test when the application completes.

### 3.4 Support for Device Drivers

It is often advantageous or necessary to use a device driver to trigger or to inject faults. Device drivers generally have privileges to resources which are protected in user mode and can run while the OS runs at high priority levels. NFT APE provides an interface to support using device drivers with its framework.

In most operating systems, device drivers define a set of functions which are executed when a user mode program makes a system call such as `open`, `read`, `write`, or `ioctl`. The `ioctl` (or I/O control) function is used to set or obtain driver-specific information. Drivers can also communicate with one another, but the methods varies from one operating system to another.

Figure 3 shows how a base trigger and a base fault injection driver are integrated to provide a common known interface to the user-level components.
To use a trigger driver, NFT APE must execute a user-level program which initializes the trigger (registers the driver with the base NFT APE trigger) and sets required parameters. This program does not need to interact with the driver while the experiment runs. A user-level fault injector can use a blocking 'read' system call on the trigger driver to wait for the trigger to fire. Alternatively, NFT APE can configure the base trigger driver to call a driver-level fault injector directly.

Using a fault injection driver is similar to using a trigger. The driver needs to be initialized (registered with the base fault injector). The fault injector should be triggered through NFT APE’s base fault injection driver, but (for debugging the driver) the fault injector usually has an `ioctl` function which user-level program can call to inject a fault directly. Alternatively, NFT APE can associate a the fault injector with a driver-level trigger so that the trigger can call the fault injector directly.

The purpose for the base fault injection and trigger driver is so that the fault injection drivers and trigger drivers can operate independently of each other. This way, NFT APE can dynamically configure any trigger driver to work with any fault injector and any fault injector driver can be triggered by any NFT APE trigger.

### 4 Lightweight Fault Injection

In this section, we describe Lightweight Fault Injectors (LWFI) to give the read a sense of the wide range of types of fault injections (and fault models) that can run in NFT APE. Loosely using some traditional terminology leads to confusing with respect to LWFI because the fault injector and the fault injection tool are separated. To help clarity this, Section 4.1 provides a taxonomy of fault injectors and Section 4.2 provides a taxonomy of fault types. It should be noted that most fault injectors can be used to inject several fault types.

NFT APE uses a new approach to building fault injectors. Instead of including common services (such as
communication, error detection, triggering, logging) with every fault injector, they can be designed to simply inject faults and rely on NFTAPE to provide the common services. The result is that fault injectors can be made very small easy to write. Such a fault injector, one whose only function is to inject faults while relying on another environment for common services, is called a lightweight fault injector (LWFI). Examining the code for FTAPE [Tsai96] showed that 95.8% performed functions other than injection faults (most of the code is configuration and communication which can be handled by NFTAPE).

The fault triggers that NFTAPE uses follow the a similar concept. The triggers simply wait for a triggering event (or combination of events) and then send a trigger event to NFTAPE. Section 3.2.2 describes how triggers can be cascaded to create complicated triggering conditions such as ‘while sending a particular message type while the system is undergoing recovery.’

### 4.1 Classifying Fault Injectors

Because thoroughly testing a system benefits from using more than one method of fault injection, NFTAPE supports numerous types of fault injection. To better understand what types of fault injection are possible, below is a method of classifying fault injectors.

Fault injectors can be classified in a few ways. One classification is the fault injection method. Although fault injection tools are difficult to classify this way because they include a hybrid fault injector and trigger (e.g., [Young93, Guthoff95, Kanawati95]), LWFIs can always be classified by these method since they rely on NFTAPE to provide a trigger.

Figure 4 classifies several fault injection methods. These methods can be classified as SWIFI, physical, or simulated. SWIFI injectors can be further classified target-specific, debugger-based, driver-based, or shared memory-based (e.g., multi-processors). A Target-Specific fault injectors is one where the target system is modified to add fault injection capabilities. Physical injectors can be divided into hardware fault injection methods such as pin-level, or special purpose hardware-based injectors, and physical fault injection methods, such as radiation, power disturbances, and electro-magnetic disturbances.

![Figure 4. Taxonomy of Fault Injection Methods](image-url)
4.2 Classifying Fault Types

Fault injectors can also be classified by the type of faults they inject, as shown in Figure 5. These types, referring to the type of faults that are injection, are target-specific faults (or errors), single event upsets (random or targeted memory faults), communication faults, and *-ability Faults. Communication faults are faults that affect the network in a distributed system (e.g., drop, corrupt, reorder, delay, and misroute messages). Single-event upsets include faults like memory, register, or bus faults (such as single-bit-flips, multiple-bit-flips, or set to zero). Target-Specific faults include faults that only make sense to the target system (e.g., corrupting a particular queue). Performance faults is a new class of faults (described below) that injects expected events that can affect the performance (e.g., a page fault). Another class of faults, labeled *-ability Faults, include several different types of faults that test the system on different measures of dependability which often use the suffix -ability, such as maintainability, usability, security, safety and robustness. For example, maintainability faults might include software upgrades and replacing components. This taxonomy is not as absolute as the taxonomy in Figure 4 because there is some overlap between categories. For example, communication faults could be used as security or robustness faults.

Another way to classify faults is the duration of the fault. The duration can be transient (the fault goes away after it is injected; e.g., a bit-flip to a value in a register), intermittent (the fault recurs at some rate; e.g., a power-supply fault), and permanent (the fault will always be present; e.g., connecting a pin to ground). Sometime the term semi-permanent is used to mean that the effect of the fault can be overwritten (e.g., if a value in memory is modified until the next store to that address), but the term transient can be used described the same set of faults. Fault duration can subdivide the entries in the fault type taxonomy in Figure 5.

In the past, fault injectors have also been classified by how the fault is triggered. Triggers can be path-based (fire when some code reaches a particular state or address), stress-based (fire when some measure of system stress exceeds a threshold), or time-based (fire at some time or interval). Sometimes space-based and time-based are used instead of these three terms. All LWFIs can use any of these methods because NFTAPE can provide a trigger based on any of these methods. Therefore, these terms should be used to describe the trigger criteria, but avoided when describing the injector.

Figure 5. Taxonomy of Transient Fault Types
4.3 Fault Injector Examples

4.3.1 Debugger-Based Fault Injectors

Debugger-based fault injectors (e.g., FERRARI [Kanawati92]) inject faults using the same interface that debuggers use (e.g., `ptrace()`). These injectors can set breakpoints in a target application, stop a target process when triggered, step through instruction, or (on many systems) trace system calls. If breakpoints are used or if the fault injector stops the program when NFTAPE triggers the fault, then the program runs at full speed except while a fault is being injected. Using trace mode (stepping through cycles or tracing system calls), on the other hand, can greatly effect system performance.

NFTAPE has been tested with debugger-based fault injectors for Linux, Solaris, and Windows NT. For each of these debugger-based fault injectors, the fault model was bit-flips to the process’s memory or register file. These fault injectors dynamically determined the pages used for the stack, heaps and code segments. When a fault was trigger, the injector randomly selected an address from one of these regions and bit position. These injectors can also be used to inject faults into a process’s copy of shared libraries. To compliment this fault injector, there is a debugger-based fault trigger that can trigger a fault on a breakpoint or on a system call.

For Linux, the fault injector uses the POSIX `ptrace(2)` interface. Most Unix systems provide system calls for this interface to be used by debuggers. The system calls allow the debugger to stop or restart a target process, to enter trace mode, and to read and write its memory and registers. This LFFI was effective in injecting faults into nodes running distributed scientific MPI programs (kmeans and matrix multiply).

Solaris uses the ‘/proc’ file system for debugging. It provides all of the features of `ptrace` as well as extra information about the process status and terminating condition. One example program that used this LWFI is a prototype of a space imaging application that will run in a high-radiation orbit.

Windows has its own set of functions for accessing memory and registers in a target process. This LWFI was used at Tandem Labs (Compaq) to test applications using a prototype of a new version of their Servernet SAN [Baker95].

4.3.2 Driver-Based Fault Injectors

There are several cases where user-level fault injectors are unable to inject faults because they lack permission to access data or because the OS will not schedule the fault injector when the fault needs to be injected. For example, a SWIFI fault injector can not inject faults while the OS is in a critical section. A solution to most of these permission barriers is to use a device driver to inject the fault because they have more privileges than user code.

One version of the driver-based fault injector injects memory faults anywhere into the system. Such a driver provides one function for injecting a fault at a given address and another function for obtaining information about the memory used by a target application. Device drivers use ‘ioctl’ calls to allow user processes to call these functions. Drivers like this have been developed for NFTAPE on Linux, LynxOS, and Windows 2000.

4.3.3 Target-Specific Fault Injector

While the first three fault injectors are useful when the target application can be treated as a black box, some high-level faults require knowledge of the target application. For example, suppose you are interested in analyzing how corrupting a message queue in one process may affect a process on another node. It is unlikely that such a fault will result from fault injectors mentioned above even though the fault model is simple and realistic.

One way to inject faults like this is to add code into the application that uses run-time information (such as the address of certain data structures) to inject high-level faults. NFTAPE provides an API for calling these functions.
In [Stott00b], this method is being used to inject faults into two SIFT (software implemented fault tolerance) middleware products, Chameleon [Kalbarczyk99a] and Voltan [Brasileiro96]. The objective of such an experiment is to compare the behavior of two systems and their abilities to protect applications against similar types of faults.

### 4.3.4 Shared-Memory-Processor Fault Injector

Embedded systems often contain more than one processor that can share resources such as memory. For example, many Motorola micro-processor boards include a small processor for debugging, that can be used to inject faults (e.g., [Benso99]). In such a setup, a fault injector can run on one processor and inject faults into the another that runs uninterrupted. Another variation of this type of fault injector was used in [Stott97] to inject faults into an embedded microprocessor in a Myrinet network interface card.

### 4.3.5 Physical Fault Injectors

Unlike SWIFI fault injectors that corrupt data using code running on the CPU, physical fault injectors use a separate piece of hardware to inject faults into the system. While this hardware works independently from NFTAPE, physical fault injectors usually have a software component for controlling the hardware component. This usually includes configuring it and turning it on and off. NFTAPE can be used to perform these processes and to collect results from the system while the target workload runs. In [Stott00a], a physical fault injector, that uses special-purpose hardware, was used to inject faults into the physical layer of a link in a Myrinet [Boden95] network. Earlier work [Stott98] showed a comparison of fault injected into the Myrinet NIC and those injected into a simulation of the system.

