

# NFTAPE: A Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors

David T. Stott, Benjamin Floering, Daniel Burke, Zbigniew Kalbarczyk and Ravishankar K. Iyer

Center for Reliable and High-Performance Computing

Coordinated Science Laboratory

University of Illinois at Urbana-Champaign

1308 W. Main St., Urbana, IL 61801-2307

Phone: (217)244-6974

E-mail: {dstott, floering, kalbar, iyer}@crhc.uiuc.edu

## Abstract

*Many fault injection tools are available for dependability assessment. Although these tools are good at injecting a single fault model into a single system, they suffer from two main limitations for use in distributed systems: (1) no single tool is sufficient for injecting all necessary fault models; (2) it is difficult to port these tools to new systems. NFTAPE, a tool for composing automated fault injection experiments from available lightweight fault injectors, triggers, monitors, and other components, helps to solve these problems.*

*We have conducted experiments using NFTAPE with several types of lightweight fault injectors, including driver-based, debugger-based, target-specific, simulation-based, hardware-based, and performance-fault injections. Two example experiments are described in this paper. The first uses a hardware fault injector with a Myrinet LAN; the other uses a Software Implemented Fault Injection (SWIFI) fault injector to target a space-imaging application.*

**Keywords:** fault injection, automated testing, dependability assessment, distributed systems

## 1. Introduction

Most of the many existing fault injection tools were developed using good software engineering practices and are modular in design. Nonetheless, each is to some extent specific to a particular, albeit novel and useful, fault injection method. For example, Xception [5] uses hardware debugging registers, Ferrari [13] uses traps, Orchestra [8] is directed toward corrupting messages, and FTAPE [20] ties fault injection to the workload. A result of this specificity is that it is nontrivial to enhance any of these tools to incorporate other fault injection mechanisms or to trigger multiple or correlated faults using a common control mecha-

nism. Consequently, no single available tool provides more than a few essential fault injection features, porting each tool is difficult, and becoming proficient with each tool is time consuming.

A motivating factor for developing NFTAPE came from failing to find an automated fault injection tool that would support the set of features needed by the Jet Propulsion Laboratory to evaluate a computer system developed to run scientific experiments in space. A partial list of the requirements includes:

1. **Multiple Fault Models:** bit-flips in registers and memory (kernel, application's virtual address space, random physical addresses), communication errors, and I/O faults,
2. **Multiple Fault Triggers:** path-based, time-based, and event-based triggers (including correlated events between nodes),
3. **Multiple Targets:** distributed (scientific) MPI (Message Passing Interface) applications, SIFT (software implemented fault tolerance) middleware layer, black box applications, communication interface hardware, and operating system, and
4. **Versatile Error Reporting Methods:** tradeoff between collecting detailed data from fault injection experiments and intrusiveness of fault injectors; ability to dump memory regions when needed.

Simulation-based tools are useful before the system is built, but they tend to do a poor job of capturing the correct behavior of corrupt systems. Hardware-based fault injectors provide valuable insights, but they cannot be used exclusively because (1) limited controllability—most physical fault injectors cannot inject controlled (specific or repeatable) faults, (2) limited accessibility—pin-level fault injectors cannot introduce register or cache faults or communi-

cation errors, and (3) high implementation cost—an extra, dedicated hardware must be used to facilitate fault injection. SWIFI (Software Implemented Fault Injection) fault injectors offer great flexibility and relative simplicity in implementation. A survey of available SWIFI tools found that they have several shortcomings, many of which also apply to hardware- and simulation-based tools.

1. **Portability Problem:** The effort to port these tools to a new system is too high. Even the cost of implementing a very simple fault model (such as terminating processes randomly) is high because the programmer also needs to write code to coordinate the experiment, to log results, and to clean up the experimental processes.
2. **Limited Support for Distributed Systems:** Support for distributed systems is limited, except for those injectors targeting communication faults. Initially, we ran into several difficulties targeting MPI applications because the implementation of MPI we used uses **rsh** to start processes on remote nodes. We are unaware of any tool that can target applications that use remote procedure calls to start new processes.
3. **Component Reuse Problem:** The components in each of the available tools are dependent on each other. Thus, even if one fault injector had an outstanding fault trigger, this component would be incompatible with other tools.
4. **Lack of Flexibility of Injection Method:** Since no single tool supports multiple fault models, we need a new tool for each fault model we are interested in. Porting so many tools to a new system is impractical. Furthermore, each tool has its own interface for configuring experiments, its own set of triggers, and its own method of data collection. Thus, even if several tools were ported to the system, it would take considerable time to learn to use them.

