April 2002

UILU-ENG-02-2202
CRHC-02-02

AN EXPERIMENTAL EVALUATION OF THE REE SIFT ENVIRONMENT FOR SPACEBORNE APPLICATIONS

K. Whisnant, R. K. Iyer, Z. Kalbarczyk, P. Jones

Coordinated Science Laboratory
1308 West Main Street, Urbana, IL 61801
University of Illinois at Urbana-Champaign

REPORT DOCUMENTATION PAGE

Form Approved
OMB NO. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE April 2002	3. REPORT TYPE AND DATES COVERED

4. TITLE AND SUBTITLE
An Experimental Evaluation of the REE SIFT Environment for Spaceborne Applications

5. FUNDING NUMBERS

6. AUTHOR(S)
K. Whisnant, R. K. Iyer, Z. Kalbarczyk, P. Jones

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(ES)
COORDINATED SCIENCE LABORATORY
1308 W. MAIN ST.
URBANA, IL 61801

8. PERFORMING ORGANIZATION REPORT NUMBER
UILU-ENG-02-2202
(CRHC-02-02)

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12 b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This paper presents an experimental evaluation of a software-implemented fault tolerance (SIFT) environment built around a set of self-checking processes called ARMORs running on different machines that provide error detection and recovery services to themselves and to spaceborne scientific applications. The experiments are split into three groups of error injections, with each group successively stressing the SIFT error detection and recovery more than the previous group. The results show that the SIFT environment adds negligible overhead to the application during failure-free runs. Only 28 cases from over 28,000 error injections were observed in which either the application failed to start or the SIFT environment failed to recognize that the application had completed. Further investigations showed that assertions within the SIFT processescoupled with object-based incremental checkpointingwere effective in preventing system failures by protecting dynamic data within the SIFT processes.

14. SUBJECT TERMS fault injection, distributed computing, software-implemented fault tolerance

15. NUMBER IF PAGES
36

16. PRICE CODE

17. SECURITY CLASSIFICATION OR REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

An Experimental Evaluation of the REE SIFT Environment for Spaceborne Applications

K. Whisnant, R.K. Iyer, Z. Kalbarczyk, P. Jones
Center for Reliable and High-Performance Computing
University of Illinois at Urbana-Champaign
1308 W. Main St.; Urbana, IL 61801
E-mail: {kwhisnan,iyer,kalbar,ph-jones}@crhc.uiuc.edu

Technical Report CRHC-02-02

April 2002

Abstract

Few distributed software-implemented fault tolerance (SIFT) environments have been experimentally evaluated using substantial applications to show that they protect both themselves and the applications from errors. This paper presents an experimental evaluation of a SIFT environment used to oversee spaceborne applications as part of the Remote Exploration and Experimentation (REE) program at the Jet Propulsion Laboratory. The SIFT environment is built around a set of self-checking ARMOR processes running on different machines that provide error detection and recovery services to themselves and to the REE applications.

An evaluation methodology is presented in which over 28,000 errors were injected into both the SIFT processes and two representative REE applications. The experiments were split into three groups of error injections, with each group successively stressing the SIFT error detection and recovery more than the previous group. The results show that the SIFT environment added negligible overhead to the application's execution time during failure-free runs. Correlated failures affecting a SIFT process and application process are possible, but the division of detection and recovery responsibilities in the SIFT environment allows it to recover from these multiple failure scenarios. Only 28 cases were observed in which either the application failed to start or the SIFT environment failed to recognize that the application had completed. Further investigations showed that assertions within the SIFT processes—coupled with object-based incremental checkpointing—were effective in preventing system failures by protecting dynamic data within the SIFT processes.

1 Introduction

In traditional spaceborne applications, onboard instruments collect and transmit raw data back to Earth for processing. The amount of science that can be done is clearly limited by the telemetry bandwidth to Earth. Processing the complete set of raw data on ground, however, can be time consuming. The Remote Exploration and Experimentation (REE) project intends to use a cluster of commercial off-the-shelf (COTS) processors to analyze the data onboard and send only the results back to Earth. This approach not only saves downlink bandwidth, but also provides the possibility of making real-time, application-oriented decisions.

While failures in the scientific applications are not critical to the spacecraft's health in this environment (spacecraft control is performed by a separate trusted computer), they can be expensive nonetheless. The commercial components used by REE are expected to experience a high rate of radiation-induced transient errors in space (ranging from one per day to several per hour), and downtime directly leads to the loss of scientific data. Hence, a fault-tolerant environment is needed to manage the REE applications. It is likely that the first experiment will continue to transmit the raw data to Earth while simultaneously using two to eight COTS processors to analyze the results. The goal is to ensure that the onboard analysis agrees with the analysis traditionally done on the ground, thus smoothing the transition to missions that exclusively use the REE platform for all computations.

The missions envisioned to take advantage of the SIFT environment for executing MPI-based [23] scientific applications include the Mars Rover, the Orbiting Thermal Imaging Spectrometer (OTIS), the Next-Generation Space Telescope (NGST), the Gamma Ray Large Area Space Telescope, and the Solar Terrestrial Probe. Although a complete set of requirements is closely dependent upon the particular characteristics of the scientific applications, some facts are clear:

- The SIFT environment must be able to detect and recover from its own crash and hang failures with minimal impact on application performance. A study of applications indicates that a performance impact of 5% or less is desirable.
- The SIFT environment must detect and recover application crashes and hangs.
- The SIFT environment must limit error propagation.
- Performance, power, and weight must be considered when designing SIFT mechanisms.

This paper presents an experimental evaluation of a SIFT environment constructed around Chameleon ARMOR processes [19] that provide error detection and recovery to themselves and to the REE applications. Applications are protected from crashes and hangs by *progress indicators*, a form of "I-m-alive" heartbeats used by the application to convey its progress to the SIFT environment. The ARMOR processes are protected through an object-based incremental checkpointing strategy called *microcheckpointing* and internal self-checking mechanisms.

An experimentation methodology was developed to progressively stress the error detection and recovery mechanisms of the SIFT environment while executing applications from the Mars Rover and OTIS missions. Three sets of injections were conducted, each set stressing the SIFT environment more than the previous:

1. *SIGINT/SIGSTOP injections.* Many faults are known to lead to crash and hang failures. A crash is defined as a process abnormally terminating. A hung process ceases to make progress or becomes unresponsive to input messages, but it does not terminate. The SIGINT/SIGSTOP injections reproduce these first-order effects of faults in a controlled manner that minimizes the possibility of error propagation or checkpoint corruption.
2. *Register and text-segment injections.* The next set of error injections represent common effects of single-event upsets by corrupting the state in the register set and text segment memory. This introduces the possibility of error propagation and checkpoint corruption.
3. *Heap injections.* Finally, the third set of experiments further broadened the failure scenarios by injecting errors in the dynamic heap data to maximize the possibility of error propagation. The results from these experiments were especially useful in evaluating the effectiveness of intraprocess self-checks in limiting error propagation.

2 REE Testbed and Applications

REE computational model. The REE computational model is shown in Figure 1. The trusted Spacecraft Control Computer (SCC) remains radiation-hardened (rad-hard) [21] and ultra-reliable; the SIFT environment is only for the scientific applications. The SCC schedules applications for execution on the REE cluster through the SIFT environment, possibly sharing the computational resources among several applications through multitasking.

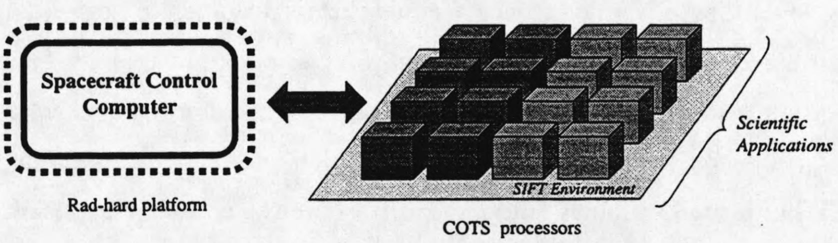


Figure 1: REE platform of SIFT-protected COTS components interfacing with rad-hard Spacecraft Control Computer

REE testbed configuration. The REE project developed several hardware testbeds ranging from 4 to 20 nodes for experimentation purposes. Since the early spaceborne experiments are expected to involve only a few processors, the experiments described in this paper were executed on 4- and 6-node systems. Figure 2 depicts the 4-node experimental testbed consisting of two boards (A and B), each with two PowerPC 750 processors running the Lynx real-time operating system. All processors communicate with each other through the Ethernet network, although the actual onboard computing platform is expected to use a high-speed interconnect such as Myrinet.

Between one and two megabytes of RAM on each processor were set aside to emulate local nonvolatile memory is available to each node. The nonvolatile RAM is expected to store temporary state information that must survive hardware reboots (e.g., checkpointing information needed during recovery). A remote file system on a Sun workstation stores program executables, application input data, and application output data. This storage device is intended to emulate nonvolatile memory visible to all nodes.

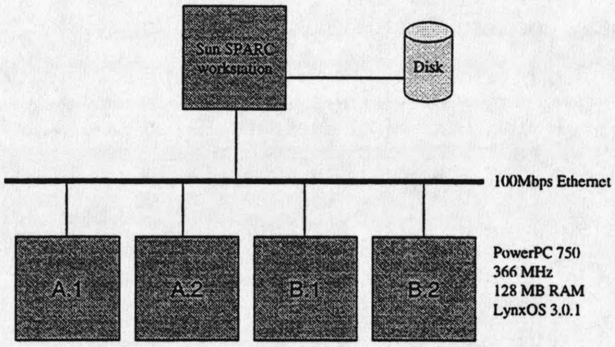


Figure 2: REE testbed configuration

REE applications. Two MPI-based applications are used in the experiments:

1. *Mars Rover texture analysis program* [7]. Cameras on the Mars Rover take images of the Martian surface and store the images on stable storage (the Sun workstation disk in the testbed). The program applies a series of filters to segment the image according to texture features. Three filters are used to extract vectors that describe image features along each of its three axes. A statistical clustering algorithm is applied to the feature vectors in order to segment the image (e.g., to distinguish between different rocks in the image). An output of the segmented image in feature vector space is written back to disk. The application takes rudimentary checkpoints by updating a status file after each filter completes. If the application restarts, it can skip filters that have already completed, but it must redo any filtering that was interrupted by the application failure. For the purposes of this experiment, the application executes on two nodes and analyzes one image per run.
2. *Orbiting Thermal Imaging Spectrometer (OTIS).* This application extracts land temperature and surface emissivities from thermal images taken from sensors. The program uses an algorithm to

compensate for atmospheric distortions in the thermal input images and an algorithm for data compression.

The primary focus of the experiments presented in this paper is the Mars Rover texture analysis program. Section 8 briefly examines both programs executing simultaneously on the REE testbed to investigate how the SIFT environment reacts to the added application load.