### 4.3.6 Simulated Fault Injectors

Simulated fault injection experiments have been used to assess systems before a prototype is available. Injecting faults into a simulation is usually easy because either the source code for the simulation is available, or (e.g., in the case of a VDHL model) the system model may be altered. When the source code for the simulation is available, simulated faults can be injected simply by adding code to inject the faults. As for the target-specific fault injector described above, if these functions use the API provided with NFTAPE, then NFTAPE can call the fault injection functions in the simulation. After a prototype and fault injector for the system are created, NFTAPE can inject the same fault model and trigger criteria into both systems. Work is underway building a simulation of a flight control system; NFTAPE will inject faults into the simulation in this way.

### 4.3.7 Performance Faults

While developing NFTAPE, we created one particularly useful new fault class—performance faults. Performance faults affect system performance instead of corrupting memory. Stalling the CPU or claiming system resources (e.g., memory or open files handles) are examples performance faults. Performance faults mimic the effect of events that are commonly observed in the real system such as page faults, buggy user processes, or resource leaks in the system. For enterprise computer systems with EDAC memory and redundant CPUs, these performance faults may be more meaningful than traditional memory bit-flips and may result in failures more frequently.

A preliminary experiment using a delay fault model for Windows 2000 demonstrated the effectiveness of performance faults with NFTAPE. In this experiment, the faults were injected by turning off interrupts and stalling the CPU for a short period. Using this simple fault model on a program for testing the robustness of network protocol interfaces, several tests failed that did not fail in the gold run.

A few other ideas for performance faults include faults are:
1. consuming all the available memory to test that the system can properly handle memory allocation exceptions,

2. consuming all the available file descriptors in the system to test the robustness of the programs running on system, or

3. swapping a page out of memory in a system to verify that timing constraints are meet when the system needs to handle the page fault.

5 NFTAPE Campaign Examples

NFTAPE has already been used with several fault injection campaigns and planned for others. These campaigns are listed below (the starred ones have not been completed yet):

1. Scientific applications (using Solaris debugger-based fault injector),

2. MPI applications (using Linux debugger-based and driver-based fault injectors),

3. Myrinet Hardware fault injector,

4. Fail-silent testing and comparison of Voltan and Chameleon, systems (using target-specific fault injectors and driver-based fault injector on Linux and Solaris), injector),

5. SEU faults on LynxOS operating system (using driver-based fault injector),

6. ServerNet on Windows 2K (using debugger-based and driver-based fault injectors and performance faults),

7. Fault Injection on Motorola fault-tolerant computer systems,*

8. Fault Injection on Sun High-Availability Cluster System,* and

9. Fault Injection into Flight System Simulation (using simulated fault injection).*

5.1 SEU Faults with Scientific Applications

One of the first experiments with NFTAPE explored the effects of single-event upset (SEU) faults in a scientific application running on a Sparc station. The target program, Next Generation Space Telescope (NGST), is a prototype of a real scientific program that the Jet Propulsion Laboratory (JPL-NASA) will run on a spacecraft in a future mission. The propose of the program is to suppress noise from the output of a sensor array and compress the image.

To test the program, NFTAPE used a debugger-based fault injector, called proc._fi. This fault injector uses Solaris’s /proc file system to control the execution of the target application and to inject faults into the program’s registers and data. The experiment showed that the noise suppression algorithm in not sufficient for protecting the output image from SEU faults. Details of the experiment are given in Appendix C.

5.2 SEU Faults with MPI Applications

Another campaign explored the effects of single event upsets in MPI applications. Two contributions from experiments are:

1. NFTAPE identified and overcame several difficulties in working with an MPI program in a cluster of workstations, and
2. NFTAPE compared the effectiveness of two different SWIFI methods injecting the same faults.

This is the only fault injection experiment we are aware of which can inject faults into different nodes of program running on a cluster of workstations where rsh is used to start each process. This paradigm (using rsh to start processes) is common in a cluster of workstations (e.g., MPLCH [Gropp96], the implementation of MPI which we used) because it is a simple way to start processes on any set of nodes without having to start a server on each node first. The problem, however, with this process is that it is difficult to determine when rsh is invoked if the target program is treated as a black box. (Here, it is actually the middleware (MPLCH) that invokes rsh.)

There are a few solutions to this problem. One solution to this is to modify rsh command in the user’s path so that it sends a message to NFTAPE before calling the real rsh or letting NFTAPE do the remote procedure call. Another solution is to trace system calls to find when the exec call to rsh is made, but this would cause the target application to run considerably slower. A third solution is to modify the target program to make an API call at start up which sends a message to NFTAPE; this solution was used. Both of these applications were modified only by adding the NFTAPE_init() and NFTAPE_exit() NFTAPE API function calls. The NFTAPE_init() call was necessary to ensure that the application runs under the Process Manager when MPICH uses rsh to start applications on remote nodes. The call to NFTAPE_exit() is used to determine if the application exited through the normal termination point.

The fault injection analysis uses two fault injectors with two target MPI applications running on a cluster of Linux workstations. One fault injector is a debugger-based fault injector (like proc_fi) which uses Linux’s ptrace() system call. The other is a driver-based fault injector that can inject memory faults into any process’s address space. The applications are k-means, a clustering algorithm, and abft-matmul, algorithm based fault tolerant matrix multiply.

The result of the experiment is that both fault injectors produce similar results when injecting memory faults. This suggests that the simpler driver-based fault injector is usually preferable to the more complicated and higher-overhead debugger-based fault injector. Results from this experiment are given in Appendix D.

5.3 Myrinet Hardware Fault Injector

To demonstrate NFTAPE’S flexibility in handling a very different types of fault injectors, we developed a campaign in which we controlled a hardware fault injector from NFTAPE. Burke and Floering developed hardware to inject faults into the physical layer of a Myrinet [Boden95] link at CRHC, UIUC. This is described in more detail in Appendix E.

5.4 Fail-Silent Comparison between Voltan and Chameleon

In [Stott00b], we compared the dependability of two methods of attaining fail-silent nodes in software (internal error checking as an alternative to full process replication). This one the first experiment we are aware of which compared the dependability of two different systems: Chameleon [Kalbarczyk99a] and Voltan [Brasileiro96]. Chameleon ARMORs claim to be able to provide fail-silence using only self-checking mechanisms (such as data signatures). A description of these self-checking mechanisms is given in [Whisnant98]. Voltan, is a middleware for providing fail-silent communication between distributed processes by replicating each process and voting on the messages from each replica.

Both systems executed two test programs, Fast Fourier Transform and the radix sort. The applications were parallelized to run on a pair of processes.

The analysis include three test sets:

1. to validate the coverage of detection mechanisms specific to each system,

2. to observe the effects specific faults for which the system offers no special protect, and
3. to compare how well each system protects the target from a similar set of faults.

The third test set showed that replication maintained fail-silence for 97% or the runs, and the self-checking method was only able to protect the system in 81.1% of the runs.

Details of this experiment are given in Appendix F.

5.5 SUE Faults to the Operating System: LynxOS

One campaign tested the reliability of a commercial real-time operating system (LynxOS 3.0.1 running on an Intel PentiumPro PC) against SEU faults in memory. The results from this experiment are purely qualitative at this point.

When porting NFTAPE to LynxOS, the regular NFTAPE distribution compiled on LynxOS without any problems or modifications. In a later version of NFTAPE, however, some incompatibilities in the POSIX threads library arose because LynxOS (like AIX 4.1) use an older version of pthreads than Linux and Solaris. These incompatibilities are handled with compiler macros and through use of the GNU autoconf program.

The only fault injector implemented for Lynx is a driver-based fault injector. It has the capabilities of injecting faults into any virtual address of any task in the system (or into the virtual address space of the kernel) or injecting register faults. Some care, however, must be taken when injecting register faults because every fault to the x86 segment registers was detected and resulted in rebooting the computer. Similarly, faults to the program counter and stack pointer, generally manifested immediately terminating the target process.

The fault injector was first tested on dummy programs. These (non real-time) programs behaved similarly to corrupt programs under other systems (e.g., Appendix C).

The interesting campaign was when we injected faults into the kernel address space. More specifically, accelerated the fault injection by locating the process table and injected faults there. Initially, we assumed that any fault to the kernel data structures would cause the computer to fail as soon as the data was accessed. To our surprise, the OS was able to withstand several (perhaps 10-100) faults before crashing. This suggests that the critical data in the kernel data structures are more sparse than we expected and they are more resilient to errors than we expected. An important lesson to be learned from this is that random faults inside the kernel usually lead to latent errors and non-silent errors.

5.6 Performance Faults and SEU Faults: ServerNet Windows2K Driver

In cooperation with Compaq Computers, Tandem Labs Division, NFTAPE was used to assess the dependability of a their ServerNet [Baker95] system area network (SAN) running in a Microsoft Windows. Campaq is using ServerNet is their high-end and highly-available computer systems which are used by large companies in the financial and communication industries (such as NYSE, Nasdaq, and Motorola) to keep their most critical systems running.

Since these high-availability servers already have protection against single-event upsets, Compaq was not interested in these faults. Instead, they had observed that most of the errors their development teams find in the late stages of development (i.e., the design or implementation faults which would gain the most benefit by being detected earlier in testing) are faults which manifest only under certain timing constraints or under very high load. For example, if one table entries is being updated while the system attempts to process a new message. The idea of accelerating errors by injecting faults which occur during the normal operation of the system instead of being caused by natural phenomena is now called performance faults (since most faults in this class effect the performance of the system).

We tried to write three fault injectors: (1) a delay-fault injector, (2) a memory-allocation injector, and (3) a traditional bit-flip injector. The first injector inserted timing errors by turning off interrupts and performing a busy wait for a specified about of time. The idea of the second injector was to allocate all of the available memory to
stress the memory exception handling routings of the drivers and programs running on the system. This injector, however, was not implemented because Windows does not appear to allow drivers to allocate memory. The third injector resembles the driver-based fault injector used in Section 5.5 and Appendix D.1. This experiment marks the first time a trigger outside the fault injector was used through NFTAPE. In this case, the trigger had to operate as a device driver so that faults could be injected while the operating system ran at a priority higher than user mode. The trigger was a timer-based device driver, but event-based triggering would possible by modifying the ServerNet device drivers (e.g., the function that received a message from the host computer could trigger such an event).

To generate workload on the network, we used a proprietary TCP/IP protocol testing harness from Microsoft. This program ran several test runs and labeled each as pass or fail.

As the workload ran, the fault injector injected delay faults with various delay times and fault rates as parameters. This simple model caused several test runs to fail. Some failures caused the same test case to fail for every run (including fault-free ones) until the system we rebooted, and others only caused a failure for a single run. The number of tests that failed increased with the delay duration, but the fault rate had little effect. It was not possible to tell if the faults manifested because of problems in the hardware or problems in the test harness.