NFTAPE uses a new approach to building fault injectors that address the shortcoming listed above. The reason that traditional monolithic fault injectors are difficult to port is that the component that injects the fault (which is system-specific and difficult to port) is dependent on other components. With NFTAPE, the fault injection component is replaced by a *LightWeight Fault Injector* (LWFI), the trigger becomes an NFTAPE trigger process (similar in concept to a LWFI), and the remaining functions (such as logging, configuration, and communication) are handled by NFTAPE. To develop new fault injectors, only the LWFI (which is much simpler than a traditional fault injector) needs to be written; NFTAPE also provides an API to further facilitate development of fault injectors. Because LWFI and triggers use a standard interface, they can be interchanged. This way, NFTAPE can inject faults using any fault injection

method and any fault model (provided that an LWFI exists). To our knowledge, NFTAPE is the first nonsimulation tool that supports an arbitrary fault model. NFTAPE was designed specifically for use in distributed systems, but it may also run on a single machine.

This paper presents the architecture of NFTAPE and provides two example experiments to justify the above claims. Our objective is to demonstrate the versatility of the NFTAPE validation framework in performing a wide range of fault injection experiments using different fault injectors and handling different applications. None of the previous injectors achieve this level of flexibility. Clearly the NFTAPE environment can also be used to evaluate a single application using different fault injectors.

The remaining of the paper is organized as follows: Section 2 describes related research. Section 3 describes the NFTAPE architecture. Section 4 describes fault injection in NFTAPE. Section 5 presents two example experiments conducted with NFTAPE one using a hardware fault injector with a Myrinet LAN and the other using a debugger-based fault injector with an image processing application. Section 6 summarizes the paper.

## 2. Related Work

Fault injection has been used to analyze the dependability of computer systems by accelerating the rate at which errors occur in the system [6, 1, 10]. In particular, fault injection aims at (1) exposing deficiencies of fault tolerance mechanisms (i.e., fault removal) and (2) evaluating the coverage of fault tolerance mechanisms (i.e., fault forecasting). Several tools have been developed for different methods of fault injection. These injectors fall into three categories: software implemented fault injectors (SWIFI), physical fault injectors, and simulated fault injectors.

SWIFI [18, 14, 5, 8] is very popular because it is inexpensive, easy to develop, and runs in software. Xception [5] uses the debugging features of modern processors to inject and to trigger faults.

Physical fault injection has the advantage of sometimes being able to use a fault model that is closest to faults experienced in the field. These tools use pin-level injection [16, 1], ion radiation and EMI [15], and even laser-based injections [17]. However, they require special-purpose hardware, are often difficult to operate or to coordinate with software on the target system, and can be harmful to the target.

Simulated fault injectors [7, 11, 9] are very flexible in terms of the fault model, the fault trigger, and data collection. This is because all the information about the fault and the system is available to the programmer. However, [19] showed that simulated fault injectors can be inaccurate in modeling the behavior of corrupt systems because specifications for failure states are usually vague or undefined.

A hybrid approach (e.g., [14]) combines features of software-based and hardware-based fault injection techniques to induce fault conditions on external buses and signals that cannot be injected through software techniques. The injection experiments are controlled and synchronized using dedicated, reconfigurable hardware.

Another key characteristic of fault injectors is the method for triggering faults. Common triggers include time-based triggers, path-based triggers, and stress-based triggers. Time-based triggers are the simplest; faults are injected at some interval (usually a random variable). Path-based (and event-based) triggers inject faults when the target (usually an application) executes specific events. Usually, these triggers can be implemented using software traps or bus monitors. Stress-based triggers [20] inject faults when the workload on the target system is above a specified threshold level. Xception [5] uses features of the microprocessor (e.g., PowerPC) to trigger on events such as “the  $n$ th cpu cycle after the  $m$ th data cache miss.” Other fault injectors, e.g. Orchestra [8], allow the user to specify conditions for fault injection based on events such as the arrival of specific messages.

### 3. NFTAPE Architecture

Fault injection tools need to provide mechanisms for (1) injecting faults, (2) triggering injections, (3) producing workloads, (4) detecting errors, and (5) logging results. Unlike other tools, NFTAPE separates these components so that new combinations of LWFI, triggers, and workloads can be selected. This approach has several advantages; for example:

- LWFIs are simpler than traditional fault injectors, which need to integrate the trigger, logging mechanism, and communication support, and
- the fault triggering and error detection mechanism developed by one party can be reused by others.