3 SIFT Environment

The REE applications are protected by a SIFT environment designed around a set of self-checking processes called ARMORS (Adaptive Reconfigurable Mobile Objects of Reliability) that execute on each node in the testbed. ARMORs control all operations in the SIFT environment and provide error detection and recovery to the application and to the ARMOR processes themselves. We provide a brief summary of the ARMOR-based SIFT environment as implemented for the REE applications; additional details of the general ARMOR architecture appear in [19].

3.1 SIFT Architecture

An ARMOR is a multithreaded process internally structured around objects called *elements* that contain their own private data and provide elementary functions or services (e.g., detection and recovery for remote ARMOR processes, internal self-checking mechanisms, or checkpointing support). Together, the elements constitute the functionality that defines an ARMOR's behavior. All ARMORs contain a basic set of elements that provide a core functionality, including the ability to (1) implement reliable point-to-point message communication between ARMORs, (2) communicate with the local daemon ARMOR process, (3) respond to "Are-you-alive?" messages from the local daemon, and (4) capture ARMOR state. Specific ARMORs extend this core functionality by adding extra elements.

Each ARMOR is addressed by a unique identification number, allowing messages to be sent to an ARMOR without prior knowledge of the ARMOR's physical location. ARMORs communicate solely through message passing, and messages are processed in separate threads within the ARMOR. A message consists of sequential events that trigger element actions. Elements subscribe to events that they are designed to process (e.g., an element can subscribe to an event that corresponds to the termination of the application), and an element's state can only be modified while processing message events. This modular, event-driven architecture permits the ARMOR's functionality and fault tolerance services to be customized by choosing the particular set of elements that make up the ARMOR.

Types of ARMORs. The SIFT environment for REE applications consists of four kinds of ARMOR processes: a Fault Tolerance Manager (FTM), a Heartbeat ARMOR, daemons, and Execution ARMORs

Fault Tolerance Manager (FTM). A single FTM executes on one of the nodes and is responsible for recovering from ARMOR and node failures as well as interfacing with the external Spacecraft Control Computer (SCC). The FTM contains all the basic ARMOR elements plus additional elements to (1) accept requests to execute applications from the SCC, (2) track resource usage of nodes in the SIFT environment, (3) send "Are-you-alive?" messages to daemons to detect node failures, (4) install Execution ARMORs for a particular application, (5) recover from failed subordinate ARMORs (i.e., Execution ARMORs and the Heartbeat ARMOR), (6) recover from node failures by migrating processes to another node, (7) recover from application failures, and (8) send application status information to SCC.

Heartbeat ARMOR. The Heartbeat ARMOR executes on a node separate from the FTM. Its sole responsibility is to detect and recover from failures in the FTM through the periodic polling for liveness. This functionality is implemented in a single element that is added to the Heartbeat ARMOR beyond the basic set of elements found in all ARMORs.

Daemons. Each node on the network executes a daemon process. Daemons are the gateways for ARMOR-to-ARMOR communication, and they detect failures in the local ARMORs. In addition to the core ARMOR configuration, the daemon contains elements that permit it to (1) install other ARMOR processes on the node, (2) communicate with local ARMORs, (3) cache location of remote ARMORs, (4) route messages to remote ARMORs, (5) send "Are-you-alive?" inquires to local ARMORs to detect hang failures, (6) detect crash failures in local ARMORs, (7) process "Are-you-alive?" inquires from the FTM, and (8) notify the FTM to initiate recovery of failed local ARMORs.

Execution ARMORs. Each application process is directly overseen by a local Execution ARMOR. In addition to the core set of elements, an Execution ARMOR contains elements to (1) launch application processes, (2) detect crash failures in application processes, (3) handle progress indicator updates from the application (to be described later), and (4) notify the FTM if the application process fails.

The ARMOR architecture permits the functionality of several ARMORs to be merged into a single process. For example, the functionality of the daemon and Execution ARMOR that execute on a node can be combined into a single ARMOR. Although this reduces the number of processes in the system, there are drawbacks to consolidating functionality. Complexity of the combined process is increased, thus increasing the probability of software design errors. Moreover, a single failure in the combined process will affect several more detection and recovery mechanisms than a single failure in which the mechanisms are distributed across multiple processes.

3.2 Executing REE Applications

Before executing any applications, the SCC first performs a one-time installation of the daemons, FTM, and Heartbeat ARMOR on the REE cluster. The SCC then launches applications through the SIFT environment, prompting the FTM to install Execution ARMORs on the appropriate nodes to support the application. Table 1 lists the steps involved in executing an MPI application, including the one-time installation of the SIFT environment. If the application executes perpetually, then the Execution ARMORs are never uninstalled; otherwise, they are removed from the SIFT environment after the application completes. If several applications are executed sequentially, then the FTM can reuse Execution ARMORs across applications.

Table 1: Steps in running an REE application in the SIFT environment

Initializing the SIFT environment:

1. The SCC issues commands that:
   a. Install daemon processes on each node that is to be part of the SIFT environment.
   b. Install the FTM process through a daemon on one of the nodes.
   c. Register all daemon processes with the FTM. The FTM instructs the daemon on the first registered node to install a Heartbeat ARMOR.

Preparing SIFT environment for executing applications:

2. The SCC submits the application to the FTM for execution, specifying the nodes on which it should execute.
3. The FTM instructs the appropriate daemons on the nodes to install Execution ARMORs, one for each prospective MPI process.

Executing the MPI application:

4. The FTM instructs one Execution ARMOR to launch the MPI process with rank 0. This process becomes the child of the Execution ARMOR.
5. The MPI process with rank 0—per the MPI implementation's protocol—remotely launches the remaining MPI processes on the other nodes.
6. The MPI process with rank 0 sends the process IDs of the other MPI processes to the appropriate Execution ARMORs via the FTM.
7. The Execution ARMORs for processes with ranks 1 through *n* establish communication channels with their respective MPI processes.
8. The application executes, periodically sending progress indicator updates to the local Execution ARMOR.
9. The FTM periodically heartbeats the registered daemons.
10. The Heartbeat ARMOR heartbeats the FTM.

Cleaning up after application completes:

11. The MPI processes terminate, notifying their local Execution ARMORs.
12. The Execution ARMOR for the rank 0 process forwards the application termination notification to the FTM.
13. Upon receiving all termination notifications, the FTM uninstalls the Execution ARMORs and reports to the SCC that the application has successfully completed.

Figure 3 illustrates a configuration of the SIFT environment with two MPI applications (from the Mars Rover and OTIS missions) executing on a four-node testbed. Arrows in the figure depict the relationships among the various processes (e.g., the application sends progress indicators to the Execution ARMORs, the FTM is responsible for recovering from failures in the Heartbeat ARMOR, and the FTM heartbeats the daemon processes). While the ARMORs can be distributed across the REE cluster in several ways, the FTM and Heartbeat ARMOR must reside on separate nodes to tolerate single-node failures. The entire SIFT environment can scale down to a minimal two-node configuration if necessary: the FTM executing on the first node, the Heartbeat ARMOR on the second, and the other ARMOR and application processes distributed across both nodes.

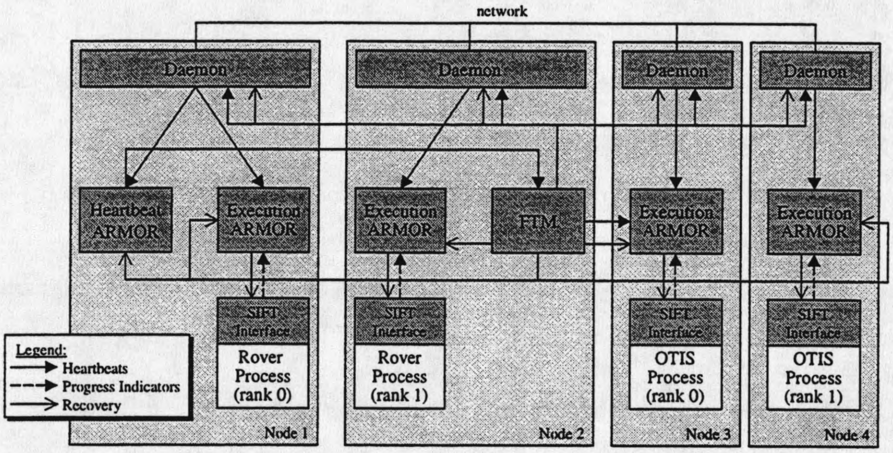


Figure 3: SIFT architecture for executing two MPI applications on a four-node network.

Each application process is linked with a SIFT interface that establishes a one-way communication channel with the local Execution ARMOR at application initialization. The application programmer can use this interface to invoke a variety of fault tolerance services provided by the ARMOR. The interface used for these experiments contains functions for initializing the communication channel, using progress indicators to detect application hangs, and closing the communication channel.

As described in Table 1, the Execution ARMORs, the Heartbeat ARMOR, and the FTM are children of their respective daemons. The MPI process with rank 0 is also a child of its Execution ARMOR. Because of the parent-child relationship, crash detection for child processes is implemented by having a thread within the parent process block on a `waitpid()` call to the operating system. Because the Execution ARMORs do not directly launch MPI processes with ranks 1 through *n*, crash failures in these MPI processes are also detected through other means, which are discussed in the next section.

3.3 Error Detection

The top-down error detection hierarchy consists of:

Node and daemon errors. The FTM periodically exchanges heartbeat messages with each daemon (every 10 s in our experiments) to detect node crashes and hangs. If the FTM does not receive a response by the next heartbeat round, it assumes that the node has failed. A daemon failure is treated as a node failure because the local ARMORs cannot communicate with other ARMORs in the environment if the daemon fails.

ARMOR errors. Each ARMOR contains a set of assertions on its internal state, including range checks, validity checks on data (e.g., a valid ARMOR ID), and data structure integrity checks. Besides assertions, other internal self-checks available to the ARMORs include preemptive control flow checking, I/O signature checking, and deadlock/livelock detection [2]. In order to limit error propagation, the ARMOR kills itself when an internal check detects an error. Because an ARMOR is a child of its daemon, the daemon detects crash failures in the ARMORs on the node via operating system calls. To detect hang failures, the daemon periodically (every 10 s in the experiments) sends "Are-you-alive?" messages to its local ARMORs.

REE applications. All application crash failures are detected by the local Execution ARMOR. Crash failures in the MPI process with rank 0 can be detected by the Execution ARMOR through operating system calls (i.e., `waitpid`). The other Execution ARMORs periodically check that their MPI processes (ranks 1 through *n*) are still in the operating system's process table. If not, it concludes that the application has crashed. An application process notifies the local Execution ARMOR through its communication channel before exiting normally so that the ARMOR does not misinterpret this exit as an abnormal termination.