The results from the campaigns are qualitative here because some of the results include proprietary information.

6 Future Direction

NFTAPE has made certain types of fault injection analyses possible, but more work needs to be done to evaluate how effective NFTAPE is at carrying out these analyses.

Some specific examples where NFTAPE can be applied are:

- **Fault Injection on Sun High-Availability Cluster System**: assess dependability of large system with several build in error detection mechanisms,

- **Fault Injection on Motorola fault-tolerant computer systems**: assess dependability of Motorola Mobile Base-station Computer System,

- **Fault Injection in Real-Time Systems**: evaluate usefulness of NFTAPE (and SWIFI, in general) for testing real-time systems and validate the target system which is an online upgrade system for real-time control programs,

- **Fault Injection into Flight System Simulation (using simulated fault injection)**: show that NFTAPE can be used in simulation for a project early in its design phase,

- **Dependability Comparison Studies between different CPUs or Operating Systems**: run fault injection campaign on a system running the same operating system and workload on different CPUs or two different operating systems on the same machine(s); this is the first step toward dependability benchmarking, and

- **Control-Flow testing**: recently, there has been interest in using software techniques to tolerate control-flow errors, but these studies are based on several untested assumptions about the frequency and behavior such faults.

6.1 Fault Injection on Sun High-Availability Cluster System

To assess the dependability of a high-availability computer system built on a Solaris platform with an Oracle database, Veritas reliable storage system, and mirrored disks, we have been asked to provide a fault injector with complicated fault triggering capabilities. An example of a fault scenario that has been suggested is:
when a particular request is made to the database, and

there is more than one outstanding disk request to two different channels,

inject a fault to cause a single disk to fail, and

after the disk fails, turn off power to the system.

To help detect the target states, code patches into Oracle and Veritas and provide a trigger. We can have access to the disk controller driver to cause a single disk to fail and to an interruptible power supply. It will also be necessary to collect log results from Oracle and Veritas.

Such a complicated fault injection scenario is not possible with any fault injection tool other than NFTAPE. A diagram of the system with all the processes, shown in Figure 6, shows two fault trigger processes and two fault injectors. The first trigger is the Oracle Context-Sensitive Trigger; it inserts a code patch into Oracle that detects the particular database request selected and passes a trigger event to NFTAPE. The second trigger, the Stress-Based Trigger, fires when there are more than two disk requests to two different channels. Because such an event could happen frequently, it makes sense to have a way of turning off the trigger when it is not in use. This is easy to implement by cascading the two triggers (as described in Section 3.2.2). That is, the stress trigger waits for a trigger event from the Oracle trigger before firing. After receiving such an event, it waits until its own trigger condition is true, and then produces a trigger event.

The diagram also shows two fault injectors. The first, the Disk Fault Injector, causes one of the mirrored disks to fail. This may be implemented by adding a call to the disk controller to take either disk off-line. This fault injector will be triggered by the stress-based trigger. The action of injecting the fault can be used as a trigger. That is, the disk fault injector’s acknowledgement can be used to trigger the second fault injector, the Power Fault Injector. This injector sends a command to the interruptible power supply to cause a system-wide power failure.

The diagram also shows two nodes to generate workload on the database. One includes a Memory Fault Injector that can be used in another campaign to see if a fault in a client application can propagate to the database.

6.2 Fault Injection on Motorola Fault-Tolerant Computer System

Another example of system-level assessment in large systems is a project with Motorola. Motorola produces several communication products each with very high availability requirements. These products range from stand-alone fault-tolerant computers with redundant CPUs to the communications systems they build like a mobile call processing system which uses several embedded processors and communication networks running various real-time operating systems. Because of the variety of different components used and because of the high availability requirements, these systems are ideally suited for NFTAPE. Due to the confidential nature of some of these systems and early stage of these projects, details of these systems are not presented here.

6.3 Fault Injection in Real-Time Systems

The target system, Simplex Architecture [Sha96], is an online software upgrade system which allows an operator to safely upgrade control software even when the new software has errors. An example of this uses two variations of a control programs (one well-tested baseline controller which is assumed to operate without error and one complex experimental controller which may contain a bug) that control a cart on a track to balance an inverted pendulum. The software runs on a real-time operating system, LynxOS on an Intel Pentium platform.

A diagram of the system is shown in Figure 7. A board with a DSP sends control commands to the cart and receives the pendulum's angle from a sensor on the cart. The control software consists of several processes (IP_Phys_IO, IP_Dec_C, IP_UIF, IP_Simplex, and IP_Complex). Each process communicates using message queues and shared memory. The I/O Interface process, IP_Phys_IO, communicates with the control board over a
The control processes (IP_Complex, and IP_Simplex) read the cart’s state from the I/O Interface and produce control values. The Decision and Control Process (IP_Dec_C) checks each controller’s control values against a range given by a function of the cart’s state. The Decision and Control Process has the ability to terminate a faulty controller; in which case, the Baseline Controller takes over. The User Interface Process (IP_UIF) allows the operator to install (execute) or remove (terminate) a new control program. This process uses a CORBA interface to communicate with the Decision and Control Process; it can be replaced with an automated program using the same interface for testing purposes.

The goal of this system is to safely allow online upgrades in real-time systems even when the new software contains bugs. This version of the system allow the upgrade to fail by:

- outputting bogus control values (i.e., any value outside of a predetermined “safe” control envelope),
- terminating abnormally (e.g., segmentation fault), or
- deadlock or livelock conditions (which do not hang the operating system).
The system is not designed to tolerate severe faults such as power failures or OS errors. The goals of the fault injection analysis are:

- to show NFTAPE can operate in a real-time environment,
- to verify the system can tolerate the faults it is designed to tolerate, and
- to analyze what failure modes a corrupt process may result in which cannot be tolerated by the system (e.g., flooding the message queues or corrupting shared memory).

The error detection for such a system is straightforward: the only failure case is where the pendulum falls. The analysis may look at triggering fault injection at interesting points such as while the system is switching between controllers. A simple type of analysis will be to inject random memory bit-flips into the Complex Controller’s address space. Faults like these will show how likely an error in the control program is likely to propagate to a shared resource causing a failure. Another important fault injection approach is to corrupt the control program’s output; this is significant because such faults will always propagate. A third type of analysis is to apply performance faults. The system needs to be able to tolerate the controller taking longer than expected to make its control decision; this can be emulated by injecting performance faults such as adding a busy wait into the controller.

### 6.4 Dependability Comparison Studies between different CPUs or Operating Systems

Surprisingly, no study has compared the dependability of different CPUs, platforms, or operating systems. Several straightforward experiments could be performed to these means. For example, to compare two operating systems, one might run the same campaign (i.e., same workload, triggers, faults, error detection, etc.) on two identical systems but with a different operating system on each. Since several operating systems have been ported to Intel x86 (Linux, Solaris, LynxOS, Windows), several such comparisons could be performed on that architecture.
Another example is to compare the dependability of two systems with different CPUs (with approximately the same performance characteristics) running the same operating system. Since Linux and Solaris run on both Intel x86 and Sparc architectures, either of these operating systems may be a good choice for such a study.

Campaigns like these will not only provide valuable information about the dependability of different architectures or operating systems, but they will also uncover many important details about issue that need to be resolved before proposing a dependability benchmark. For example, how do we provide a fault model that make sense for several different systems.

6.5 Control-Flow Testing

Recently, a few papers about software techniques for tolerating control-flow errors (e.g., [Alkhalifa99]) have sparked debate over such errors in real systems. A control-flow error is one which causes the program to execute a different set of instructions than the gold version (e.g., the fault changed an instruction into a branch instruction or corrupted a value in a jump table). Some control-flow errors are data dependent; that is, a fault may cause the branch condition to be incorrect (e.g., branch is not taken at if (i < 10) statement because fault changed i from 2 to 130).

Software approaches are promising for detecting control-flow that produce an invalid control-flow, but offer no protection against data-dependent control-flow errors. Previous studies on have found that a non-negligible fraction of faults in a CPU manifest as control-flow errors, but none of these studies distinguished between data-dependent and general control-flow errors. It is easy to see that data corruptions are likely to result in control-flow errors. But, in order for a fault to result in a general control-flow error, the following conditions must be met:

1. the fault must corrupt the instruction stream,
2. the corrupt instruction must be a branch instruction (corrupt function pointers or return address may also lead to a different class of control-flow errors that can not be protected by most techniques),
3. the branch must be taken (if the branch were not taken, the instruction would be treated like a no-op since no data is updated),
4. the destination of the branch must be a mapped address (otherwise the branch would generate an error), and
5. the destination of the branch must be a valid instruction (architectures that use variable-width instructions like x86 are more like to generate an illegal instruction than those with fixed-width instructions).

Because so many conditions must be met for a general control-flow error, it is reasonable to speculate that the frequency of general control-flow errors is negligible compared to data-dependent control-flow errors. But such a statement is purely speculation until verified (by fault injection).

I have proposed a scheme for performing a fault injection experiment that will calculate the probability of these errors. This scheme is generally applicable for any cycle-accurate testing. Figure 8 shows a diagram of this scheme. The basic idea is to run two copies of a target process and compare the registers (and data) of each program. Though this limits the analysis to deterministic programs with no real-time constraints, that is sufficient for learning about the how faults may propagate to control-flow errors. This procedure is almost identical to typical physical fault injection tools (e.g., [Karlsson94]) where two copies of the test hardware are used. In such experiments, one copy of the hardware is subjected to physical faults and special comparator hardware compares the pin-level output from each system every bus cycle to detect the effect of the fault. I am unaware of any study where this has been applied to a SWIFI approach.

The main component is the Debugger-Based Comparator and Trigger (DBCT). This process will execute two copies of the target application and run them in trace mode. The DBCT will trigger the fault on a condition
Figure 8. Diagram of Fault Injection Scheme for Cycle-by-cycle Program Testing

such as after executing a particular address \( n_a \) times, after \( n_s \) system calls, after executing \( n_c \) cycles, or some combination of these. The trigger will cause the fault injector to inject a fault into the test copy of the application (Application_test). A possible fault injector would be the debugger-based fault injector described in Section 5.1 to inject bit-flips into the instruction where the program counter is. After the fault is injected, the DBCT will trace the two copies of the application in single-step mode. At each cycle, the DBCT will compare the registers for each program to trace the effect of the fault. Optionally, DBCT can monitor all store instructions to trace the propagation of faults into memory. An experimental run continues until one of the following conditions is met (after which, it may be possible to compare the programs’ output files):

1. the program counters differ between the two copies immediately after a branch instruction; this is a data-dependent control-flow error,

2. the program counters differ between the two copies after a non-branch instruction (in the gold version); this is a general control-flow error,

3. the fault causes an exception (e.g., a segmentation fault),

4. a control-flow error detection mechanism detects the error,

5. the program successfully executes \( n \) instructions (where \( n \) is a predetermined threshold); the fault is considered masked, or

6. the fault is masked if it can be determined that there are no differences in memory and the register files are identical.