To facilitate this approach NFTAPE (1) defines an interface for each component, (2) handles communication and provides an API for the components to communicate with one another, (3) logs all activities during a fault injection experiment, and (4) manages processes involved in each experiment. Figure 1 shows a block diagram of the NFTAPE components and their relationships. The following sections describe these components.

#### 3.1. Control Host

All control decisions are made by the Control Host. This node is generally separate from the set of Target Nodes. The job of the Control Host is to process a Campaign Strategy script, which is a file that specifies all the parameters for a

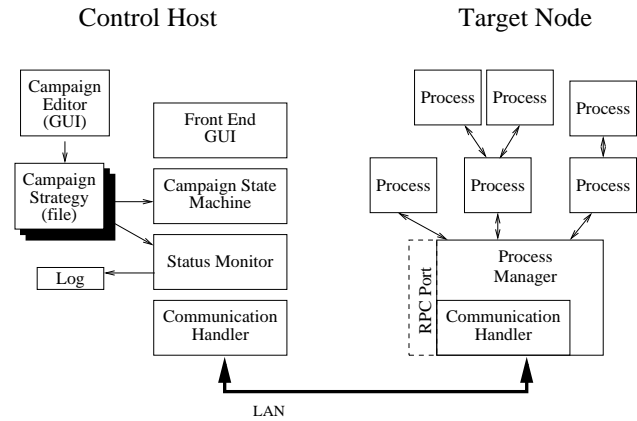


Figure 1. NFTAPE Architecture

fault injection experiment. The Campaign Editor helps the user to create or to 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 yet general way to customize a fault injection experiment. A fault injection campaign is the basic input to NFTAPE. It 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 runtime information by reading messages from the target nodes. The Campaign State Machine (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 were obtained using a preliminary version of the State Machine. The Campaign State Machine processes messages from the Process Manager (including every line of output from the processes running on each node). When an incoming message matches a known message format (e.g., the format the Process Manager uses to report a process termination, or a predefined string which an error detection process prints to report an error), the State Machine performs appropriate actions such as moving on to the next experiment or starting a LWFI. It should be noted that, in the final version, the state machine will be automatically configured based on the Campaign Script.

#### 3.2. Process Manager

Distributed fault injection based analyzes need to manage several cooperating processes (e.g., workload generators, monitors or heartbeats, target applications, loggers, triggers, acceptance tests, and LWFI processes). Managing

them entails providing parameters for the process, starting the process, recognizing error conditions (such as invalid filename or arguments), processing data from the process, and capturing the process termination. In order to provide these services, we require the Process Manager to be a parent of all processes involved in a given fault injection campaign (e.g., injectors, monitors, application).

To start a process with the Process Manager, the Control Host sends a command to the Process Manager - the command line to execute, and an optional timeout value. Every message from the Process Manager to the Control Host and line of output from any process is tagged with an ID so that the Control Host can run several commands at once and separate the output by the command that created it. 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 in detecting whether a fault caused a process to hang.

Supporting processes on a target node (monitors, acceptance test, etc.) generally output information to standard streams. The Process Manager logs output data to the log file on the Control Host where the Status Monitor can take appropriate actions (corresponding to the logged data) or the data can be processed off-line.

### 3.3. Fault Injection

Lightweight fault injectors in NFTAPE are small programs responsible for injecting faults. Unlike traditional fault injectors, they usually do not include error detection, fault triggering, etc. The LWFI uses the NFTAPE API function call to wait for a trigger.

An existing fault injector can be used with NFTAPE without modification. This is achieved 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 input method that injector expects.

Section 4 describes examples of several types of LWFIs used with NFTAPE. Section 4.2 gives an example of how to add a new fault injector to NFTAPE.

### 3.4. Fault Trigger

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. In a more complicated example, a fault can be triggered when a system or application enters a particular state (e.g., during recovery). To inject faults when the application is in a specific state, the application can be modified to support fault triggering. On newer CPUs, performance counters can be used to trigger faults (e.g., [5]).

In NFTAPE, the output from a trigger is a message to inject a fault. A LWFI can receive a trigger message from any

trigger process. This way, NFTAPE can reuse one party's trigger with a fault injector from another.