A polling technique is used to detect application hangs in which the Execution ARMOR periodically checks for *progress indicator* updates sent by the application. A progress indicator is an "I'm-alive" message containing information that denotes application progress (e.g., a loop iteration counter). If the Execution ARMOR does not receive a progress indicator within an application-specific time period, the ARMOR concludes that the application process has hung. Since the texture analysis program executes functions in an external fast Fourier transform (FFT) library for about 20 s per filter, the Execution ARMOR cannot check for application progress more often than every 20 seconds. Finer checking granularity can be achieved by instrumenting the FFT functions with progress indicators. The application can also have internal checks as well such as algorithm-based fault tolerance (ABFT) to protect its computation [17] [14]. As with the ARMOR self-checks, the application kills itself if it cannot correct errors that are detected through internal checks.

3.4 Error Recovery

Nodes. The FTM migrates the ARMOR and application processes that were executing on the failed node to other working nodes in the SIFT environment.

ARMORs. Instead of consuming network bandwidth by reloading the ARMOR executable binaries to recover a failed ARMOR, the daemon copies its own executable image to the address space of the recovered ARMOR[1]. This is possible because all SIFT processes share a common ARMOR architecture.

[1] This is exactly what the `fork()` system call does on Unix platforms such as Lynx.

The recovered ARMOR is then configured by enabling and disabling the appropriate elements within the process (e.g., enabling Execution ARMOR elements while disabling the daemon-specific elements)[1].

To protect the ARMOR state against process failures, a checkpointing technique called *microcheckpointing* [36] is used. Microcheckpointing leverages the modular element composition of the ARMOR process to incrementally checkpoint state on an element-by-element basis. To process a message, an ARMOR sequentially delivers the events in the message to the elements that have subscribed to the events. After each event delivery, the state of the affected element is copied to a *checkpoint buffer* within the ARMOR process. Each element is assigned a disjoint region within the checkpoint buffer. Figure 4 illustrates the accumulation of state changes in the checkpoint buffer as a single thread processes a message. Several threads in the ARMOR can process messages in this way and update the checkpoint buffer without interference.

When the ARMOR decides to make the checkpoint permanent, it copies the checkpoint buffer to stable storage. In the REE testbed, checkpoints are committed to stable storage after every ARMOR message transmission. This ensures that the set of checkpoints in the system is always globally consistent. Thus, only a single process must be rolled back in the event of a failure.

The local RAM disk on each node serves as stable storage in the experiments unless otherwise noted. This configuration permits recovery from process failures. Tolerating node failures requires that the checkpoints be saved to a centralized location, such as separate nonvolatile memory that is independent of the failed node.

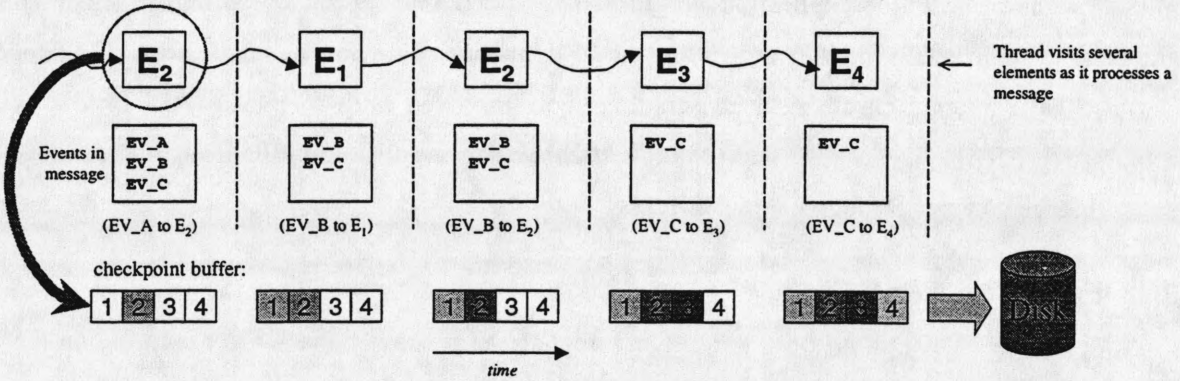


Figure 4: Microcheckpointing updates the ARMOR's checkpoint buffer as messages are processed

Nodes. The FTM migrates those ARMOR and application processes that were executing on the failed node to spare nodes.

[1] If the ARMOR repeatedly fails after being recovered in this manner, then the error may reside in the daemon's text segment, requiring that the ARMOR's image be reloaded from disk.

4 Injection Experiments

Error injection experiments into the application and SIFT processes were conducted to:

- Stress the detection and recovery mechanisms of the SIFT environment.
- Determine the failure dependencies among SIFT and application processes.
- Measure the SIFT environment overhead on application performance.
- Measure the overhead of recovering SIFT processes as seen by the application.
- Study the effects of error propagation and the effectiveness of internal self-checks in limiting error propagation.

The experiments used NFTAPE [32], a software framework for conducting injection experiments. NFTAPE separates the control, monitoring, and data collection aspects of injection experiments from the code that actually injects faults/errors. This design philosophy allowed us to use a different error injector for each error model while leaving the rest of the NFTAPE environment unchanged.

4.1 Error Models

The error models used the injection experiments represent a combination of those employed in several past experimental studies [13] and those proposed by JPL engineers [4]. Table 2 summarizes the error models used and the definition of failure for each model.

Table 2: Error models used in injection experiments

ERROR MODEL	DESCRIPTION	FAILURE DEFINITION
SIGINT	Lynx operating system delivers a SIGINT signal to the target process.	Target process terminates upon receipt of the SIGINT signal, simulating a crash failure.
SIGSTOP	Lynx operating system delivers a SIGSTOP signal to the target process.	All threads in the target process are suspended upon receipt of the SIGSTOP signal, simulating a process hang.
Register	Bits in the registers of the target process are periodically flipped until a failure is induced.	Target process can fail by crashing, hanging, or producing a detectably incorrect output. Failures can only be induced if a thread reads a corrupted register.
Text segment	Bits in the text segment of the target process are periodically flipped until a failure is induced.	Target process can fail by crashing, hanging, or producing a detectably incorrect output. Failures can only be induced if a corrupted instruction is executed or corrupted data in the text segment are read.
Heap	Bits in allocated regions of the heap memory in the target process are periodically flipped.	Target process can fail by crashing, hanging, or producing a detectably incorrect output. Failures can only be induced if corrupted data in the heap are read.

SIGINT/SIGSTOP. These signals were used to mimic "clean" crash and hang failures as described in the introduction.

Register and text-segment errors. Fault analysis has predicted that the most prevalent faults in the targeted spaceborne environment will be single-bit memory and register faults, although shrinking feature sizes have raised the likelihood of clock errors and multiple-bit flips in future technologies [4]. Since the experiments aimed at assessing the effectiveness of the SIFT environment in recovering from failures when they occur (as opposed to assessing coverage or likelihood of failure scenarios), register and text-segment errors were injected with the purpose of inducing failures. Several error injections were uniformly distributed within each run since each injection was unlikely to cause an immediate failure, and only the most frequently used registers and functions in the text segment were targeted for injection.

Heap errors. Heap injections were used to study the effects of error propagation. One error was injected per run into non-pointer data values only, and the effects of the error were traced through the system.

Errors were not injected into the operating system since our experience has shown that kernel injections typically led to a crash, led to a hang, or had no impact. Maderia et al. [22] used the same REE testbed to examine the impact of transient errors on LynxOS.

4.2 Definitions and Measurements

System. We use the term *system* to refer to the REE cluster and associated software (i.e., the SIFT environment and applications). The system does not include the rad-hard SCC or communication channel to the ground.

Experiment and run. An error injection *experiment* targeted a specific process (application process, FTM, Execution ARMOR, or Heartbeat ARMOR) using a specific error model from Table 2. For each process/error model pair, a series of *runs* were executed in which one or more errors were injected into the target process.

Activated errors. An injection causes an error to be introduced into the system (e.g., corruption at a selected memory location or corruption of the value in a register). An error is said to be *activated* if program execution accesses the erroneous value. Only activated errors can result in a failure.

Failures and system failures. A *failure* refers to a process deviating from its expected (correct) behavior as determined by a run without fault injection. The application can also fail by producing output that falls outside acceptable tolerance limits as defined by an external application-provided verification program.

A *system failure* occurs when either (1) the application cannot complete within a predefined timeout or (2) the SIFT environment cannot recognize that the application has completed successfully. These failures are caused by errors that propagate to an ARMOR's checkpoint or to other processes. System failures require that the SCC reinitialize the SIFT environment before continuing, but they do not threaten either the SCC or spacecraft integrity[1].

Recovery time. Recovery time is the interval between the time at which a failure is detected and the time at which the target process restarts. For ARMOR processes, this includes the time required to restore the ARMOR's state from checkpoint. In the case of an application failure, the time lost to rolling back to the most recent application checkpoint is accounted for in the application's total execution time, not in the recovery time for the application.

Perceived application execution time. The perceived execution time is the interval between the time at which the SCC submits an application for execution and the time at which the SIFT environment reports to the SCC that the application has completed.

Actual application execution time. The actual execution time is the interval between the start and the end of the application. The difference between perceived and actual execution time accounts for the time required to install the Execution ARMORs before running the application and the time required to uninstall the Execution ARMORs after the application completes (see Figure 5). This is a fixed overhead independent of the actual application execution time. The REE applications envisioned to take advantage of this environment are expected to be long-running, so the performance impact of the fixed overhead will be less apparent than in our testbed applications that use small input data sets. We differentiate between the perceived and actual execution times because it is important to assess how the SIFT environment responds to errors during the setup and takedown phases of an application's execution.

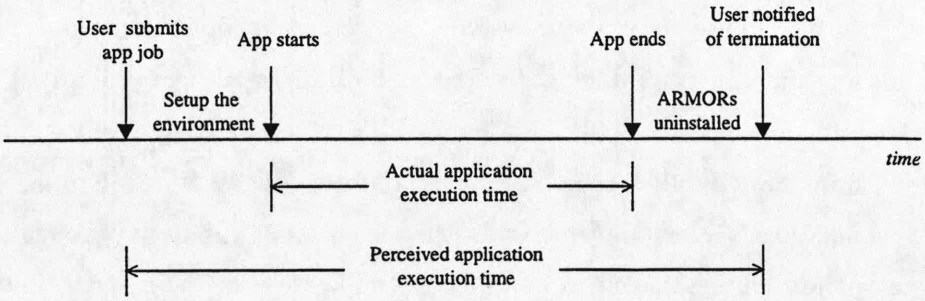

User submits app job
App starts
App ends
User notified of termination
Setup the environment
ARMORs uninstalled
time
Actual application execution time
Perceived application execution time

Figure 5: Perceived application execution time vs. actual application execution time

Baseline application execution time. In the injection experiments, the perceived and actual application execution times are compared to a baseline measurement in order to determine the performance overhead added by the SIFT environment and recovery. Two measures of baseline application performance are used: (1) the application executing without the SIFT environment and

[1] While the vast majority of failures in the SIFT environment will not affect the trusted SCC, in reality there exists a nonzero probability that the SCC can be impacted by SIFT failures. We discount this possibility in the paper because there is not a full-fledged SCC available for conducting such an analysis.

without fault injection, and (2) the application executing in the SIFT environment but without fault injection. The difference between these two measures provides the overhead that the SIFT processes impose on the application. Table 3 shows that the SIFT environment adds less than two seconds to the perceived application execution time. The actual execution time overhead is not statistically significant.