Using this process, we can determine the relative number of faults that fall into each of the following categories: (a) the fault manifested as a data-dependent control-flow error, (b) the fault manifested as a general control-flow error, (c) an error is detected by the system, (d) an error is detected by a control-flow error detection mechanism, and (e) fault is masked. The category of runs with an error detected by the system (b) can be divided into those detected at a branch instruction (in the test copy) and others. The last two categories may be divided by whether or not the error propagated to an incorrect output, an exception after successfully executing instructions after the branch, and those producing the correct output.
It should be noted that there are no special properties about this scheme that prevent it from operating without NFTAPE. Here, NFTAPE would be primarily used for its ability to automate the experimental procedure and to collect results.

7 Status and Conclusion

Several contributions have come out of studying how to make fault injection more efficient:

1. introduction of Lightweight FaultInjectors,
2. development of NFTAPE, an environment to use LWFIs,
3. porting of NFTAPE and several fault injectors to many systems,
4. execution of several dependability experiments, and
5. and examining how to benchmark systems in terms of dependability.

The concept of a Lightweight Fault Injector is a new idea. By separating the part of a fault injection tool that actually injectors faults from the rest of the tool, it become much easier to implement new fault injectors for exploring new fault models.

NFTAPE is a tool that provides an environment to run automated fault injection campaigns using LWFIs. It offers a new flexibility of combining fault triggering methods with fault injection methods. For example, before NFTAPE, it was not possible to take a fault trigger from one fault injection tool and use it to trigger a fault in another tool.

NFTAPE has already been ported to several platforms (Linux, Solaris, LynxOS, AIX, and Windows). Many different fault injection methods have already been demonstrated (debugger-based, driver-based, target-specific, and special-purpose hardware-based fault injection).

Dependability benchmarking has received growing interest lately. Such a procedure will require a tool for testing a system under several different types of stress-loads and fault-loads. Assessing the dependability of complex, heterogeneous computer systems share similar needs. NFTAPE is ideally suited for this type of analysis because

1. any stress- or fault-load configured and applied with NFTAPE, once the appropriate injector exists,
2. NFTAPE is portable and LWFI are easier to port than monolithic fault injection tools,
3. different fault injection methods can use the same trigger and workload,
4. NFTAPE provides a standard mechanism for starting processes in a distributed system and communicating with them, and
5. NFTAPE provides single method to log the results from campaigns instead of having different logging procedures for every fault injector.
A Sample Campaign Script

This section gives an example of a campaign script used in a simple campaign sample. In this example, we have a SIFT (Software Implemented Fault Tolerance) program similar to Chameleon [Kalbarczyk99a]. In this example, the SIFT middleware executes replicas of an application program on each of three nodes. Each node runs a manager called SIFT_Manager which monitors the replica on that node. If the replica fails (e.g., terminates abnormally or asserts an error), the manager will create a new replica. The manager has been enhanced to produce NFTAPE events, REPL_fail and REPL_restart when the replica fail and is restarted, respectively.

The goal of the campaign is to inject bit-flip faults into the third node (moe) at 1 fault per second while another node (larry, or curly) is recovering. To achieve this, two fault injectors and two trigger are added:

1. a fault injector, TerminatorFI, which terminates the application process,
2. a timer trigger runs on larry to determine when trigger the Terminator fault injector,
3. a fault injector (such as the proc_fi debugger-based fault injector) running on moe to inject bit-flips into the application, and
4. another timer trigger to trigger the fault injector on moe.

A diagram of the system is given in Figure 9. The diagram show three nodes (larry, curly, and moe). Each runs a copy of the SIFT_Manager, one of the replicas, and the NFTAPE Process Manager. One node has a Timer Trigger and the TerminatorFI. Another has a Timer Trigger and Fault Injector.

Below is an example of the campaign script for the campaign described above. The script first identifies the nodes where the Process Manager must run. Next the script lists several states. For each state, the script lists a set of commands and a set of possible next states. The only command used is run, which starts a process on a target node. The id keyword in the command associates the new process with an NFTAPE id which the script later uses to identify the process. The next_state command lists the events that must occur before the campaign moves to a new state. To leave the initial state, state_start_run, either all three process must start, or if any process fails, the next state will be the failure state. Finally, the script has a section to map events from one process to another. This allows the script to configure how events are routed (e.g., between triggers and fault injectors). The script supports renaming events (e.g., to convert REPL_fail events to EV_TRIGGER_ON); this way it can configure the replica failure events to turn the fault injector on or to turn the injector off (supposing you only want to inject faults while all three replicas are running instead of when one has failed).
Use nodes
   larry
   curly
   moe

state_start_run
  # start manager on each node
  run larry id=1 timeout=60 SIFT_Man
  run curly id=2 SIFT_Man
  run moe id=3 SIFT_Man
  next_state = state_manager_running (EV_APP_START 1 &
       EV_APP_START 2 & EV_APP_START 3)
  next_state = state_fail EV_APP_START_FAIL

state_manager_running
  # start application on each node
  run larry id=4 demo_appq
  run curly id=5 demo_appq
  run moe id=6 demo_appq
  next_state = state_app_running (EV_APP_START 4 &
       EV_APP_START 5 & EV_APP_START 6)
  next_state = state_fail EV_APP_START_FAIL

state_app_running
  # start fault injectors and triggers
  run larry id=7 Terminator -proc_name
  run curly id=8 SUE_FI -pid @pidof(6)
  run larry id=9 timer_trig -rate 4
  run curly id=10 timer_trig -rate 1
  next_state = state_all_running (EV_APP_START 7 &
       EV_APP_START 8 & EV_APP_START 9
       EV_APP_START 10)
  next_state = state_fail EV_APP_START_FAIL

state_all_running
  # stay in all_running state until first manager terminates
  next_state=cleanup EV_APP_DONE 1

event_mapping
  # Process    Event    SrcID DestID  [New_Event]
  SIFT_MAN    REPL_fail  1  9  EV_TRIGGER_ON
  SIFT_MAN    REPL_restart  1  9  EV_TRIGGER_OFF
B  Code Examples using the NFTAPE API

B.1  Sample Trigger Program using NFTAPE API

/*
*   Example timer trigger using NFTAPE API
*/

#include <stdio.h>
#include "nftapeapi.h"

/* nftape_interface data structure maps
 API commands to handler functions */

nftape_command Tt_trig_commands = {
    /* basic commands and commands for debugging */
    { "QUIT", &nftape_quit
        "terminate local control interface" },
    { "LIST_COMMANDS", &nftape_list_commands,
        "list available commands" },
    { "HELP_COMMANDS", &nftape_help_commands,
        "help for each command" },
    { "ECHO", &nftape_echo,
        "echo arguments" },
    /* commands for triggers */
    { "REJECTED_FAULT", &nftape_rejected_fault,
        "Inform trigger that fault has been injected" },
    { "PROC_DONE", &nftapeproc_done,
        "Announce completion of child process" },
    /* Empty entry to mark end of array */
    {0, } 
}

nftape_interface Timer_trigger_interface = {
    "Timer trigger interface",
    Tt_trig_commands
};

int main ( int argc, char *argv[] )


double rate_fps = 1.0;
int rate_us = 1e6/rate_fps;
int i;
nftape_command_data_t trigdata;
/* nftape_command_data_t has 'done' field which gets updated
 when the target has completed */
memset ( &trigdata, 0, sizeof ( trigdata ) );

for ( i = 1 ; i < argc; i++ )
{
    if ( strncmp ( argv[i], "-rate", 5 ) == 0 && argv[i+1] )
    {
        rate_fps = strtod ( argv[i+1], NULL );
        if ( rate_fps < 1.0e-6 )
        {
            rate_fps = 1.0;
        }
        rate_us = 1e6/rate_fps;
        fprintf ( stderr, "Setting fault interval to %d \n microseconds (rate = %g faults per second)\n",
            rate_us, rate_fps);
        i++;
    }
}

/* use setbuf to turn off buffering on stdout or use linebuffering */
setlinebuf ( stdout );

/* send interface data structure to API */
nftape_init_commands( &Timer_trigger_interface );

while ( !trigdata.done )
{
    usleep ( rate_us );
    trigdata.faultcount++;
    nftape_send_trigger ( "Timer Trigger" /* string is optional */ );
    trigdata.waitfornotice++;

    /* when the trigger consumer(s) send a reply, the API decrements
     * trigdata.waitfornotice */
    while ( trigdata.waitfornotice > 0 )

nftape_wait_for_message ( STDIN_FILENO,
       &trigdata
       /* API call updates trigdata */ );
}

fprintf ( stderr, "%s: exiting
\n", argv[0] );
exit ( 0 );

B.2 Sample Lightweight Fault Injector Program

/*
   * Example Fault Injector using NFTAPE API
   */

#include <stdio.h>
#include "nftapeapi.h"

/* nftape_interface data structure maps
   API commands to handler functions */
nftape_command Sample_fi_commands = {
   /* basic commands and commands for debugging */
   { "QUIT", &nftape_quit
     "terminate local control interface" },
   { "LIST_COMMANDS", &nftape_list_commands,
     "list available commands" },
   { "HELP_COMMANDS", &nftape_help_commands,
     "help for each command" },
   { "ECHO", &nftape_echo,
     "echo arguments" },
   
   /* commands for triggers */
   { "INJECT_FAULT", &nftape_inject_fault,
     "Fault injection has been triggered" },
   { "PROC_DONE", &nftape_proc_done,
     "Announce completion of child process" },

   /* Empty entry to mark end of array */
   {0, } }

nftape_interface Sample_fi_interface = {
   "Fault injector interface",
   Sample_fi_commands
}

32
void inject_function ( command_data_t *fidata )
{
    /* The purpose of this program is to demonstrate how to write a
       fault injector using the NFTAPE API. Instead of injecting an
       actual fault, this program tries to spook the system into making
       a mistake by outputting 'BOO'.
    */

    fprintf ( stderr, "BOO!!!!!\n" );
}

int main ( int argc, char *argv[] )
{
    int i;
    command_data_t fidata;

    /* no command line args to parse */

    /* use setbuf to turn off buffering on stdout or use linebuffering */
    setlinebuf ( stdout );

    /* initialize the command_data_t structure, including injfunc */
    memset ( &fidata, 0, sizeof(fidata) );

    /* the fault injection function needs to be set in fidata so that
       the default handler (nftape_inject_fault()) knows which function
       to call */
    nftape_set_fi_function ( &fidata, inject_function );

    /* set the command set */
    nftape_init_commands ( &Sample Fi interface );

    /* main loop */
    while (fidata.done == 0)
    {
        nftape_wait_for_message ( STDIN_FILENO /* read from stdin */, &fidata );
    }

    fprintf ( stdout, "%s complete\n", argv[0] );
    exit ( 0 );
}
C  SEU Faults with Scientific Applications

One of the early experiments for NFTAPE was testing the effects of single event upsets to memory on a scientific application. The target application is an image processing application (Next Generation Space Telescope (NGST), an application from Jet Propulsion Laboratory (JPL-NASA) for cosmic rays suppression and data compression). Its developers were interested in analyzing how the program reacts to faults similar to those it will experience in space because the environment in space is known to cause single event upsets to electronics. Because the system expects the sensor to frequently saturate from radiation in space, the algorithm first remove outliers it suspects were caused by cosmic rays depositing charge on the sensor. This version of the program (executing as a single process on Sparc Ultra-1) processes a series of 256x256 pixel image files (representing a noisy inputs from a sensor array) into a single filtered image. Next, the algorithm smoothes the image using the remaining points. The images in Figure 10 show (a) the original image (this is also the ideal output image), (b) a input image with noise added, (c) a fault-free run of the application, and (d) an example of a corrupt run.