### 3.5. Evaluation Metrics and Error Detection

NFTAPE can evaluate a system using a wide range of dependability metrics including availability, reliability, coverage, error latency, and others. Rather than providing only one generic metric for evaluating the effects of all fault injections, NFTAPE 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. NFTAPE relies on the user to provide some methods of identifying errors and selecting dependability metrics. An exception to this is a support provided by the Process Manager in detecting the process termination due to an error. When a process terminates, the Process Manager records the exit status (for most operating systems, this includes any uncaught signal, such as a segmentation fault). In addition, the Process Manager uses a timeout to detect process hangs.

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 determine the status of the network, for example using a heartbeat monitor, which may also collect data about the latency in sending messages. NFTAPE can configure the heartbeat monitor and collect data, which can be interpreted off-line. In the second example, NFTAPE 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 by an acceptance test that runs on the program's output. The application API can be used for reporting errors to the Control Host. If the API is not used, then the program's output could be searched off-line for the assertion check's error message. If an acceptance test that runs on the programs output exists, then NFTAPE can be configured to automatically run the test when the application completes.

### 3.6. Application API and Communication

Processes running in NFTAPE can communicate in either of two ways: using the NFTAPE API or using standard streams. The Process Manager reads, processes, and logs all data that processes write to standard streams. The API provides functions for passing data such as trigger information, process status, etc. This way, a black-box application can run in NFTAPE. For example, a system performance monitor can periodically (e.g., every second) output performance data to be logged by the Control Host.

## 4. NFTAPE Fault Injector Examples

This section gives examples of six classes of fault injectors used in NFTAPE and demonstrates how to integrate an example hardware fault injector with NFTAPE.

### 4.1. Fault Injection Methods

Because thoroughly testing of a system requires using more than one method of fault injection, NFTAPE supports numerous types of fault injection. So far, NFTAPE has been used with software implemented fault injection (debugger-based, driver-based, and target-specific), simulated fault injection, physical fault injection, and performance faults (an alternative to memory bit-flips or source-level errors).

#### 4.1.1 Debugger-Based Fault Injector

Debugger-based fault injectors (e.g., FERRARI [13]) inject faults using the same interface that debuggers use. These injectors can set breakpoints in a target application, stop a target process when injection is triggered, step through instructions, or (on many systems) trace system calls. In NFTAPE if breakpoints are used or if the fault injector stops the program when NFTAPE triggers the fault, 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-based platforms. For each of these debugger-based fault injectors, the fault model was bit-flips to the process's memory or the register file. These fault injectors dynamically determined the pages used for the stack, heaps and code segments. When a fault was triggered, the injector randomly selected an address from one of these regions and a bit position to inject. These injectors can also be used to inject faults into a process's copy of shared libraries.

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

Solaris operating system 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 process termination conditions. One example program that used this LWFI is a prototype of a space imaging application that will run in a high-radiation orbit.

Windows operating system 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 [2].

#### 4.1.2 Driver-Based Fault Injector

There are several cases where user-level fault injectors are unable to inject faults because they lack permission to access data or because the operating system will not schedule the fault injector when the fault needs to be injected. For example, a SWIFI fault injector cannot inject faults while the operating system is in a critical section. A solution to overcome most of these permission barriers is to use a device driver to inject faults. The device drivers 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 Linux, LynxOS, and Windows 2000 operating systems.

#### 4.1.3 Driver-Based Performance Fault Injector

Performance faults affect system performance instead of corrupting memory. Stalling the CPU or claiming system resources (e.g., memory or open files handles) are example 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 protected (error detecting and correcting) memory and redundant CPUs, these performance faults may be more representative than traditional memory bit-flips and may result in system failures more frequently.

A preliminary experiment using a delay fault model for Windows 2000 demonstrated the effectiveness of performance faults. 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 although the same tests did not fail in the gold run.

#### 4.1.4 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 random faults to memory or registers even though the fault model is simple and realistic.

One way to inject faults like this is to add code into the application which uses run-time or compile-time information (such as the address of certain data structures) to inject high-level faults. NFTAPE provides an API for calling these functions.

This method is being used to compare two SIFT (software implemented fault tolerance) middlewares, Chameleon [12] and VOLTAN [4] to determine if it is possible to attain the same fail-silence coverage using run-time assertions without duplication.

#### 4.1.5 Simulated Fault Injector

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 the system model may be altered (e.g., in the case of a VDHL model).

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 (including trigger condition) into both systems. Work is underway building a simulation of a flight control system.

#### 4.1.6 Physical Fault Injector

Unlike SWIFI fault injectors which 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 the hardware fault injector 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. Section 5.1 describes a fault injector used to inject physical layer faults into a Myrinet [3] network. Earlier work [19] showed a comparison of fault injected into the Myrinet NIC (network interface card) and those injected into a simulation of the system.