Table 3: Baseline application execution time without fault injection

	APP. EXEC. TIME (s)	
	PERCEIVED	ACTUAL
Application executing outside SIFT environment (Baseline No SIFT)	75.71 ± 0.65	75.71 ± 0.65
Application executing in SIFT environment (Baseline SIFT)	77.97 ± 0.48	75.74 ± 0.48

The sections that follow add a third measurement, namely the application execution time in the presence of failures and recovery. Comparing this measurement to the baseline measurement provides the amount of overhead (as seen by the application) in recovering from failures in the system. The mean application execution time and recovery time are calculated for each fault model. Ninety-five percent confidence intervals (t-distribution) are also calculated for all measurements.

5 Crash and Hang Failures

This section presents results from SIGINT and SIGTOP injections into the texture analysis application and SIFT processes that were used to evaluate the SIFT environment's ability to handle crash and hang failures. We first summarize the major findings from over 700 crash and hang injections:

- All injected errors into both the application and SIFT processes were recovered.
- Recovering from errors in SIFT processes imposed a mean overhead of 5% to the application's actual execution time. This 5% overhead takes into account 25 cases out of roughly 700 runs in which the application was forced to block or restart because of the unavailability of a SIFT process. Neglecting these cases, the overhead imposed by recovering SIFT processes is insignificant.
- Correlated failures involving some SIFT processes and the application were observed. Although they did not directly corrupt state, the crash and hang failures caused the SIFT processes to become unavailable for a period of time. In a few cases, this unavailability impacted the application processes that expect timely responses from the failed SIFT process, thus causing the application to fail as well. All correlated failures were successfully recovered.

- Recovery from the correlated failures was possible because the checking and recovery processes in the SIFT environment are decoupled from the entities involved in correlated failures.

SIGINT and SIGSTOP signals were injected at random intervals during the application's execution. Results for 100 runs per target are summarized in Table 4. In some cases, the injection time (used to determine when to inject the error) occurred after the application completed. For these runs, no error was injected.

Table 4: SIGINT/SIGSTOP injection results

TARGET	ERRORS INJECTED	SUCCESSFUL RECOVERIES	APP. EXEC. TIME (s)		RECOVERY TIME (s)
			Perceived	Actual	
			SIGINT		
Baseline	-	-	74.78 ± 0.55	72.68 ± 0.49	-
Application	100	100	89.80 ± 1.50	87.88 ± 1.50	0.48 ± 0.05
FTM	81	81	79.60 ± 1.61	73.89 ± 0.25	0.64 ± 0.16
Execution ARMOR	100	100	77.91 ± 1.01	75.98 ± 1.00	0.61 ± 0.07
Heartbeat ARMOR	97	97	75.26 ± 0.92	74.39 ± 0.96	0.47 ± 0.12
			SIGSTOP		
Baseline	-	-	71.96 ± 0.32	70.03 ± 0.27	-
Application	84	84	112.21 ± 1.87	110.21 ± 1.87	0.47 ± 0.05
FTM	97	97	76.20 ± 1.94	70.09 ± 0.88	0.79 ± 0.15
Execution ARMOR	98	98	85.01 ± 4.41	82.21 ± 4.28	0.63 ± 0.15
Heartbeat ARMOR	77	77	71.88 ± 0.24	70.24 ± 0.24	0.56 ± 0.21

Table 4 shows the experimental data for injections into both the application and SIFT processes. The first row shows the execution time for the application during failure-free runs (no injections). The results show that the SIFT environment was able to recover from all errors injected into the system. Even though no failed recoveries were observed, there still exists a nonzero probability that some failure scenarios will be unrecoverable. If p is the failure probability of a run (the parameter that we wish to estimate), then the probability of no failures occurring in n runs is $(1\text{-}p)^n$. In order to be 95% confident that no failures occur in n runs,

$$(1\text{-}p)^n \geq 0.95$$
$$1\text{-}p \geq (0.95)^{(1/n)}$$
$$p \leq 1\text{-}(0.95)^{(1/n)}$$

With $n = 734$ runs from the table, we can solve for p to show that less than 0.01% of all SIGINT/SIGSTOP failures will be unrecoverable.

5.1 Application Recovery

Table 4 shows that the application execution time under hang failures (SIGSTOP injections) is greater than the execution times under crash failures (SIGINT injections). Recall that hang failures are detected through a timeout, whereas application crashes can be detected almost immediately by the Execution ARMOR through operating system calls. The extra detection latency accounts for the difference between these two measurements.

As discussed in section 3.3, application hangs are detected using a polling technique. The Execution ARMOR executes a thread that wakes up every 20 seconds to check the value of a counter that is incremented by progress indicator messages sent by the application.

Because the Execution ARMOR polls the counter value at fixed intervals, the error detection latency for hangs can be up to twice the checking period. Figure 6 shows an example of the application updating its progress indicator from $c = 3$ to $c = 4$ before it hangs but after the Execution ARMOR has last checked the progress indicator value. At the next check, the Execution ARMOR sees the progress indicator has been updated to $c = 4$, so it concludes that the application has made progress during the last checking interval even though it has hung. Only on the next check does the Execution ARMOR see that the progress indicator is unchanged at $c = 4$. In the experiments, this phenomena can add up to 40 s to the application's execution time.

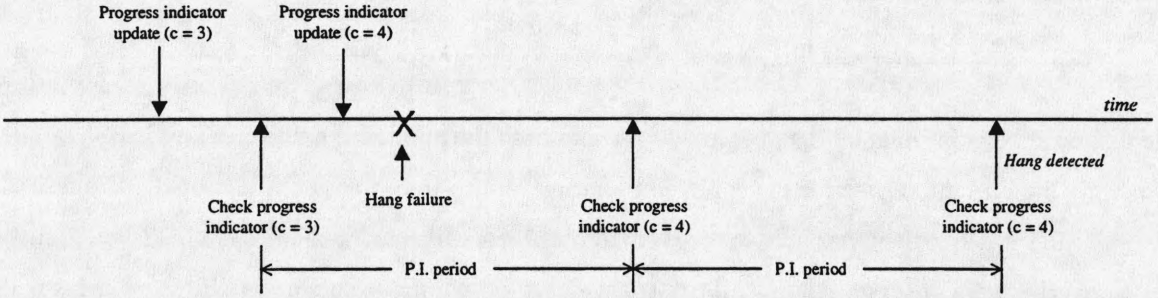

Progress indicator update (c = 3)
Progress indicator update (c = 4)
time
Hang detected
Hang failure
Check progress indicator (c = 3)
Check progress indicator (c = 4)
Check progress indicator (c = 4)
P.I. period
P.I. period

Figure 6: Application hangs detected through progress indicator

To reduce the hang detection latency, parameters of both the application and Execution ARMOR must be adjusted:

1. The application must frequently send progress indicator updates to the Execution ARMOR. To increase the rate, additional progress indicators must be added to the application code. This requires a good understanding of the application's execution behavior (e.g., knowing what functions are executed often, knowing inner loops that would be good candidates for placing progress indicators, etc.).
2. The Execution ARMOR must frequently check its counter updated by the progress indicators from the application. Clearly, the Execution ARMOR should not check the counter faster than the rate at

which the application sends progress indicator updates, as this leads to false alarms (detecting application hangs when there are none).

Detection latency can also be reduced by using an interrupt-driven design in which the progress indicator message from the application interrupts the checking thread in the Execution ARMOR to reset the thread's watchdog timer. By resetting the timer to expire 20 s from the last progress indicator update, any future hang will be detected within a 20-second window. Note that this interrupt-driven approach causes the thread updating the progress indicator counter to be coupled to the checking thread. This introduces additional thread interaction and complexity that is not present in the polling-based approach.

It should also be emphasized that the time lost by having to redo computation after a rollback is accounted for in the application's execution time, not in the application recovery time. In addition to rollback recovery, the REE applications are also expected to support forward recovery as well. The REE applications are designed to operate on new data each iteration cycle, so the application can either recompute the interrupted cycle or wait for new data in the next cycle when an error occurs. Our experiments assume the former, since input data is available for reprocessing when the application restarts. If the application is required to complete a fixed number of cycles before completing, however, the execution time will be the same on average for both rollback and forward recovery.

5.2 SIFT Environment Recovery

FTM. The perceived execution time for the application is extended if (1) the FTM fails while setting up the environment before the application execution begins or (2) the FTM fails while cleaning up the environment and notifying the Spacecraft Control Computer that the application terminated (see Figure 7). The application is decoupled from the FTM's execution after starting, so failures in the FTM do not affect it. The only overhead in actual execution time originates from the network contention during the FTM's recovery, which lasts for only 0.6-0.7 s.

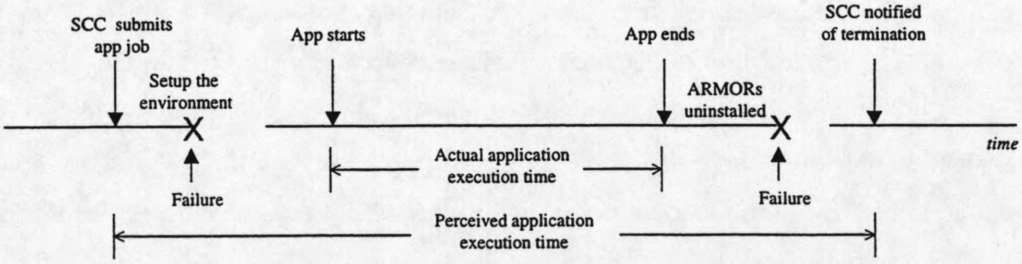

SCC submits app job
Setup the environment
App starts
App ends
SCC notified of termination
ARMORs uninstalled
time
Failure
Actual application execution time
Failure
Perceived application execution time

Figure 7: FTM failures in setup and takedown of SIFT processes affect perceived application execution time

FTM-application correlated failure. The error injections also revealed a correlated failure in which the FTM failure caused the application to restart in 2 of the 178 runs. Recall that during the setup phase the FTM installs an Execution ARMOR and the MPI process with rank 0 on the first node. The MPI process then installs the other MPI process on the second node. The rank 0 process sends the process ID of the other MPI process to the Execution ARMOR on the second node via the FTM. If the FTM fails during this period, then the rank 0 MPI process times out waiting for the other process to start (i.e., the MPI application aborts). Once the FTM recovers, the application is restarted.