![Figure 10. Images from Space Imaging Application](image)

(a, upper left) undistorted input file, (b, upper right) distorted input file, (c, lower left) application output without faults, (d, lower right) application output with faults.

Sometimes, the program can mask the effect of a fault by treating corrupt data like noise in the input image. In these cases, the effect of the fault may be acceptable, but blindly comparing the output image to a gold run will detect an error. For this reason, visually inspecting the output data may be a good approach to judge the effect of the fault. The goal here is to demonstrate that the fault injection can provide valuable insight into the application behavior under faults on an early design stage of the application. A secondary goal is to prove that
 noise suppression is not a sufficient means of fault tolerance.

The graphics in Figure 10 demonstrate that while the suppression code (the tested application) can remove the effects of the cosmic radiation (see Figure 10(c)) it cannot handle (cover) the impact of injected faults (see Figure 10(d)). When the program processes the data with a corrupt value, it should either reject the data point if the corrupt value is outside of the acceptable range, or process the corrupt data as if it were noise in the image. In either case, the output image should be slightly different than the gold run. Generally, such differences are acceptable. Visual inspection of the output data is needed to determine the effect of the fault in these cases. The application is an early prototype and it is difficult to determine whether the damage of the image provoked by the fault is significant or it can be neglected. The goal here is to demonstrate that the fault injection can provide valuable insight into the application behavior under faults on an early design stage of the application.

C.1  Proc-fi, /proc File System Fault Injector

The fault injector used for the scientific application experiment is called proc-fi, since it uses the Solaris /proc file system. The /proc file system was designed to allow debuggers to control the execution of a target process and to access its memory and registers. The input parameters to proc-fi are the target process’s id and the fault location (stack, heap, or register). When the NGST data was taken, the fault rate was also an input parameter, but now that rate is handled by any NFTAPE trigger. Before injecting a memory fault, it can query the system to find the process’s current stack and heap sizes. When the application completes, the fault injector collects information about the terminating condition (e.g., the exit status, any uncaught signal and cause of the signal). For example, if a fault causes a bus error, the return status would include the fault type such as ‘misaligned memory’ and the address of the memory access causing the error. Whenever a fault is injected, the fault injector displays process information state including the register file, the stack and heap size. All of this information is available from the system call to the /proc file system.

Figure 11 gives an example of a fault injected into register O7 (the return address). It changed the contents of the register from 0xEF6E6544 to 0xEB6E6544 (an address not mapped into physical memory). When returning from the subroutine, the program loaded the corrupt register value into the program counter generating a bus error. The excerpt from an actual fault injection run shows that the bus error resulted from an unmapped address and the offending address was 0xEB6E654C (i.e., the corrupt value plus 4).

The results from this campaign are shown in Table 2. Each row represents one set of runs; a set executes the application 25 times with the same injection parameters (location is either heap memory or register file; injection rate is constant and given in faults per seconds). The runs are classified as ‘No Error’ indicating the output file is identical to the one from the gold run, ‘< 25 errors’ if at most 25 points in the output file are incorrect, ‘> 25 errors’ if more than 25 points differ from the gold run. If the application failed to create an output file, then the run is classified as ‘No Output’. Most of the ‘No Output’ runs had some error signal such as a bus error. In some
cases, the application was able to write some or all of the output file before exiting with such an error; this is why the number of signals is sometimes greater than the number of ‘No Output’ runs.

The results suggest that the injecting faults to the register file has higher probability of severe errors (those causing the program to terminate) than for memory faults. But, the memory faults were much more likely to cause errors in the output data. One reason for this may be that most of the runs that exited abnormally would have produced corrupt data if they had continued. As expected, the number of more severe outcomes increases with the fault rate.

Table 2. Summary of Results from NGST Campaign

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>memory</td>
<td>1</td>
<td>0</td>
<td>25</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>memory</td>
<td>2</td>
<td>0</td>
<td>24</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>memory</td>
<td>5</td>
<td>0</td>
<td>21</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>memory</td>
<td>8</td>
<td>0</td>
<td>18</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>memory</td>
<td>10</td>
<td>0</td>
<td>13</td>
<td>7</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>memory</td>
<td>15</td>
<td>0</td>
<td>2</td>
<td>11</td>
<td>12</td>
<td>3</td>
<td>9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>memory</td>
<td>20</td>
<td>0</td>
<td>1</td>
<td>20</td>
<td>4</td>
<td>4</td>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>memory</td>
<td>25</td>
<td>0</td>
<td>1</td>
<td>7</td>
<td>17</td>
<td>4</td>
<td>14</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>memory</td>
<td>40</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>19</td>
<td>5</td>
<td>14</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>register</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>23</td>
<td>6</td>
<td>6</td>
<td>15</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>register</td>
<td>2</td>
<td>6</td>
<td>0</td>
<td>18</td>
<td>0</td>
<td>0</td>
<td>17</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>register</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>24</td>
<td>4</td>
<td>20</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

D SEU Faults with MPI Applications

The this fault injection campaign, NFTAPE is used to inject faults into two parallel MPI applications running on a network of workstations. These experiments run on a cluster of three 200MHz-Pentium PC’s running Linux 2.0.35 and two 200MHz-PentiumPro PC’s also running Linux 2.0.35. The applications used version 1.1.1 implementation of MPICH [Gropp96].

The first benchmark application a k-means clustering algorithm (which calculates $k$ cluster means for an $m$ data point input set in $n$ dimensions such that the choice of cluster means minimizes the sum of the distance between each point and the nearest mean squared). The data set uses 50000 points in 2 dimensions. The algorithm uses an iterative process to approach the optimal set of means. To parallelized the process, the data points are distributed between the nodes and each node calculates the contribution of its own data points to the RMS error term for the current estimate of the means. The algorithm terminates when the means converge then creates a file containing the set of means.

The second application is a parallel algorithm-based fault tolerant matrix multiply (abft_mm) [Huang84]. By augmenting each matrix with a checksum row and column before multiplying them, the product of the new matrices is the product of the original matrices augmented by a checksum row and column like the other matrices. The elements of the $i$th checksum row is simply the sum of the elements in the $i$th column and vice versa for the checksum column. The program verifies the correctness of the multiplication by checking the checksum of the product matrix. The input matrices are generated randomly at each node from a common seed value. The size of the matrices used are 400x100 and 100x300. The program repeats the multiplication twenty times (checking the product each iteration) so that the program will spend most of its time in the multiplying the matrices rather than setting up the matrices. To parallelize the operation, each node computes the product using a different sub matrix.
in the multiplier matrix. The checksums are added to the sub matrices rather than the original one. The program assumes the coverage of the abft is perfect and does not check the product besides by the checksums.

Early in the program’s execution, MPICH’s use of rsh to spawn processes caused problems for the distributed fault injector. The source of the problem is that the fault injector needs that the processes are spawned without modifying the MPI middleware. One solution to this is to modify rsh command in the user’s path so that it sends a message to NFTAPE before calling the real rsh or letting NFTAPE do the remote procedure call. Another solution is to trace system calls to find when the exec call to rsh is made, but this would cause the target application to run considerably slower. A third solution is to modify the target program to make an API call at start up which sends a message to NFTAPE; this solution was used. Both of these applications were modified only by adding the NFTAPE_init() and NFTAPE_exit() NFTAPE API function calls. The NFTAPE_init() call was necessary to ensure that the application runs under the Process Manager when MPICH uses rsh to start applications on remote nodes. The call to NFTAPE_exit() is used to determine if the application exited through the normal termination point.

D.1 Fault Injectors for MPI Experiments

NFTAPE uses two different fault injectors in this set of campaigns. The first injector, called ptracefi, is similar to procfi in Appendix C except that it uses the Linux ptrace() system call instead the Solaris /proc file system (this approach is similar to the one used in [Kanawati92]. The second one, driverfi uses a device driver to inject memory faults to an application.

The ptracefi fault injector takes an application and a configuration files as input. The configuration file specifies rates for each of four types fault injection areas including register file, stack and either of two heaps used in ELF binaries (the binary format used in Linux and Solaris divides the heap into two memory areas). The distribution of the injection rate is specified as an exponential, constant, uniform or normal random variable. In order to get the size of the memory regions, the fault injector run in trace mode to trace systems calls. In particular, the brk(), mmap(), and munmap() system calls update the size of the heaps. The size of the stack is determined by reading the stack pointer. Tracing system calls is also useful for following fork and exec calls which create and run new processes. When injection faults into the register file, it is often useful to protect some registers from injection. For example, every fault injected into the Intel x86 segment registers (CS, DS, SS, ...) causes a segmentation error since Linux uses paged rather than segmented memory. The program counter, stack pointer and frame pointer registers generally also lead to program termination. For these experiments the set of registers used included EAX, EBX, ECX, EDX, ESI, EDI, EFL they are the general purpose registers and condition code.

The driverfi fault injector has two components; the driver which injects faults and the client which controls the device driver. Its interface includes ioctl() calls to: (1) set the process id of the target application, (2) get the current memory usage of an application, (3) inject a memory fault, and (4) provide random access to the application’s memory space.