### 4.2. Adding a Hypothetical Fault Injector to NFTAPE

This section describes a hypothetical pin-level fault injector as an example of how a typical LWFI can be used with NFTAPE. The fault injector uses pin-level injections on the I/O bus to test specific devices on the bus. Before

each run, the user gives initialization information (e.g., the address range(s) of the target device). The LWFI can be commanded to inject a fault immediately (on the next bus cycle in the address range) or it can be given a pattern to match on the bus (e.g., a particular address) before injecting a fault. The command also includes a bit pattern to inject.

To integrate this fault injector with NFTAPE, the injector needs (1) to provide a profile information and (2) to use the API to communicate with NFTAPE environment.

The profile contains information about the injector such as the command line format (which will be set at run time) and a list of supported NFTAPE functions. To set the injection pattern and bitmask during runtime, the profile provides a reference to a file containing a fault list. The NFTAPE API provides a function to register the fault injector so the injector can be invoked whenever NFTAPE sends a trigger event. Similar functions can be registered for turning the fault injector on or off and setting parameters. The API also facilitates collecting and logging (or printing to standard I/O) runtime status information reported by the injector (e.g., confirmation that a fault was injected and when).

## 5. Experiments

This section describes two experiments carried out using NFTAPE. The experiments were chosen to demonstrate NFTAPE's flexibility in conducting a wide range of fault injection experiments. The first example is a hardware fault injector which injects errors to data and control signals transmitted over a Myrinet LAN. The second example is a SWIFI campaign on a scientific space application.

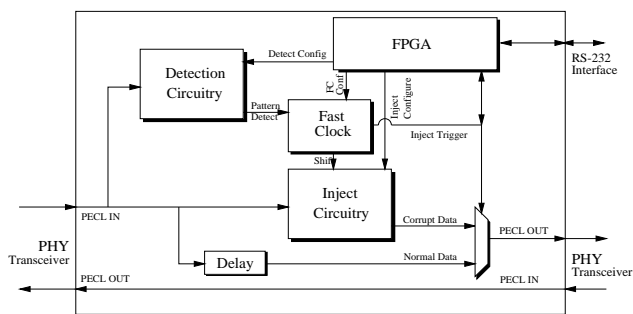
### 5.1. Hardware-Based Myrinet Fault Injector

A hardware-based fault injector was developed to provide fault injection functionality for Myrinet LAN networks. The ability to monitor signals on the data lines and trigger injection by matching a state or sequence of states on the data lines was desired. The ability to inject a variety of electrical faults including bit inversion, bit clipping, narrowing, delaying, and spurious behavior was required. Minimum insertion delay and skew were 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 [21]. 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. 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 5V programmable devices such as FPGAs. The high-speed nature of Myrinet will enable this design to be applicable to other high-speed data transmission media such as fiber channel and 100BaseT.

The final design is a balance of speed and complexity. A SRAM-based Xilinx part with the highest speed grade at the time was used to provide an interface with the controlling device. The Xilinx part was also used to configure and interact with the faster ECL logic. The following is a summary of the basic blocks of the system as indicated in Figure 2.



**Figure 2. Basic Blocks of Myrinet Hardware Fault Injector**

**Detection Circuitry:** High-speed ECL for the detection of up to four sequences of states on the incoming data. Upon detection of this state or states, the Pattern Detect line is asserted.

**Fast Clock:** This finely-tuned ECL logic provides the Trigger Signal which in turn causes the inject data to be selected. A Fast Clock is applied to the Inject Circuitry which applies the configured fault to the output. The Fast Clock is required for the synthesis of narrow, delay, and spurious faults. The Fast Clock can be configured by the FPGA to output any number of pulses or to simply act as a pass-through.

**Inject Circuitry:** Injection of the configured data is performed by this ECL logic. Inject Patterns are loaded by the FPGA and shifted to the output by the fast clock.

**FPGA:** An RS-232 interface is used to communicate injection parameters to the FPGA from an external device. Several injection operations can be controlled in real-time by the external device by issuing commands through the RS-232 Line. This allows for greater injection flexibility

as described later. The FPGA has the ability to change the Detection Circuitry, Fast Clock Config, and Inject Patterns at any time. It also can monitor the Inject Trigger for statistics purposes as well as assert the trigger line itself.