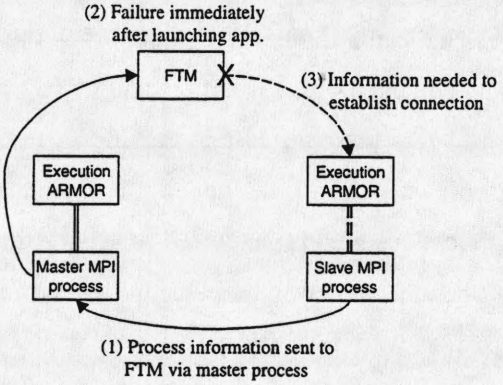

(2) Failure immediately after launching app.
FTM
(3) Information needed to establish connection
Execution ARMOR
Execution ARMOR
Master MPI process
Slave MPI process
(1) Process information sent to FTM via master process

Figure 8: Slave MPI process blocks because Execution ARMOR cannot get information from failed FTM

The SIFT environment is able to recover from this correlated failure because the components performing the detection (Heartbeat ARMOR detecting FTM failures and Execution ARMOR detecting application failures) are not affected by the failures. The Execution ARMOR resends the "application-failed" message to the FTM until it receives an acknowledgment. Once recovered, the FTM receives the Execution ARMOR's message and restarts the application.

Execution ARMOR. Of the 198 crash/hang errors injected into the Execution ARMORs, 175 required recovery only in the Execution ARMOR. For these runs, the application execution overhead was negligible. The overhead reported in Table 4 (up to 10% for hang failures) resulted from the remaining 23 cases in which the application was forced to restart.

Execution ARMOR-application correlated failure. If the application process attempted to contact the Execution ARMOR (e.g., to send progress indicator updates or to notify the Execution ARMOR that it is terminating normally) while the ARMOR was recovering, the application process blocked until the Execution ARMOR completely recovered. Because the MPI processes are tightly coupled, a correlated failure is possible if the Execution ARMOR overseeing the other MPI process diagnosed the blocking as an application hang and initiated recovery.

This correlated failure occurred most often when the Execution ARMOR hung (i.e., due to SIGSTOP injections): 22 correlated failures were due to SIGSTOP injections as opposed to 1 correlated failure resulting from an ARMOR crash (i.e., due to SIGINT injections). This is because Execution ARMOR crash failure is detected immediately by the daemon through operating system calls, making the Execution ARMOR unavailable for only a short time. The time for which the Execution ARMOR is unavailable from hangs, however, can be significant since hang failures are detected via 10-second heartbeat. Increasing the daemon-to-Execution ARMOR heartbeat frequency is one way to reduce the detection latency and, thus, improve Execution ARMOR availability. Care must be taken, however, not to lower the heartbeat period to a point at which false alarms are raised.

Modeling the probability of correlated failures. The likelihood of correlated failures depends upon the failure rate of the SIFT process and several performance parameters, including the frequency at which the application interfaces with the SIFT process, timeout used to detect application hangs, application recovery time, and SIFT recovery time. These factors can be incorporated into the stochastic activity network (SAN) shown in Figure 9, which models one application's behavior when attempting to interface with the local SIFT process.

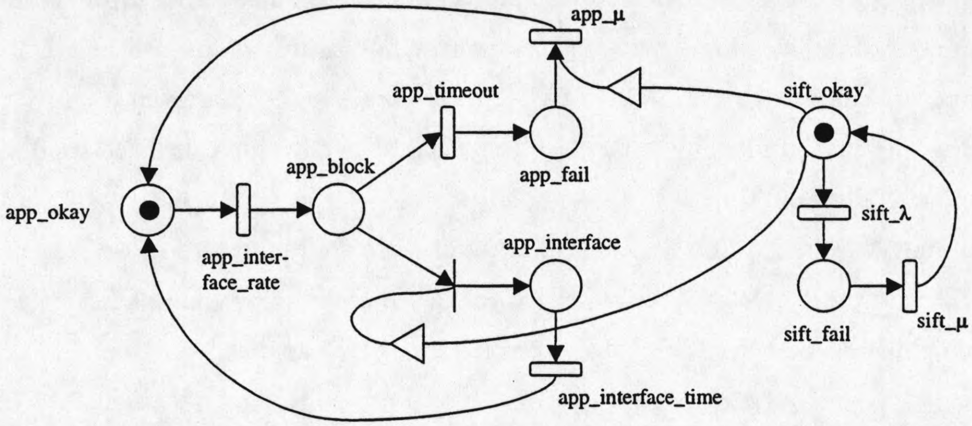


Figure 9: Stochastic activity network for modeling SIFT-induced application failures

The model begins with tokens in the `app_okay` and `sift_okay` places, indicating that both the application and SIFT process are operating normally. From these normal states, two independent activities are enabled:

1. The application can attempt to interface with the local SIFT process (e.g., to send a progress indicator update) through the `app_interface_rate` activity, placing the application in the `app_block` state.
2. The SIFT process can fail through the `sift_λ` activity, temporarily placing the SIFT process in the `sift_fail` state until it is recovered via the `sift_μ` activity.

If the SIFT process is in the `sift_okay` state, then the instantaneous activity leading to `app_interface` is enabled, causing the application to transition out of the `app_block` state. To simplify the model, it is assumed that SIFT failures do not affect the application once the application enters the `app_interface` state (i.e., once the SIFT process receives a request, it is able to send a reply to the application without failing).

If the application attempts to interface with the SIFT process while the SIFT process is unavailable, then the application remains in the `app_block` state until either (1) the SIFT process is recovered, at which time the instantaneous activity is re-enabled or (2) the application process times out waiting for the SIFT process, represented by the `app_timeout` activity. If the latter case occurs, the application transitions into the `app_fail` state until it is recovered via `app_μ` activity. Application recovery is conditioned on the SIFT process being in the non-failed state, since the SIFT process is responsible for detecting application failures and restarting the application process. Note that the application process does not independently fail in this model—all failures are induced by the SIFT process being unavailable to process application requests within an application-defined timeout period.

The potential for correlated failures is a concern for general designs [10], not just the SIFT environment presented in this paper. Although the probability is small that a SIFT process failure causes the application to fail as well[1], the impact on the application's unavailability can be significant. An experimental analysis on a high-availability VAX cluster data showed that even a small correlation between failures can cause availability to be much less than the availability calculated from models that assume uncorrelated failures [33].

Heartbeat ARMOR. Direct SIGINT/SIGSTOP injections into the Heartbeat ARMOR did not affect the application. The Heartbeat ARMOR was not needed for the setup or completion phases of a run, so its failure affected neither the perceived nor actual execution time of the application.

5.3 Impact of Heartbeat Frequency

This section examines the effect of varying the heartbeat frequency on the application execution time and recovery time. The FTM was chosen because its failures are detected solely through heartbeat timeouts. Results from this examination apply to detecting and recovering from ARMOR hang failures as well since these failures are also detected through heartbeats.

SIGINT injections into the FTM were repeated with varying heartbeat rates for both the Heartbeat ARMOR-to-FTM and FTM-to-daemon heartbeats. Results for these experiments are shown in Table 5. Each row provides the mean values based on 30 injection runs.[2]

The increased network load from frequent heartbeats does not have a significant impact on the actual application execution time (less than 1% comparing the 5 s heartbeat period to the 30 s period) since the

[1] Results from later sections show that 1.6% of all SIFT failures from register, text-segment, and heap injections induced failures in the application.

[2] These experiments were conducted on a different day than the experiments summarized in Table 4; thus, the measurements for the 10 s heartbeat period in Table 5 are different than the measurements for the FTM in Table 4.

heartbeat messages are small. The perceived application execution time improves with more frequent heartbeats because an FTM failure is detected more quickly.

It should be noted that the extra load during recovery may result in heartbeats not being processed in time if the heartbeat period is too small, thus raising false alarms that other processes have failed. Based upon experimentation, we chose to use 10-s heartbeats to eliminate the chance of a false alarm.

Table 5: Application execution time and recovery time with varying heartbeat periods

HEARTBEAT PERIOD (s)	APP. EXEC. TIME (s)	
	Perceived	Actual
5	77.93 ± 1.58	73.21 ± 0.47
10	80.60 ± 3.38	72.85 ± 0.25
20	86.72 ± 5.01	72.83 ± 0.23
30	96.68 ± 7.71	72.69 ± 0.29

6 Register and Text-Segment Injections

This section expands the scope of the injections to further stress the detection and recovery mechanisms by allowing for the possibility of checkpoint corruption and error propagation to another process. Results from approximately 9,000 single-bit errors into the register set and text segment of the application and SIFT processes show that:

- Most register and text-segment errors led to crash and hang failures that were recovered by the SIFT environment.
- Eleven of the approximately 700 observed failures led to system failures in which either the application did not complete or the SIFT environment did not detect that the application successfully completed. These 11 system failures resulted from injected errors that corrupted an ARMOR's checkpoint or propagated outside the injected process.
- Text-segment errors were more likely than register errors to lead to system failures. This was because values in registers typically had a shorter lifetime (i.e., they were either never used or quickly overwritten) when compared to information stored in the text segment.

Table 6 summarizes the results of approximately 6,000 register injections and 3,000 text-segment injections into both the application and ARMOR processes. The goal was to achieve between 90 and 100 error activations per target (represented by the "Failures" column) to obtain statistically stable results. Failures were classified into four categories:

1. **Segmentation fault.** An attempt is made to access an unmapped or invalid memory address. Corrupting a register that stores a pointer is a common cause for this type of failure. Segmentation faults should result in crash failures similar to those observed from SIGINT injections.

2. **Illegal instruction.** The processor attempts to execute an instruction with an invalid opcode. For example, an indirect jump or branch (i.e., jumping or branching to a location specified by the contents of a register) can fail due to corruption of the destination register. An illegal instruction exception should result in a crash failure similar to those observed from SIGINT injections.
3. **Hang.** The target process ceases to make progress. Hangs are detected through heartbeats and progress indicators as described in section 3.
4. **Assertion.** The error is detected through assertion checks within the ARMOR.

Table 6: Register and text-segment injection results

TARGET	FAILURES	SUC. REC.	FAILURE CLASSIFICATION				APP. EXEC. TIME (s)		REC. TIME (s)
			SEG. FAULT	ILLEGAL INSTR.	HANG	ASSERT.	PERCEIVED	ACTUAL	
Baseline	-	-	-	-	-	-	71.96 ± 0.32	70.03 ± 0.27	-
Register Injections									
Application	95	95	71	4	20	0	90.70 ± 2.57	88.81 ± 2.57	0.70 ± 0.21
FTM	84	84	58	6	16	4	75.65 ± 1.54	73.42 ± 1.28	0.71 ± 0.03
Execution ARMOR	80	77	56	6	15	3	76.19 ± 1.82	73.56 ± 1.83	0.45 ± 0.08
Heartbeat ARMOR	77	77	62	6	8	1	73.00 ± 0.22	70.66 ± 0.21	0.31 ± 0.04
Text-segment Injections									
Application	82	82	41	23	18	0	89.47 ± 2.87	87.49 ± 2.88	1.05 ± 0.33
FTM	88	84	53	28	5	2	76.47 ± 2.87	71.00 ± 2.31	0.51 ± 0.05
Execution ARMOR	95	93	45	31	11	8	77.48 ± 1.93	74.83 ± 1.86	0.43 ± 0.04
Heartbeat ARMOR	97	95	53	33	11	0	73.23 ± 0.37	71.21 ± 0.36	0.30 ± 0.01

In Table 6, the "Suc. Rec." column denotes the number errors from which the system successfully recovered. Errors that were not successfully recovered led to system failures. Table 6 shows that, unlike the SIGINIT/SIGSTOP injections, register and text-segment injections resulted in 11 system failures: 4 FTM failures, 5 Execution ARMOR failures, and 2 Heartbeat ARMOR failures.