The client uses this interface to inject faults. The fault rate and target application are given as input parameters. The driverfi first starts the target application as a child process to get its process id and so that the injector can capture the terminating condition of the application. Between injections, the client sleeps for a interval corresponding to the fault rate. Periodically (every 8 injection periods) the client uses the driver to update the data structure containing the application’s memory usage. When injecting a fault, the client uses this data to generate a random heap address and bit position. The driverfi does not guarantee that all requested faults will be injected. For example, faults will not be injected if the memory address has been unmapped since reading the memory usage or if the memory address is paged to disk.
D.2 Results from MPI Experiments

Each of the two applications is executed with various parameter settings. The results are logged and processed off-line. For each run the following actions are taken: (1) set-up parameters for the fault injection and the application, (2) execute the gold run for the k-means application, (3) remove old output files (4) run the fault injector until the application completes or times out, (5) execute the acceptance test for k-means, and (6) repeat from step 3 until all runs complete.

The parameters used include the application selection (k-means or abft\_mm), the method and area of fault injection (heap using driver\_fi, register file using ptrace\_fi, stack, the first heap, the second heap, both heaps, all memory) and the fault rate (when multiple regions of memory are targeted, the rate is the same for each region; i.e., if faults are injected into 3 regions with a rate of 1 fault per second, then on average three faults are injected every second.) Each trial used 25 or 50 runs including the gold run.

In presenting the results from the campaign, each run is classified by the checking the correctness of outputs of the two programs (using an acceptance test and assertion check for kmeans’s and abft\_mm’s, respectively. The program’s output can be classified as ‘output good’, ‘output differs’, ‘error detected by application’, and ‘no output’ depending on if the application produced valid output, output differing from the gold run, error detected by the application, or exited without producing any output, respectively.

Tables 3 and 4 show the results of the campaign using the driver\_fi and the ptrace\_fi fault injectors. The first column of each table is the application under test (either abft\_mm or kmeans). For ptrace\_fi the table includes the fault location (either register, stack, heap or all memory regions) in the second column; for the driver fault injector, all faults were memory faults, so this column is omitted. The next two columns (1/rate) show the fault injection interval (mtbf in seconds) and the number of runs for those parameters. Faults are injected according to an exponential distribution with the rate given in the tables. The next four columns give the number of runs that fall into each result category (output good, output differs from gold run, error detected by application, or no output). Since abft\_mm does not use a gold run, the category ‘output differs from gold run’ does not apply, and since kmeans has no error detection, the category ‘error detected’ does not apply. All runs are represented by one of these four categories. The final category gives the number of runs that produced a signal (a segmentation fault or a bus error); these runs generally fell into the ‘no output’ or ‘error detected’ category.

### Table 3. Results from Driver Fault Injector from MPI Experiments

<table>
<thead>
<tr>
<th>application</th>
<th>1/rate</th>
<th>num. runs</th>
<th>good</th>
<th>output differs</th>
<th>errors</th>
<th>no output</th>
<th>signals</th>
</tr>
</thead>
<tbody>
<tr>
<td>abft_mm</td>
<td>.25</td>
<td>50</td>
<td>3</td>
<td>n.a.</td>
<td>42</td>
<td>5</td>
<td>47</td>
</tr>
<tr>
<td>abft_mm</td>
<td>.5</td>
<td>50</td>
<td>1</td>
<td>n.a.</td>
<td>37</td>
<td>12</td>
<td>40</td>
</tr>
<tr>
<td>abft_mm</td>
<td>1</td>
<td>50</td>
<td>4</td>
<td>n.a.</td>
<td>29</td>
<td>17</td>
<td>34</td>
</tr>
<tr>
<td>abft_mm</td>
<td>2</td>
<td>50</td>
<td>19</td>
<td>n.a.</td>
<td>17</td>
<td>14</td>
<td>17</td>
</tr>
<tr>
<td>kmeans</td>
<td>.25</td>
<td>50</td>
<td>1</td>
<td>8</td>
<td>n.a.</td>
<td>41</td>
<td>39</td>
</tr>
<tr>
<td>kmeans</td>
<td>.5</td>
<td>50</td>
<td>6</td>
<td>24</td>
<td>n.a.</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>kmeans</td>
<td>1</td>
<td>50</td>
<td>32</td>
<td>0</td>
<td>n.a.</td>
<td>18</td>
<td>11</td>
</tr>
</tbody>
</table>

As expected, the number of runs resulting in an error detected by abft and those resulting in a signal increase with the fault rate. For the most part, the number of runs that produced a good result decreases with the fault rate. In each case the number of runs that did not complete increases with the fault rate. For kmeans, runs that did not finish appear as ‘no output’, but for abft\_mm, runs that do not complete can be categorized as ‘error detected’ or ‘no output’. One interesting result is that the number of runs that produce a different output than the gold run reaches maximum with the middle fault rate. The reason for this is that the low fault rate does not produce enough faults to create an incorrect output, but with the higher fault rate, the number of faults causing the program to terminate before producing an output dominate.
Table 4. Results from ptrace_fi Fault Injector from MPI Experiments

<table>
<thead>
<tr>
<th>application</th>
<th>location</th>
<th>rate</th>
<th>num. runs</th>
<th>good</th>
<th>differs</th>
<th>errors</th>
<th>no output</th>
<th>signals</th>
</tr>
</thead>
<tbody>
<tr>
<td>abft_mm</td>
<td>allmem</td>
<td>.67</td>
<td>25</td>
<td>14</td>
<td>n.a.</td>
<td>10</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>abft_mm</td>
<td>allmem</td>
<td>1.3</td>
<td>25</td>
<td>21</td>
<td>n.a.</td>
<td>0</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>abft_mm</td>
<td>stack</td>
<td>1</td>
<td>25</td>
<td>11</td>
<td>n.a.</td>
<td>0</td>
<td>14</td>
<td>8</td>
</tr>
<tr>
<td>abft_mm</td>
<td>heap</td>
<td>2</td>
<td>25</td>
<td>21</td>
<td>n.a.</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>abft_mm</td>
<td>reg.</td>
<td>.5</td>
<td>25</td>
<td>15</td>
<td>n.a.</td>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>abft_mm</td>
<td>reg.</td>
<td>1</td>
<td>25</td>
<td>6</td>
<td>n.a.</td>
<td>0</td>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td>abft_mm</td>
<td>reg.</td>
<td>2</td>
<td>50</td>
<td>24</td>
<td>n.a.</td>
<td>0</td>
<td>26</td>
<td>2</td>
</tr>
<tr>
<td>abft_mm</td>
<td>reg.</td>
<td>4</td>
<td>25</td>
<td>17</td>
<td>n.a.</td>
<td>1</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>kmeans</td>
<td>allmem</td>
<td>.67</td>
<td>25</td>
<td>1</td>
<td>0</td>
<td>n.a.</td>
<td>24</td>
<td>5</td>
</tr>
<tr>
<td>kmeans</td>
<td>allmem</td>
<td>1.3</td>
<td>25</td>
<td>19</td>
<td>0</td>
<td>n.a.</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>kmeans</td>
<td>stack</td>
<td>1</td>
<td>25</td>
<td>10</td>
<td>0</td>
<td>n.a.</td>
<td>15</td>
<td>9</td>
</tr>
<tr>
<td>kmeans</td>
<td>heap</td>
<td>2</td>
<td>25</td>
<td>5</td>
<td>8</td>
<td>n.a.</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>kmeans</td>
<td>reg.</td>
<td>.5</td>
<td>25</td>
<td>1</td>
<td>0</td>
<td>n.a.</td>
<td>24</td>
<td>6</td>
</tr>
<tr>
<td>kmeans</td>
<td>reg.</td>
<td>1</td>
<td>50</td>
<td>8</td>
<td>6</td>
<td>n.a.</td>
<td>36</td>
<td>8</td>
</tr>
<tr>
<td>kmeans</td>
<td>reg.</td>
<td>2</td>
<td>50</td>
<td>16</td>
<td>1</td>
<td>n.a.</td>
<td>33</td>
<td>6</td>
</tr>
<tr>
<td>kmeans</td>
<td>reg.</td>
<td>4</td>
<td>50</td>
<td>9</td>
<td>5</td>
<td>n.a.</td>
<td>36</td>
<td>19</td>
</tr>
</tbody>
</table>

In the second set, we see the effect of changing the target area of memory has little effect on the results. Stack corruption lead to a terminating signal more frequently than the heap injections. Heap injections tended to lead to detectable errors more frequently than stack injections because stack corruption terminated more frequently and because the data for these applications is usually stored on the heap. Register corruption also lead to a wide range of failure modes.

E Myrinet Hardware Fault Injector

This experiment show how NFTAPE can be used with a physical fault injector rather than with SWIFI. To test the reliability of a Myrinet LAN [Boden95], Burke and Floering developed hardware to inject faults into the physical layer of a Myrinet link at CRHC, UIUC. Details of this injector can be found in [Stott00a].

One objective of for this fault injector was to monitor signals on the data lines and trigger injections on a match of a state or sequence of states on the data lines. The fault model included a variety of electrical faults including bit inversion, bit clipping, narrowing, delaying, and spurious behavior was required. The minimum insertion delay and skew were almost guaranteed due to the nature of the fast ECL (Emitter Coupled Logic) circuitry that was used, but care was taken when adding extra logic to not violate data transmission constraints. The delay needed to be small enough such that the fault injector could be inserted without exceeding the allowable link delay as described by the Myrinet LAN specification [ANSI98]. Lastly, the device was designed to be fully reconfigurable and controllable from an external interface, preferably a networked workstation running NFTAPE.

The Myrinet LAN was chosen because of its high speed and digital nature and because of its acceptance by several academic and government institutions such as the Office of Navy Research and JPL. The signals are transmitted digitally in the physical layer, making demodulation devices (which would be necessary for ethernet and other modulated transmission standards) for monitoring purposes unnecessary. The LAN signals are converted to PECL (Positive Emitter Coupled Logic) for manipulation with a single chip, so expensive transceivers were unnecessary. PECL was chosen as the domain for signal manipulation because of the availability of high speed logic and compatibility with 5 Volt programmable devices such as FPGAs.

The following is a summary of the basic blocks of the system as indicated in Figure 12; their meanings are explained in more detail in [Stott00a].
Figure 12. Basic Blocks of Myrinet Hardware Fault Injector

All fast decisions are made with the ECL discrete logic. The FPGA (1) provides registers for trigger and injection parameters, (2) reconfigures the external logic to respond to a particular location in the trigger data stream for a specific number of sub-bit slices, and (3) finally aligns injection data with the desired fault target location. This guarantees that all timing intensive operations are done in the ECL, while setup, communication with the host, and other complex yet timing insensitive tasks are performed by the FPGA.