**Physical Interface:** The injector accepts data at PECL data levels, compatible with several types of transceivers and easily converted from LVDS (Myrinet LAN) and similar digital transmission signaling. This modular design guarantees compatibility with many types of transmission media.

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.

### 5.1.1 Hardware Fault Injector Results

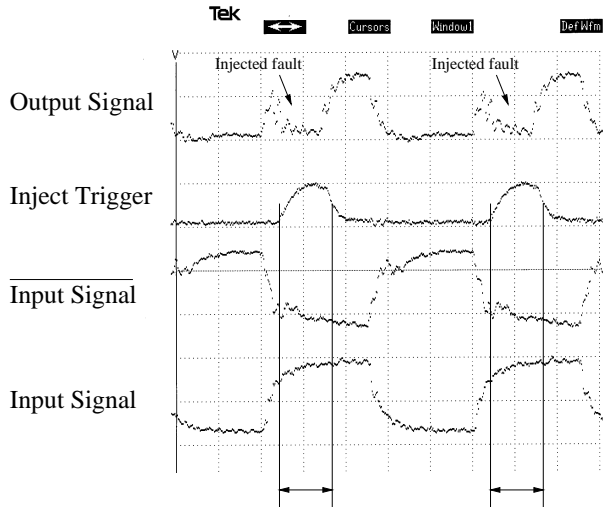
The fault injector performed up to its specification from a timing point of view. Total propagation delay was measured to be about 3ns, which is roughly the delay caused by one half of a meter of copper cable and well within the maximum of 10 meters allowable length for 1.28Gbps Myrinet [21]. Maximum 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.

Three injection methods have been tested in the current version of the fault injector:

**Command Trigger:** A command from the system connected through the RS-232 Interface 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.

**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 3. The indicated regions in Figure 3 are time periods where all nine lines meet injection criteria (the  $x$  axis is scaled at 5ns/division). 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.

**Fast Trigger:** 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 num-



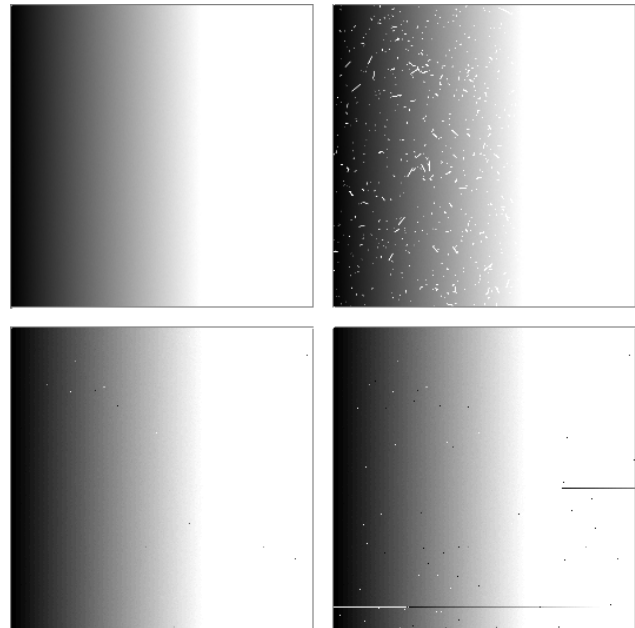
**Figure 3. Injection Triggered from Level Sensing Circuitry**

ber 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.

## 5.2. SWIFI Fault Injector and Space Imaging Application

In this fault injection campaign, NFTAPE injects memory faults into a scientific image processing application (Next Generation Space Telescope, an application from Jet Propulsion Laboratory (JPL-NASA) for cosmic rays suppression and data compression) executing as a single process on an Sparc Ultra-1 running Solaris 2.6. Since this application will operate in space, its developers are interested in analyzing how the program reacts to faults similar to those it will experience in space. This version of the program processes a series of 256x256 pixel image files (representing a noisy inputs from a sensor array) into a single filtered image. Since the system expects the sensor to frequently saturate from radiation in space, the algorithm first removes outliers it suspects were caused by radiation on the sensor. Next, the algorithm smoothes the image using the remaining points. The images in Figure 4 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 which is almost identical to the original image (observe several “dots” that indicate presence of the remaining noise), and (d) an example of a corrupt run.

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



**Figure 4. Images from Space Imaging Application: (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.**

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.

### 5.2.1 Proc.fi, /proc File System Fault Injector

The LWFI used for this experiment is called “proc.fi” because 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 fault injectors input parameters are the target process’s id and the fault location (stack, heap, or register). It can inject faults to the process’s register file or address space (stack, heap, or code). Before injecting a memory fault, it can query the system to find the process’s current stack and heap sizes.