6.1 SIFT Environment Recovery

FTM. Table 6 shows that the FTM successfully recovered from all register injections. Two text-segment injections were detected through assertions on the FTM's internal data structures, and both of these errors were recovered. The extent to which assertions prevent corrupted state from escaping the process is investigated via heap injections in section 7.

Table 6 also shows that the FTM could not recover from four text-segment errors. In each case, the error corrupted the FTM's checkpoint prior to crashing. Because the checkpoint was corrupted, the FTM crashed shortly after being recovered. This cycle of failure and recovery repeated until the run timed out.

There were seven cases of a correlated failure in which the FTM failed during the application's initialization: three from text-segment injections and four from register injections. Both the FTM and application recovered from all seven correlated failures.

FTM-daemon correlated error. Text-segment injections during the Execution ARMOR's initialization uncovered a race condition during the early experiments between the thread installing the ARMOR and the thread notifying the FTM of failure. This race condition prevented the FTM from recovering the failed Execution ARMOR.

Figure 10(a) illustrates the interactions between the FTM, the daemon, and the Execution ARMOR. The FTM first instructs the daemon to install an Execution ARMOR. After the process is spawned, the daemon sends an acknowledgement back to the FTM. This acknowledgment prompts the FTM to register the ARMOR (i.e., the FTM adds the Execution ARMOR to its list of subordinate ARMORs). Meanwhile, the daemon detects a failure in the Execution ARMOR and notifies the FTM. The FTM then initiates recovery by having the daemon reinstall the Execution ARMOR.

Under a different timing scenario, depicted in Figure 10(b), the failure notification from the daemon to the FTM reaches the FTM before the acknowledgment of the ARMOR installation. In this case, the FTM has no record of the Execution ARMOR, and the failure notification thread aborts. The acknowledgment later arrives, and the Execution ARMOR is registered. The daemon, not having received an acknowledgment for its failure notification message, eventually times out and resends the notification. The FTM detects this as a duplicate message and drops it before processing; thus, the Execution ARMOR is not recovered. This race condition was eliminated by adding the Execution ARMOR in the FTM's table before instructing the daemon to install the ARMOR.

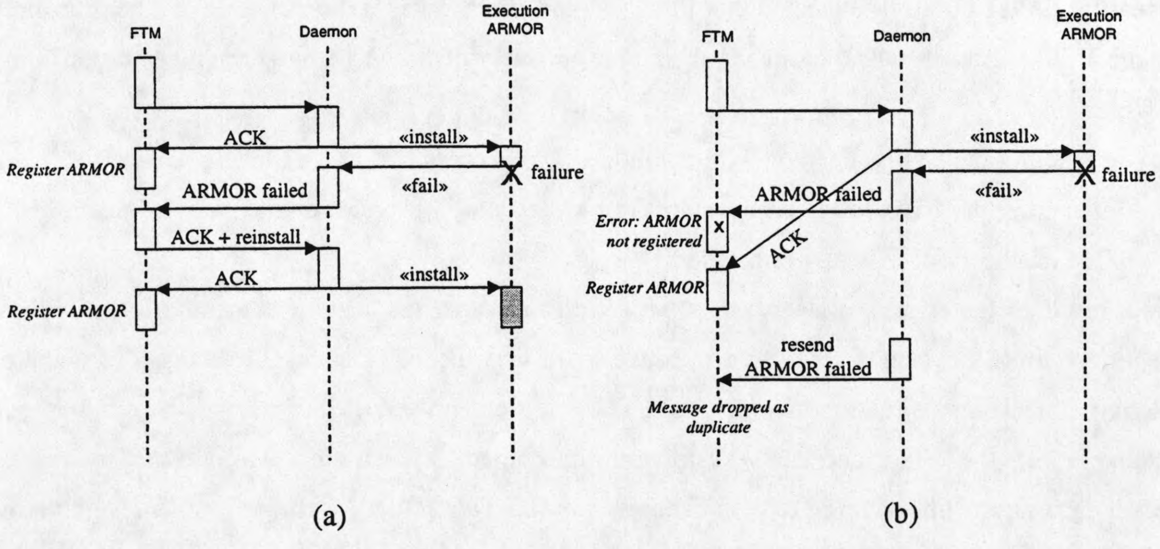


Figure 10: (a) Correct process interactions; (b) Process interactions arising from race condition

Execution ARMOR. There were three register injections and two text-segment injections into the Execution ARMOR that led to a system failure. In each of these cases, the error propagated to other ARMOR processes or to the Execution ARMOR's checkpoint:

1. One text-segment injection and three register injections caused errors in the Execution ARMOR to propagate to the FTM (i.e., the error was not fail-silent). Although the Execution ARMOR did not crash, it sent corrupted data to the FTM when the application terminated, causing the FTM to crash. The FTM state in its checkpoint was not affected by the error, so the FTM was able to recover to a valid state. Because the FTM did not complete processing the Execution ARMOR's notification message, the FTM did not send an acknowledgment back to the Execution ARMOR. The missing acknowledgment prompted the Execution ARMOR to resend the faulty message, which again caused the FTM to crash. This cycle of recovery followed by the retransmission of faulty data continued until the run timed out.
2. One of the text-segment injections caused the Execution ARMOR to save a corrupted checkpoint before crashing. When the ARMOR recovered, it restored its state from the faulty checkpoint and crashed shortly thereafter. This cycle repeated until the run timed out.

In addition to the system failures described above, three text-segment injections into the Execution ARMOR resulted in the restarting of the texture analysis application. All three of these correlated failures were successfully recovered.

Heartbeat ARMOR. The Heartbeat ARMOR recovered from all register errors as illustrated in Table 6, while text-segment injections brought about two system failures. Although no corrupted state escaped the Heartbeat ARMOR, the error prevented the Heartbeat ARMOR from receiving incoming messages. Thus, the Heartbeat ARMOR falsely detected that the FTM had failed, since it did not receive a heartbeat reply from the FTM. The ARMOR then began to initiate recovery of the FTM by attempting to perform the following steps:

1. Instructing the FTM's daemon to reinstall the FTM process.
2. Instructing the FTM to restore its state from checkpoint after receiving acknowledgment that the FTM has been successfully reinstalled.

As a result of the error, the Heartbeat ARMOR never received the acknowledgment in step two, thus preventing it from sending a follow-up message to restore the FTM state. Although the immediate problem (i.e., causing a situation in which the FTM is left unrecovered) can be solved by combining the reinstallation of the FTM and state restoration into a single operation without the intermediate acknowledgment, the underlying problem persists: the Heartbeat ARMOR suffers from receive omissions and will continue to detect a failed FTM during subsequent heartbeat rounds.

To detect the receive omission error, an element can be added to the Heartbeat ARMOR that performs a series of self-tests on key ARMOR functionality before the heartbeat messages are sent. These self-tests generate a signature, which can be verified by either the local daemon or by the receiving ARMOR. Additional error injection experiments can be used to evaluate the coverage of these additional self-checks on ARMOR functionality.

Among the successful recoveries from text-segment errors shown in Table 6, four involved corrupted heartbeat messages that caused the FTM to fail. Although faulty data escaped the Heartbeat ARMOR, the corrupted message did not compromise the FTM's checkpoint. Thus, the FTM was able to recover from these four failures.

7 Heap Injections

Finally, two sets of experiments are conducted to further broaden the scope of the injections by exclusively targeting the dynamic data stored in heap memory. The first set of experiments continues along the lines of the register and text segment injections: errors are repeatedly injected into the heap memory until the target process fails. The second set of experiments focuses the heap injections on specific subsets of data in heap memory in order to better examine error propagation and the impact of internal assertions in preventing system failures from error propagation.

7.1 SIFT Processes: Crash and Hang Failures

In the first set of experiments, all regions of the target's heap memory were candidates for error injection. Each of the 100 runs per target shown in Table 7 involved several injections to bring about a crash or hang failure; as a result, approximately 6,700 single-bit heap errors were injected across all targets. Even with the high injection rate, only about half of the 100 runs per target showed any effects on the system.

FTM. All manifested heap errors in the FTM were successfully recovered, including three instances in which the error propagated to the Execution ARMOR. The propagated errors caused the Execution ARMOR to crash when accepting progress indicator updates from the application.

Execution ARMOR. The application restarted on eight occasions in which the Execution ARMOR failed, causing the application execution time to be more than the baseline measurement. In all but one of 41 failure scenarios, the application completed successfully after the Execution ARMOR recovered. The one exception occurred because corrupted state in the Execution ARMOR prevented it from recognizing that the application had completed.

Heartbeat ARMOR. All errors were successfully handled by the SIFT environment. Three errors corrupted the heartbeat messages sent by the ARMOR to the FTM, causing the FTM to crash. In one run, corrupted message header data caused the Heartbeat ARMOR's daemon to crash. Because daemon failures are treated as node failures, the FTM migrated the Heartbeat ARMOR to another node. The application was able to complete in spite of the daemon failure and subsequent Heartbeat ARMOR migration.

Table 7: Heap injection results

TARGET	FAILURES	SUC. REC.	APP. EXEC. TIME (S)		RECOVERY TIME (S)
			PERCEIVED	ACTUAL	
Baseline	-	-	71.96 ± 0.32	70.03 ± 0.27	-
FTM	54	54	74.35 ± 2.08	72.40 ± 2.07	0.62 ± 0.05
Execution ARMOR	41	40	77.30 ± 2.93	75.33 ± 2.93	0.56 ± 0.09
Heartbeat ARMOR	28	28	71.58 ± 0.26	69.90 ± 0.26	0.31 ± 0.01

7.2 SIFT Processes: Targeted Injections into Heap Data

Injections described in the previous subsection did not discriminate as to the type of data injected. Data structures on the heap contain a mix of data fields that store information and pointers that connect the various items of the data structures (such as forward and backward pointers in doubly-linked lists). Careful examination of the experimental results showed that crash failures were most often caused by segmentation faults raised when a corrupted pointer was dereferenced. To maximize the chances of observing system failures from error propagation, a set of experiments was performed in which a only a single error in data (not pointers) was injected. There is a good chance that these data errors propagate and cause system failures, since dynamic data are often used either directly or indirectly[1] by the SIFT processes.

These experiments targeted the FTM because it contains the most state of all the ARMOR processes and because the FTM is used in all three phases of the run's execution (initialization of the SIFT environment, executing the application, and cleanup of the SIFT environment), thus giving more opportunities for system failures to result from escaped errors.