E.1 Hardware Fault Injector Results

The fault injector performed up to its specification from a timing point of view. Total propagation delay was measured to be about 3ns, which is roughly the delay caused by one half of a meter of copper cable and well within the maximum of 10 meters allowable length for 1.28Gbps Myrinet [ANSI98]. Maximum skew caused by signals going through the inject logic was measured to be no more than 1ns, which is under the allowable 1.5ns, or one quarter of the character period, that is required by the specification.

E.2 Myrinet Hardware Fault Injector Outputs

Three injection methods have been tested in the current version of the Myrinet Hardware Fault Injector (Command Trigger, Data Sensing, and Fast Trigger).

E.2.1 Command Trigger

A command from the system connected through the RS-232 port can cause an injection signal toggle. The FPGA has been programmed to toggle the trigger signal when this command is received. This causes a single NRZI (the encoding used in Myrinet, Non-Return to Zero, Invert) bit-flip to occur at an interval that is easily programmed on the control node. This method is especially useful for random injection tests.

E.2.2 Data Sensing

The injector can monitor voltage levels on all nine lines of the Myrinet physical media and cause the trigger to be asserted when a particular pattern is sensed. This behavior has been captured using a digitizing oscilloscope and is presented in Figure 13. The indicated regions in Figure 13 are time periods where all nine lines meet injection criteria. Note that the figure shows the output of only one of the nine data lines and a single trigger can modify any combination of output signals. Both axes are scaled at 5ns/division.
A third injection method supports high-speed injection which allows corruption of adjacent bytes of the transmitted data. The FPGA is programmed to apply a number of high frequency pulses to the trigger when the fast trigger command is received on the serial line. This method can also be used to inject the CRC (Cyclic Redundancy Check) polynomial into the data stream therefore causing corrupt data to pass Myrinet’s error detection.

**E.2.3 Fast Trigger**

In [Stott00b], we compared the dependability of two methods of attaining fail-silent nodes in software (internal error checking as an alternative to full process replication). This one the first experiment we are aware of which compared the dependability of two different systems—Chameleon [Kalbarczyk99a] and Voltan [Brasileiro96]. Chameleon ARMORs [Kalbarczyk99a] claim to be able to provide fail-silence using only self-checking mechanisms (such as data signatures). A description of these self-checking mechanisms is given in [Whisnant98]. Voltan, is a middleware for providing fail-silent communication between distributed processes by replicating each process and voting on the messages from each replica.

The analysis consisted of three separate experiments test sets.

1. Test Set A validates the coverage of detection mechanisms specific to each system. The fault model included directed message control-flow and text segment corruptions for Chameleon and message corruptions, reordering, drop, and duplication for Voltan.

2. Test Set B observed the effects of worst-case fault scenarios for each system (specifically injecting faults for which the system offers no special protect). For Chameleon ARMORs, this test set consists of injections after the self-checks have been done, and for Voltan, it consists of injections where the faults are aliased by the signatures.

3. Test Set C injects random memory faults are injected instead of selecting faults based on features of the system being examined. The fault model includes random, low-level faults into the text, heap and stack of
the target processes. The purpose of this test set compare how well each system protects the target from a similar set of faults.

F.1 Test Applications

To test the systems, two simple target applications which can be parallelized into exactly two nodes. We selected Fast Fourier Transform (FFT) and radix sort.

The first program is a version of the FFT, a common algorithm for several scientific applications, such as signal processing. In our implementation (based on [Press92]), the algorithm runs in $\log_2(N)$ iterations, where each iteration performs $N/2$ parallel operations (the transform operates on an array of complex numbers with $N$ elements). The master node reads the input file from disk. For each iteration after the first one, (a) a copy of the array is sent to the slave node, (b) the operations are divided evenly among the two nodes, and (c) the slave node sends a copy of the array back to the master. For the experiments, we used a 4096 element input array.

The radix sort algorithm is a linear time integer sorting algorithm. For the $i$th iteration, the algorithm partitions the data based on whether the $i$th least significant bit is 0 or 1. To parallelize the algorithm, each node finds a different partition (the master finds the 0’s and the slave finds the 1’s). The slave node may send a different number of bytes on each iteration, depending upon the size of its partition. Unlike the FFT, every data value transmitted is processed. For the experiments, we sorted a 10000 element array.

F.2 Experimental Testbed for Voltan and Chameleon Campaigns

Figure 14 show the configurations in which systems run. Both systems ran their own version of two target applications–Fast Fourier Transform (FFT) and the radix sort. Both applications are parallelized into two tasks, denoted as Master and Slave. In Voltan, each task is run as a duplicated process: a Leader and a Follower. Thus, there are four application processes: Master Leader, Master Follower, Slave Leader, and Slave Follower. In Chameleon, for the purpose of this study, duplication of processes was not used, and hence, there were only two application processes, Master and Slave, corresponding to the two application tasks. In both systems, fault injection targets the application processes.

F.3 Target-Specific Fault Injectors

In this set of experiments, we use two basic classes of fault injectors. The first class injects traditional single event upsets to the target process’s memory (heap (data), stack, and text (code) segments). Because this class uses a
simple, easy-to-understand fault model that applies to any system, it is a good choice from which to build the basis of our comparison. The second class uses what is called program-specific fault injectors. This class of injectors includes those that inject faults from functions within the target program. Using such an injector, faults can be tailored toward the specific implementation (e.g., perform message corruptions in Voltan when the messages are in the Delivered Message Queue or corrupt the control fields in the ARMOR messages). The specific faults that were injected are described in Sections F.5.1-F.6.2.

The message queues in Voltan appear to be a vulnerable component, since most of the work in Voltan for providing fail-silence is processed in them. The effect of corrupting the queues will resemble timing errors or event ordering errors between nodes, which are common causes of errors in agreement protocols. Thus, we are interested in the effect of corrupting the queues with faults such as dropping queued messages, reordering messages, inserting duplicate messages, and corrupting messages. In particular, the DMQ and the VMQ queue (see [Stott00b] or [Brasileiro96] for a description of these queues) may be most vulnerable, since they send messages directly to the application or to the other node.

To support these corruptions, only a few simple modifications to the Voltan application library were needed. First, methods were added to the Queue class: FiDrop(), FiReorder(), FiDuplicate(), and FiModify() (which respectively popped a message form the queue, reordered two adjacent messages in the queue, duplicated a message in the queue, and corrupted the contents of a message). Next, a function was added that takes a string describing a fault (the name of a queue and the type of fault) as input and calls the appropriate fault method (from the four faults above). Next, in order to access this function, a simple class was added that opens a named pipe (e.g., a Unix socket) to wait for trigger events. At this point, any program can inject a fault by writing a message to the named pipe.

In Chameleon, the ARMOR processes communicate the data and control through messages. Hence, the fail-silence property of the system depends on the integrity of the messages being exchanged. Therefore, we are interested in injecting faults that affect messages, such as message corruption, message drop, and message duplication. To support this fault model, a fault injection element was added to the application written as an ARMOR. Since the element executes as a thread within the same address space as the target application, it has direct access to the application’s memory and messages. The fault injection element can be triggered either by an internal timer or by sending a message from an external source on a named pipe.

**F.4 Configuring Target-Specific Fault Injectors**

It is often desirable to use program-specific information (such as the addresses of special data structures) to guide fault or error injection. One approach to doing this is to add functions to the program to perform such injections. The disadvantage of this approach is that the programmer needs to include a way to trigger the fault and log information about the injection parameters. But, with NFTAPE all of these functions are provided by the API library.

To trigger the Voltan fault injection methods through the named pipe interface, a small program was needed to receive NFTAPE triggers (a function provided in the NFTAPE API library). Figure 15 show the complete Voltan fault injector. To inject memory faults, NFTAPE executes and triggers a preexisting driver-based fault injector [Stott99].

To trigger the fault injection elements in Chameleon through the named pipe interface, it uses an interface program just like the one for Voltan. This small interface program to convert the NFTAPE API format to Chameleon’s named pipe interface.
F.5 Test Set A: Assessment of Specific Detection Mechanisms

F.5.1 Assessment of Specific Detection Mechanisms in Voltan

In this test set, faults are selected to evaluate the coverage of the detection techniques in each system.

As described in Section F.3, we added the capability of injecting specific errors into the queues used by Voltan to process messages. The faults injected were the modification of a message waiting in a queue, the removal of a message from a queue, the addition of a duplicate of a message held in a queue, and reordering pairs of messages held in a queue.

Initially these faults were injected by NFTAPE directly into a specific queue. It was found that if injections are performed at regular time intervals, then in the vast majority of cases, no messages are present in the relevant queue. Therefore, instead of using a time-based trigger, we injected faults when messages were pushed onto a particular queue; this guarantees that a message is always present when a fault is injected.

The queues selected for injection were the Delivered Message Queue (DMQ) and the Voted Message Queue (VMQ) of the slave processes of the radix sort and FFT applications. Errors in these queues represent faults arising when messages arrive and depart from the Voltan nodes. The results in Table 5 show the effect of injecting the faults listed above into the given Voltan queues. No variations in behavior were displayed between the 25 runs with 6 faults each while running the radix sort, and the same behavior was observed for FFT.

Table 5. Results from Test Set A for Voltan

<table>
<thead>
<tr>
<th>Queue</th>
<th>Reorder</th>
<th>Drop</th>
<th>Modify</th>
<th>Duplicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leader DMQ</td>
<td>Correct</td>
<td>Fail-Silent</td>
<td>Fail-Silent</td>
<td>Correct</td>
</tr>
<tr>
<td>Follower DMQ</td>
<td>Correct</td>
<td>Fail-Silent</td>
<td>Fail-Silent</td>
<td>Correct</td>
</tr>
<tr>
<td>Leader VMQ</td>
<td>Correct</td>
<td>Correct</td>
<td>Fail-Silent</td>
<td>Correct</td>
</tr>
<tr>
<td>Follower VMQ</td>
<td>Correct</td>
<td>Correct</td>
<td>Fail-Silent</td>
<td>Correct</td>
</tr>
</tbody>
</table>

The results show that Voltan provides 100% fail-silence coverage for this type of message injection. Injections into the Leader and the Follower produced identical results. Nizam, the process that oversees the application process’s initialization, is the only unprotected process in Voltan. It is only invoked at the beginning and at the end of the application run. Faults are not injected into Nizam because it is likely to crash.
F.5.2 Assessment of Specific Detection Mechanisms in Chameleon Self-Checking ARMORs

The results from Test Set A of the directed message injections into ARMORs are shown in Table 6. In this Test Set, program-specific fault injections are conducted. The injection is targeted at the control fields of the ARMOR’s messages. This type of injection is meant to stress the coarse-grained signature mechanism of ARMORs. The injection occurs after the message has been generated by the application element but before the coarse-grained signature element has checked the message. This type of fault injection cannot be time-triggered exactly, rather it is a hybrid of message-rate-based and time-based triggering. In our implementation, the fault injection element attempted to inject a fault once per second; if no message was present, the fault injector would set a flag to inject the next message(s).