The trigger, “proc.trig,” also uses the **/proc** file system to determine when to inject a fault. The trigger can stop the target program using a timer, using a breakpoint, or by tracing system calls. To trigger a fault, the trigger stops the process and sends a message to NFTAPE to inject a fault.

```

Injecting register 15 (07) bit 26 from value ef6e6544
...
pr_why: 0x6 reason='Faulted'(6) fault=Bounds(6) 'BUS'(11): 'unmapped addr'(1)
addr=eb6e654c trapno=0
pr_brkbase: 0010e54c pr_brksize: 566004 pr_stkbase: efff6000 pr_stksize: a000
G: 00000000 eb6e6544 04000000 00000000 00000000 00800000 00000000 00000000
O: 00000000 00020b40 fffffa80 00000001 00000000 00000000 effff4d0 eb6e654
L: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
I: 00079b00 00000000 0003be84 00000000 ef72227c ef6c679c effff530 ef6e643
PSR=fe401003 PC=eb6e654c nPC=eb6e6550 Y=00000000 WIM=00000000 TBR=00000000

```

**Figure 5. Example Excerpt from proc.fi Output.**

The first line describes the fault being injected, the next has data about the signal the fault generated. The last five lines are the 32 general-purpose registers and the 6 special-purpose registers).

**Table 1. Summary of Results from Experiment 1**

Location	Rate (flt/sec)	No Error	<= 25 Errors	> 25 Errors	No Output	Alignment Error	Unmapped Access	Permission Error	Illegal Instr.
memory	1	0	25	0	0	0	0	0	0
memory	2	0	24	0	1	1	0	0	0
memory	5	0	21	0	4	2	3	0	0
memory	8	0	18	1	6	7	0	0	0
memory	10	0	13	7	5	3	2	0	0
memory	15	0	2	11	12	3	9	0	0
memory	20	0	1	20	4	4	7	0	0
memory	25	0	1	7	17	4	14	0	0
memory	40	0	0	6	19	5	14	0	0
register	1	1	0	1	23	6	15	1	1
register	2	6	0	1	18	0	17	1	0
register	5	1	0	1	24	4	20	0	0

After the fault injector injects the fault, it sends a message back to NFTAPE and the trigger will allow the process to continue.

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.

Figure 5 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 experiment shows that the bus error resulted from an unmapped address and the offending address was `0xEB6E654C` (i.e., the corrupt value plus 8).

The results from this experiment are shown in Table 1. 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.

## 6. Conclusions

In this paper we described NFTAPE, a configurable environment for conducting automated fault injection experiments. Earlier fault injection tools exhibit shortcomings in portability, support for distributed systems, component reuse, and flexibility of injection method.

The component-based architecture made it easy to port NFTAPE to variety of platforms including Solaris, Linux, Windows, and Lynx (a real-time operating system). The architecture was designed primarily to run on distributed systems with the control part separated from the target nodes. By defining an interface for the NFTAPE components, NFTAPE can be easy used to perform a wide range of fault injection experiments on different platforms. In particular, NFTAPE reuses the mechanisms for communicating between components, controlling experiments, logging, and managing processes.

To illustrate the flexibility of NFTAPE, this paper describes two examples of fault injection campaigns. One uses a hardware fault injector to inject bit errors into the physical layer of a Myrinet LAN link. The second uses a debugger-based fault injector on a real space imaging application.

No other fault injection tool has shown as much flexibility as NFTAPE in being able to execute SWIFI, hardware-based fault injection, or simulation-based fault injections. Detailed NFTAPE performance numbers are not provided, as the current goal was to demonstrate that the proposed framework can function in a coherent manner. Several optimizations to the architecture are being designed and implemented, and the results should be soon forthcoming.

A long-term contribution that we expect from this work is a library of reusable components built by others for NFTAPE.