Results from these single injections into FTM heap memory are grouped by the element into which the error was injected (recall from section 3.1 that an ARMOR is composed of elements and that each element contains private state). Table 8 shows the number of system failures observed from 100 error injections per element, classified as to the their effect on the system.

Many data errors were detectable through internal assertions within the FTM, but not all assertions were effective in preventing system failures. One of four scenarios results after a data error is introduced:

[1] A `load` instruction is an example of an instruction directly accessing heap data. Indirect access of heap data include instructions that manipulate data in registers that were loaded from the heap.

1. The data error was not detected by an assertion and has no effect on the system. The application completes successfully as if there were no error.

Table 8: System failures observed through heap injections

SYSTEM FAILURE ELEMENT	UNABLE TO REGISTER DAEMONS	UNABLE TO INSTALL EXECUTION ARMORS	UNABLE TO START APPLICATION	UNABLE TO UNINSTALL EXECUTION ARMORS AFTER APPLICATION COMPLETES	TOTAL
`mgr_armor_info`. Stores information about subordinate armors such as location and element composition.	4	1	5	4	14
`exec_armor_info`. Stores information about each Execution armor such as status of subordinate application.	0	0	5	4	9
`app_param`. Stores information about application such as executable name, command-line arguments, and number of times application restarted.	0	0	0	0	0
`mgr_app_detect`. Used to detect that all processes for MPI application have terminated and to initiate recovery if necessary.	0	0	0	0	0
`node_mgmt`. Stores information about the nodes, including the resident daemon and hostname.	0	14	0	0	14
TOTAL	4	15	10	8	37

2. The data error was not detected by an assertion but led to a system failure. Because the SIFT process could not detect an incorrect condition in the system, only higher-level timeouts from the Spacecraft Control Computer can detect these situations (i.e., if the SCC does not receive application results within *x* seconds, then it can conclude that the submission request failed). Table 8 shows the phases of a run that are affected by system failures. None of the system failures impacted the application while it was executing—the failures either prevented the application from starting or prevented the SIFT environment from cleaning up after the application completed.
3. The data error was detected by an assertion check before propagating to the FTM's checkpoint or to another process. After an assertion fired, the FTM killed itself and recovered as if it had experienced an ordinary crash failure. By rolling back its state to a previous checkpoint, the error was removed from the FTM and normal execution resumed.
4. The data error was detected by an assertion check but after the error had propagated to the FTM's checkpoint or to another process. Rolling back the FTM's state in these circumstances was ineffective. System failures resulted from which the SIFT environment could not recover. These

results show that error latency is a factor when attempting to recover from errors in a distributed environment.

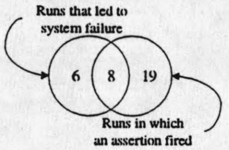
The injection results in Table 8 show that some state information was more sensitive to error propagation than others. The least sensitive elements were those modules whose state was substantially read-only after being written early within the run. With assertions in place, none of the data errors led to system failures. At the other end of the sensitivity spectrum, 28 errors in two elements caused system failures. In contrast with the elements causing no system failures, the data in `mgr_armor_info` and `node_mgmt` were repeatedly written to during the initialization phases of a run.

Table 9 show the efficiency of assertion checks in preventing system failures. The rightmost two columns in Table 9 represent the total number of runs in which assertions detected errors in a given element. For example, assertions in the `mgr_armor_info` element detected 27 errors, and 19 of those errors were successfully recovered (this information is depicted by the Venn diagram to the right of Table 9).

The data show that assertions coupled with the incremental microcheckpointing were able to prevent system failures in 58% of the cases (27 of 64 runs in which assertions fired). Recall that after an event within a message is processed by an element, only this element's state is copied to the checkpoint buffer. Incidental corruption to other elements (e.g., an error causing the event to overwrite another element's data) will not be saved to the checkpoint buffer. Thus, a clean copy of the corrupted element's state exists in the ARMOR's checkpoint for recovery as long as future events do not legitimately write to the corrupted element.

Table 9: Efficiency of assertion checks in preventing system failures

ELEMENT	SYSTEM FAILURES WITHOUT ASSERTION FIRING	SYSTEM FAILURES AFTER ASSERTION FIRES	SUCCESSFUL RECOVERY AFTER ASSERTION FIRES
`mgr_armor_info`	6	8	19
`exec_armor_info`	4	5	9
`App_param`	0	0	2
`Mgr_app_detect`	0	0	4
`node_mgmt`	0	14	3
TOTAL	10	27	37


Runs that led to system failure
6
8
19
Runs in which an assertion fired

On the other hand, Table 9 shows that assertions detected the error too late to prevent system failures in 27 cases. For example, 14 of the 17 runs in which assertions detected errors in the `node_mgmt` element resulted in system failures. This element translates hostnames into daemon IDs. When the SCC instructs the FTM to execute an application on a particular set of nodes, the FTM translates the hostnames to daemon IDs via the `node_mgmt` element. If the element cannot perform the translation, it uses a default daemon ID of zero for its response. The FTM attempts to send a message to the translated daemon ID, but it currently does not check to make sure that the returned daemon ID is nonzero. If the translation fails because of an error, the FTM's daemon detects that the message destination ID is invalid. The detection occurs too late, however, since the error already propagated outside the FTM.

7.3 Application Processes

A set of experiments was conducted in which one single-bit error was injected into the application during its lifetime. Few of the injected errors had any effect on the application.

Table 10 shows that for 1,000 injections into the application heap, 981 did not impact the application. Data on the heap were mostly floating point matrices, and single-bit flips in floating point variables often did not substantially change the value (only the precision). As a result, many of the heap errors did not cause the outputs to fall outside application-specific tolerance limits. Ten of the errors, however, affected the data enough to result in incorrect output.

Table 10: Results from 1,000 heap injections into the application

No effect (correct output)	981
Incorrect output	10
Crash	9
Hang	0

8 Multiple Applications

In this section, we briefly present results in which the Mars Rover and OTIS applications are executed simultaneously on a six-node testbed. This configuration was chosen so that each application process executes on a dedicated node; thus, the processor utilization for each node is comparable to that in the previous single-application experiments. These experiments demonstrate that having the SIFT environment control another application does not degrade the performance or dependability of the system.

Table 11 summarizes the mean performance characteristics of the two-application testbed across all targets and error models (SIGINT, SIGSTOP, register, and text segment). The first row shows the baseline execution of both applications simultaneously executing on the six-node testbed without the SIFT environment. The second row reports mean application execution time and recovery time when the master OTIS application is targeted for error injection, while the final row reports similar figures for injections into the SIFT processes averaged across all ARMORs (the FTM, Execution ARMOR for the master OTIS process, and Heartbeat ARMOR).

Table 11: Performance summary under error injection when running two applications simultaneously

TARGET	ROVER EXEC TIME (S)		OTIS EXEC TIME (S)		RECOVERY TIME (S)
	PERCEIVED	ACTUAL	PERCEIVED	ACTUAL	
Baseline (no SIFT)	-	151.30 ± 4.45	-	190.99 ± 1.01	-
OTIS app	142.51 ± 3.36	141.45 ± 3.34	225.18 ± 9.21	224.04 ± 9.21	0.39 ± 0.05
ARMORs	157.37 ± 2.21	156.17 ± 2.19	194.75 ± 0.92	193.45 ± 1.00	0.49 ± 0.02

The Mars Rover application execution time actually improved when the OTIS application was injected with errors. While the OTIS application was hung or recovering from an error, the Mars Rover application no longer contended with OTIS for network resources.

The last line in Table 11 shows that recovering from failures in the SIFT processes adds only 1-3% overhead the application baseline execution times. Two observations suggest that the SIFT environment adds a fixed amount of overhead to the system regardless of the application load:

- There is only a one-second difference between the perceived and actual application execution times. This represents the extra time spent installing and uninstalling the SIFT processes necessary to support the application. Note that this difference is comparable to the perceived/actual difference when running the Mars Rover application alone.
- ARMOR recovery time is similar to the ARMOR recovery time when running only one application. This indicates that the added application load does not impact recovery of the SIFT processes.

Injection results also showed that the kinds of errors observed were similar to those from the previous experiments. Table 12 groups the results from almost 11,000 injections according to error model: SIGINT/SIGSTOP and register/text-segment injections. The two rows marked "ARMORs" show cumulative numbers for all SIFT processes.

Table 12: Error classification when running two applications simultaneously

INJECTION TARGET	FAILURES	SUC. REC.	FAILURE CLASSIFICATION			
			SEG. FAULT	ILLEGAL INSTR.	HANG	SELF-CHECK
SIGINT/SIGSTOP Injections						
OTIS app	193	191	-	-	-	-
ARMORS	563	563	-	-	-	-
Register/Text-segment Injections						
OTIS app	194	194	147	27	10	10
ARMORS	566	552	397	85	78	6

All but two of the SIGINT/SIGSTOP errors were successfully recovered. The two that led to system failures were SIGSTOP injections into OTIS before the application had "created" its progress indicators[1]. Until the progress indicators are created, the Execution ARMOR cannot detect hang failures in the application. To remedy this situation, the Execution ARMOR can be designed to assume that the application has hung if progress indicators are not created within *x* seconds of starting unless the application tells the ARMOR otherwise. By assuming this, however, the application becomes less transparent to the SIFT environment (i.e., programmers *not* wanting to use progress indicators will be forced to make a function call in their programs to disable checking within the Execution ARMOR).

As with the single-application experiments, Table 12 shows that a majority of the register and text-segment errors resulted in crash (segmentation fault and illegal instruction exceptions) and hang failures. All but 14 of the 566 errors were recoverable. Of these recoverable errors, 25 were correlated failures involving a SIFT process and an application process. The 14 system failures were caused by errors that propagated either to the ARMOR's checkpoint or to another process. Text-segment errors caused 12 of the 14 system failures.

9 Lessons Learned

SIFT overhead should be kept small. System designers must be aware that SIFT solutions have the potential to degrade the performance and even the dependability of the applications they are intended to protect. Our experiments show that the functionality in SIFT can be distributed among several processes throughout the network so that the overhead imposed by the SIFT processes is insignificant while the application is running.

SIFT recovery time should be kept small. Minimizing the SIFT process recovery time is desirable from two standpoints: (1) recovering SIFT processes have the potential to affect application performance by contending for processor and network resources, and (2) applications requiring support from the SIFT environment are affected when SIFT processes become unavailable. Our results indicate that fully recovering a SIFT process takes approximately 0.5 s. The mean overhead as seen by the application from SIFT recovery is less than 5%, which takes into account 10 out of roughly 800 failures from register, text-segment and heap injections that caused the application to block or restart because of the unavailability of a SIFT process. The overhead from recovery is insignificant when these 10 cases are neglected.