Table 6. Results from Test Set A for Chameleon

<table>
<thead>
<tr>
<th>Result</th>
<th>Control-field Corruption</th>
<th>Delay</th>
<th>Drop</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FFT</td>
<td>rsort</td>
<td>FFT</td>
</tr>
<tr>
<td>Good</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Hang</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Crash Det.</td>
<td>11</td>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>Coarse-grain Sig. Det.</td>
<td>15</td>
<td>10</td>
<td>30</td>
</tr>
<tr>
<td>Non-Fail-Silent</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>30</td>
<td>30</td>
<td>30</td>
</tr>
</tbody>
</table>

The results from Table 6 show that the coarse-grained signature is effective in all but one case of message corruption. The important point is that in 35-60% of cases, the error is detected without a crash of the process. The single case of non-fail-silent behavior observed with FFT resulted from the corruption of the pointer field of one element of the linked list of message operations. As a result, the FFT application skipped an operation on one element of the input data, and the output result differed from the golden run for one data point. The message delay and drop either cause a good result or a process crash. Thus, all the cases are fail-silent. Radix sort (rsort) shows greater resilience to delayed and dropped messages because of the native implementation.

Another set of injections was targeted at the text segment of the FFT and radix sort applications (only results for FFT are presented). Random single-bit-flips were injected into the text segment, which is protected through the text segment signature in Chameleon. The results of these injections are shown in Table 7.

Table 7. Results from Text Segment Chameleon Injections for FFT

<table>
<thead>
<tr>
<th>Result</th>
<th>With Text Signature</th>
<th>W/O Text Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Good</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Hang</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Crash Det.</td>
<td>24</td>
<td>20</td>
</tr>
<tr>
<td>Text Sig. Det.</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>Non-Fail-Silent</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>30</td>
<td>30</td>
</tr>
</tbody>
</table>

All the elements in the Chameleon library (about 40) are statically linked into each ARMOR, though the ARMOR may be using just a few elements from the library (e.g., FFT only uses three elements, including the core
FFT element). When randomly injecting over the entire text segment, the text segment signature check always
detected the error unless the fault caused the program to abort first. Since there is such a large amount of dead
code compiled into the text segment, targeting injections to code that is more likely to execute provides a better
idea of how effective the signature check is. To test this, we targeted faults into the text segment’s pages that
contained elements in use and into those of the application element. The results from these injections, shown in
Table 7 (columns Elem. and FFT), demonstrate that the text segment signature is able to capture most of the faults
and again not only prevent fail-silence violations but also capture the error before a process crash. It can be seen
that text segment signature results in some false positives; that is, it signals an error when it detects a bit-flip in the
text segment, though that fault may not otherwise have caused any error. As a result, there are no good runs with
the text signature turned on. To reduce the number of false positives, we have proposed a modification of the text
signature technique to flag an error only when the fault is in a page of the text segment that is being used or that is
going to be used shortly.

F.6 Test Set B: Assessment of System Vulnerabilities

F.6.1 Assessment of System Vulnerabilities in Voltan

In this Test Set, directed fault injection is done into areas for which no direct detection technique is present in
the system.

The message injections described in Section F.5.1 do not investigate the situation where a modified message
produces the same signature as the original message. This possibility can be made vanishingly small by using
digital signature techniques. However, the existing implementation uses only a simple checksum in the range
0-255 to authenticate data. We investigated the effect of injecting an error into messages that leave the signature
unaltered. If corruption occurs within a message, there is a 1/256 chance of its being accepted as authentic by
Voltan. We injected faults into the Delivered Message Queue of the leader slave node. This represents the worst-
case fault scenario in a Voltan node, where corrupt messages have been processed by both leader and follower
nodes. The results obtained are shown in Table 8.

<table>
<thead>
<tr>
<th></th>
<th>radix sort</th>
<th>FFT</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct Results</td>
<td>6</td>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td>Fail-Silent Behavior</td>
<td>12</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>Non-Fail-Silent</td>
<td>12</td>
<td>22</td>
<td>34</td>
</tr>
<tr>
<td>Total</td>
<td>30</td>
<td>30</td>
<td>60</td>
</tr>
</tbody>
</table>

In 12 of the 22 runs of the FFT that produced incorrect results, all the output data were accurate to at least 5
significant digits. In the radix sort, the incorrect results were always sorted but values had been modified. One
reason for the difference between the applications may be that the radix sort can detect corrupt data if the size of
each node’s partitions are different from the gold run. It can be seen that in this scenario, fail-silent violations can
occur, but there is still a significant chance (43%) that fail-silent violations can be prevented.

F.6.2 Assessment of System Vulnerabilities in Chameleon Self-Checking ARMORs

For Chameleon, the message injections in Section F.5.2 do not investigate the situation in which the data field
of a message is corrupted. The purpose of Test Set B is to look at how ARMORs react to different kinds of faults
in the messages. In this case, faults were randomly injected into the entire message, both control and data fields. In addition, duplicate messages were sent.

### Table 9. Results from Test Set B for Chameleon

<table>
<thead>
<tr>
<th>Result</th>
<th>Duplicate FFT rsort</th>
<th>Modify FFT rsort</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Good</td>
<td>0 4 0 2</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>Hang</td>
<td>2 0 0 0</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Crash Det.</td>
<td>0 0 0 0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Course-Grain Det.</td>
<td>9 14 16 8</td>
<td></td>
<td>47</td>
</tr>
<tr>
<td>Non-Fail-Silent</td>
<td>19 12 14 20</td>
<td></td>
<td>65</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>30 30 30 30</td>
<td></td>
<td>120</td>
</tr>
</tbody>
</table>

The results show that Chameleon ARMORs are susceptible to message duplication and message corruption. Message corruptions are done after the messages have passed all the intra-ARMOR checks. Because application-specific messages consist primarily of data (the size of the data is up to 64 times the size of the control) random injections usually corrupt the message data that are not protected in the current Chameleon implementation. While both data audits to protect against data error and duplicate message checks have been implemented, they had not been ported to the target environment at the time of the experiments.

Another set of injections is done to the application process’s heap data. Since the heap data is not protected in the current Chameleon implementation, this class of injections also falls in the category of exposing the system vulnerabilities. Results from this injection are presented in the column titled Heap in Table 11.

### F.7 Test Set C: Assessment of Random Memory Faults

#### F.7.1 Assessment of Random Memory Faults in Voltan

The Test Set set tested the Voltan system against random memory faults (memory bit-flips into the text, stack, and heap sections of each of the test applications). Each combination of fault location and application included 25 runs, with a fault rate of one fault per second. The results of each run are shown in Table 10.

### Table 10. Results from Random Memory Bit-Flips in Voltan

<table>
<thead>
<tr>
<th>Result</th>
<th>Heap FFT rsort</th>
<th>Text FFT rsort</th>
<th>Stack FFT rsort</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Good</td>
<td>13 13</td>
<td>11 16</td>
<td>19 18</td>
<td>90</td>
</tr>
<tr>
<td>Hang</td>
<td>9 11</td>
<td>2 0</td>
<td>6 7</td>
<td>35</td>
</tr>
<tr>
<td>Seg. Fault</td>
<td>0 0</td>
<td>12 9</td>
<td>0 0</td>
<td>21</td>
</tr>
<tr>
<td>Non-Fail-Silent</td>
<td>3 1</td>
<td>0 0</td>
<td>0 0</td>
<td>4</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>25 25</td>
<td>25 25</td>
<td>25 25</td>
<td>150</td>
</tr>
</tbody>
</table>

The heap injections provided interesting results. This is where most of the program data and the messaging data are stored. For about half the runs, the program produced the correct output. In four runs (three for FFT and one for the radix sort), the output from one of the nodes differed from the golden run in exactly one data point. This probably happened because the fault corrupted a value between the time it was received from another node and the time the value was written to file, or because the fault affected data that was not used by the other node.

In [Stott00b], results from inject faults in nodes other than the Master-Leader node are given.
In all but 4 of the 150 (2.67%) runs, the program either produced the correct output (60%), hung (23%), or aborted on a segmentation fault (14%). In each of these cases, the program either produced the correct output or no output (fitting the definition of fail-silent).

Process duplication in Voltan protected an application from producing an error in 55 of the 59 runs where a fault manifested by either hanging a node or crashing silently. (Presumably, the cases where a node crashed would have also crashed if Voltan was not used.) Only 6.7% of these runs had errors that escaped Voltan.

F.7.2 Assessment of Random Memory Faults in Chameleon Self-Checking ARMORs

The results from Test Set C for Chameleon are shown in Table 11. In this Test Set, we examined the effects of injecting random bit-flips into the heap, stack, and text segment of a Chameleon ARMOR. In addition to the core application element, there are also a few Chameleon-specific elements in the ARMOR, e.g., the named pipe management element for establishing communication with the local daemon, the coarse and fine-grained signature elements, and the text segment signature element. In the experiments, the heap injection is done to the entire ARMOR’s heap, the stack injection is done to every thread’s stack, and the text injection is done to the text segment of the application element only.

<table>
<thead>
<tr>
<th>Result</th>
<th>Heap FFT</th>
<th>Stack FFT rsort</th>
<th>Text FFT rsort</th>
<th>Total FFT rsort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Good</td>
<td>9</td>
<td>7</td>
<td>5</td>
<td>14</td>
</tr>
<tr>
<td>Hang</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Crash Det.</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>22</td>
</tr>
<tr>
<td>L2 Det.</td>
<td>2</td>
<td>2</td>
<td>24</td>
<td>33</td>
</tr>
<tr>
<td>Non-Fail-Silent</td>
<td>18</td>
<td>15</td>
<td>0</td>
<td>19</td>
</tr>
<tr>
<td>Total</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>90</td>
</tr>
</tbody>
</table>

The results show a fail-silence coverage of 81.1% averaged over the two applications and using both text segment injections (individually 78.9% and 83.3% for the radix sort and FFT respectively). The signature mechanisms in the target implementation protect the text segment and control-flow of the ARMORs, but they do not provide any detection for the data (which accounted for all but one of the Non-Fail-Silent runs).

Another observation is that the text injection does not cause any control-flow errors. If a control-flow error occurs, it is detected by the fine-grained signature. Or, in other words, the fine-grained signature did not detect any errors.
References