## References

- [1] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, "Fault injection for dependability validation: A methodology and some applications," *IEEE Trans. Software Eng.*, vol. 16, no. 2, pp. 166–182, 1990.
- [2] W. E. Baker, R. W. Horst, D. P. Sonnier, and W. J. Watson, "A flexible servernet-based fault-tolerant architecture," in *Proc. of the 25th Int'l Symp. on Fault-Tolerant Computing (FTCS-25)*, pp. 2–11, June 1995.
- [3] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W.-K. Su, "Myrinet: A gigabit-per-second local-area network," *IEEE Micro*, vol. 15, pp. 29–36, Jan. 1995.
- [4] F. Brasileiro, P. Ezhilchelvan, S. Shrivastava, N. Speirs, and S. Tao, "Implementing fail-silent nodes for distributed systems," *IEEE Trans. Computers*, vol. 45, pp. 1226–1236, Nov. 1996.
- [5] J. Carreira, H. Madeira, and J. G. Silva, "Xception: Software fault injection and monitoring in processor functional units," in *Proc. of the 5th IFIP Int'l Working Conf. Dependable Computing for Critical Applications (DCCA-5)*, pp. 135–149, Sept. 1995.
- [6] R. Chillarege and N. S. Bowen, "Understanding large system failures—a fault injection experiment," in *Proc. of the 19th Int'l Symp. on Fault-Tolerant Computing (FTCS-19)*, pp. 356–363, June 1989.
- [7] G. S. Choi and R. K. Iyer, "FOCUS: An experimental environment for fault sensitivity analysis," *IEEE Trans. Computers*, vol. 41, pp. 1515–1525, Dec. 1992.
- [8] S. Dawson, F. Jahanian, T. Mitton, and T.-L. Tung, "Testing of fault-tolerant and real-time distributed systems via protocol fault injection," in *Proc. of the 26th Int'l Symp. on Fault-Tolerant Computing (FTCS-26)*, pp. 404–414, June 1996.
- [9] K. K. Goswami, R. K. Iyer, and L. Young, "DEPEND: A simulation-based environment for system level dependability analysis," *IEEE Trans. Computers*, vol. 46, pp. 60–74, Jan. 1997.
- [10] R. K. Iyer and D. Tang, "Fault-tolerant computer system design," in *Experimental Analysis of Computer System Dependability* (D. K. Pradhan, ed.), Prentice Hall, 1996.
- [11] E. Jenn, J. Arlat, M. Rimén, J. Ohlsson, and J. Karlsson, "Fault injection into VHDL models: The MEFISTO tool," in *Proc. of the 24th Int'l Symp. on Fault-Tolerant Computing (FTCS-24)*, pp. 66–75, June 1994.
- [12] Z. Kalbarczyk, R. K. Iyer, S. Bagchi, and K. Whisnant, "Chameleon: A software infrastructure for adaptive fault tolerance," *IEEE Trans. Parallel and Distributed Systems*, June 1999.
- [13] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "FERRARI: A tool for the validation of system dependability properties," in *Proc. of the 22nd Int'l Symp. on Fault-Tolerant Computing (FTCS-22)*, pp. 336–344, July 1992.
- [14] N. A. Kanawati, G. A. Kanawati, and J. Abraham, "Dependability evolution using hybrid fault/error injection," in *Proc. of the Int'l Test Conf.*, pp. 224–233, Apr. 1995.
- [15] J. Karlsson *et al.*, "Using heavy-ion radiation to validate fault-handling mechanisms," *IEEE Micro*, vol. 14, pp. 25–40, Feb. 1994.
- [16] J. Lala, "Fault detection, isolation, and reconfiguration in FTMP: Methods and experimental results," in *Proc. of the 5th AIAA/IEEE Digital Avionics Systems Conf. (DASC)*, pp. 130–146, 1983.
- [17] J. R. Samson, W. Moreno, and F. Falquez, "A technique for automated validation of fault tolerant designs using laser fault injection (LFI)," in *Proc. of the 28th Int'l Symp. on Fault-Tolerant Computing (FTCS-28)*, pp. 162–167, June 1998.
- [18] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, A. R. F. Dancey, and T. Lin, "FIAT—fault injection based automated testing environment," in *Proc. of the 18th Int'l Symp. on Fault-Tolerant Computing (FTCS-18)*, pp. 102–107, June 1988.
- [19] D. T. Stott, G. Ries, M.-C. Hsueh, and R. K. Iyer, "Dependability analysis of a high speed network using software implemented fault injection and simulated fault injection," *IEEE Trans. Computers Special Issue on Dependable Computing*, vol. 47, pp. 108–119, Jan. 1998.
- [20] T. K. Tsai and R. K. Iyer, "An approach to benchmarking of fault-tolerant commercial systems," in *Proc. of the 26th Int'l Symp. on Fault-Tolerant Computing (FTCS-26)*, pp. 314–323, June 1996.
- [21] VMEbus Int'l Trade Assoc., *ANSI Standard ANSI/VITA 26-1998, American National Standard for Myrinet-on-VME Protocol Specification*, 1998.