SIFT/application interface should be kept simple. In any multiprocess SIFT design, some SIFT processes must be coupled to the application in order to provide error detection and recovery. The Execution ARMORs play this role in our SIFT environment. Because of this dependency, it is important to

[1] Before any progress indicators are sent, the application must tell the Execution ARMOR at what frequency to check for progress

make the Execution ARMORs as simple as possible. All recovery actions and those operations that affect the global system (such as job submission, preparing the node to execute an application, and detecting remote node failures) are delegated to a remote SIFT process that is decoupled from the application's execution. This strategy appears to work, as only 5 of 373 observed Execution ARMOR failures[1] led to system failures.

SIFT availability impacts the application. Low recovery time and aggressive checkpointing of the SIFT processes help minimize the SIFT environment downtime, making the environment available for processing application requests and for recovering from application failures.

If the SIFT environment cannot recover from a failure, then responsibility rests on the SCC or the ground station to recover the REE cluster. This externally controlled recovery, however, can be quite expensive in terms of application downtime, since the entire cluster must be diagnosed and reinitialized before restarting the SIFT environment. Downtime can be on the order of hours if not days under such scenarios if ground control is required, underscoring the need for rapid onboard detection and recovery.

System failures are not necessarily fatal. Only 28 of the approximate 28,000 injections resulted in a system failure in which the SIFT environment could not recover from the error. These system failures were not catastrophic in the sense of impacting the spacecraft or SCC. In fact, none affected an executing application.

To reduce the number of system failures, a timeout can be placed on the application connecting to the SIFT environment. Because the time between submission and connection is usually small, errors that occur in the critical phase of preparing the SIFT environment for a new application can be detected using this timeout without significant delay. Once the application starts, our experience has shown that it is well-protected and relatively immune to errors in the SIFT environment.

10 Related Work

Few experimental assessments of distributed fault tolerance environments have been undertaken. Three notable exceptions include:

MARS. Three types of physical fault injection (pin-level injections, heavy-ion radiation from a Californium-252 isotope, and electromagnetic interference) were used to study the fail silence coverage of the Maintainable Real-Time System (MARS) [20]. MARS achieved fail silence in these experiments through process duplication across nodes. A real-time control program was used as the test application

indicator updates. This is when progress indicators are "created" from the perspective of the Execution ARMOR.

[1] SIGINT, SIGSTOP, register, and text-segment injections caused 100, 98, 80, and 95 failures, respectively, in the experiments involving only the texture analysis program.

for these experiments. A later study compared software-implemented fault injection to the three physical injection approaches [13].

Delta-4. Pin-level injections were performed to evaluate the fail silence coverage of the Delta-4 atomic multicast protocol [1]. Fail silence was achieved by designing network interface cards around duplicated hardware on which the atomic multicast protocol executes.

Hades. Software-implemented fault injectors were used to inject errors into the Chorus microkernel and the Hades middleware, a collection of run-time services for real-time applications executing on COTS processors [8]. This experiment evaluated the coverage of the Hades error detection mechanisms while running an object-tracking application.

It is not clear if any of these studies validated how well the fault tolerance environment recovers from its own errors or how such errors impact performance. All were primarily interested in showing that the environment's error detection and masking were sufficient to maintain fail silence.

The overall research into software-implemented fault tolerance is summarized in Table 13. Each of the related works can be characterized by its support for (1) an external environment for managing application and handling failures and (2) services that are incorporated within the application to provide fault tolerance. Table 13 specifies the support in these two areas, plus the experiments done to evaluate the dependability of the SIFT solutions and the applications used in the evaluations.

Apart from the specific cases mentioned above, none of the work in Table 13 has been evaluated using a substantial application. Most use either synthetic benchmarks or a program with the complexity on the order of an echo server. It is difficult to evaluate the SIFT environment's ability to handle correlated failures and error propagation when the application process interactions—including interactions involving other application processes or the SIFT processes—are simple and infrequent.

Finally, few of the SIFT solutions presented in Table 13 have utilized extensive fault injection to demonstrate that their infrastructures are fault-tolerant. Some have undergone testing in which the user kills processes from the command line, but few have gone beyond using crash and hang failures to validate functionality. As our experiments have shown, injections into the text segment, registers, and heap were required to see correlated failures, error propagation, corrupted checkpoints, and system failures.

Table 13: Summary of related work

WORK	ENVIRONMENT TO MANAGE FAILURES	FAULT TOLERANCE PROVIDED TO APPLICATION	APPLICATION USED AND/OR EVALUATION
AQuA [9]	Fault masking through replication.	Intercepts remote CORBA method invocations and multicasts to replicas through Ensemble. Various replication strategies (active with pass-first reply, active with majority voting, passive) available to CORBA objects.	Measured replication overhead using echo server and board game.
Arjuna [30]	No external support.	Object-oriented framework for constructing replicated objects around atomic actions.	Distributed database constructed using Arjuna [6].
Cactus [16]	No external support.	Dependability protocols (such as reliable multicasting, voting) built from set of microprotocols.	Measured response time of checking account object using protocols.
CoCheck [31]	No external support.	Synchronous checkpointing of MPI application.	None.
Delta-4 [27]	Fault management software to recover from detected errors, reintegrate failed replicas, diagnose failures.	Active and passive replication, atomic multicast communication among replicas.	Hardware fault injection to evaluate coverage of atomic multicast protocol.
Eternal [25]	Reintegration of failed replica using state from good replica.	Intercepts remote CORBA method invocations and multicasts to replicas through Totem. Various replication strategies (cold, warm, active) available to CORBA objects.	Measured performance of echo server and a "packet-driver" (a degenerate echo server).
FRIENDS [11]	No external support.	Metalevel supports primary-backup replication, leader-follower replication, authentication, and multicasting.	Measured of performance overhead for object instantiation and invocation using a checking account application.
FTCT [18]	Agent processes oversee execution of applications on local node. Recovery actions and cluster management coordinated by replicated central manager.	Recovery from node crashes and hangs.	Performance measurements of replicated manager without application.
FT-MPI [12]	No external support.	API for managing failures in MPI application.	None.
GUARDS [28]	Middleware for fault tolerance and integrity management.	Replication, voting, clock synchronization.	Petri net models of three typical instantiations of the architecture.
Hades [8]	Either shutdown the node or invoke app-defined exception after detecting error; no explicit support for recovery.	Error detection mechanisms; active, passive, and semi-active replication of application tasks.	Fault injection into operating system and middleware while running an object tracking application.
Isis [5], Horus [34], Totem [24], Ensemble [15]	No external support.	Group communication services.	Used as foundation for several other research projects (e.g., Eternal, AQuA).
MARS [13]	No external support.	Active replication.	Physical fault injection [20] and software fault injection [13] while running real-time control program

11 Conclusion

This paper has presented a series of experiments in which the error detection and recovery mechanisms of a distributed SIFT environment have been stressed through over 28,000 error injections into a Mars Rover texture analysis program and the SIFT processes themselves. The results show that:

1. Structuring the fault injection experiments to progressively stress the error detection and recovery mechanisms is a useful approach to evaluating performance and error propagation in distributed SIFT environments.
2. Even though the probability for correlated failures (those SIFT failures that lead to application resets) was small, the potential impact on application availability was significant. When the correlated failure scenarios were not considered, the application experienced virtually no overhead due to SIFT recovery. When the correlated failures were taken into account, the mean overhead on application execution time rose to 5%.

3. The SIFT environment successfully recovered from all correlated failures involving the application and a SIFT process because the processes performing error detection and recovery were decoupled from the failed processes. This was due to the fact that SIFT functionality not directly related to monitoring and interfacing with the application was delegated to remote processes, thus insulating the application from a substantial number of SIFT errors.
4. Targeted injections into dynamic data on the heap were useful in further investigating system failures brought about by error propagation. These injections were biased to produce as many error propagations as possible: only non-pointer values were injected, and injections were limited to specific modules within the SIFT process to better trace the error effects. Assertions within the SIFT processes were shown to reduce the number of system failures from data error propagation by up to 42%. This suggests that detection mechanisms can be incorporated into the common ARMOR infrastructure to preemptively check for errors before state changes occur within the SIFT processes, thus decreasing the probability of error propagation and checkpoint corruption.

12 References

[1] J. Arlat, M. Aguera, Y. Crouzet, J.-C. Fabre, E. Martins, and D. Powell, "Experimental evaluation of the fault tolerance of an atomic multicast system," in *IEEE Transactions on Reliability*, vol. 39, no. 4, pp. 455-467, October 1990.

[2] S. Bagchi, "Hierarchical error detection in a software-implemented fault tolerance (SIFT) environment," Ph.D. Thesis, University of Illinois, Urbana, IL, 2001.

[3] R. Batchu, et al., "MPI/FT: Architecture and taxonomies for fault-tolerant, message-passing middleware for performance-portable parallel computing," in *Proceedings of the First International Symposium of Cluster Computing and the Grid*, pp. 26-33, 2001.

[4] J. Beahan, et al., "Detailed radiation fault modeling of the remove exploration and experimentation (REE) first generation testbed architecture," in *Proceedings of the IEEE Aerospace Conference*, vol. 5, pp. 279-281, 2000.

[5] K. Birman and R. van Renesse, Reliable Distributed Computing with the Isis Toolkit. Los Alamitos, CA: *IEEE Computer Society Press*, 1994.

[6] L. Buzato and A. Calsavara, "Stabilis: A case study in writing fault-tolerant distributed applications using persistent objects," Technical Report 400, University of Newcastle upon Tyne, Newcastle on Tyne, United Kingdom, 1992.

[7] F. Chen, et al., "Demonstration of the Remote Exploration and Experimentation (REE) fault-tolerant parallel-processing supercomputer for spacecraft onboard scientific data processing," in *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 367-372, 2000.

[8] P. Chevocot and I. Puaut, "Experimental evaluation of the fail-silent behavior of a distributed real-time run-time support build from COTS components," in *Proceedings of the 2001 International Conference on Dependable Systems and Networks*, pp. 304-313, 2001.

[9] M. Cukier et al., "AQuA: An adaptive architecture that provides dependable distributed objects," in *Proceedings of the 17th Symposium on Reliable Distributed Systems*, pp. 245-253, 1998.

[10] J. Dugan, "Correlated hardware failures in redundant systems," in *Proceedings of the 2nd Dependable Computing for Critical Applications Conference*, 1991.

[11] J.-C. Fabre and T. Pérennou, "A metaobject architecture for fault-tolerant distributed systems: The FRIENDS approach," *IEEE Transactions on Computers*, vol. 47, no. 1, pp. 78-95, January 1998.

[12] G. Fagg and J. Dongarra, "FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world," *Lecture Notes in Computer Science*, vol. 1908, Springer-Verilag: Berlin, pp. 346-353, 2000.

[13] E. Fuchs, "Validating the fail-silence assumption of the MARS architecture," in *Proceedings of the 6th Dependable Computing for Critical Applications Conference*, pp. 225-247, 1998.

[14] J. Gunnels, D. Katz, E. Quintana-Ortí, and R. van de Geijn, "Fault-tolerant high-performance matrix multiplication: theory and practice," in *Proceedings of the 2001 International Conference on Dependable Systems and Networks*, pp. 47-56, 2001.

[15] M. Hayden, "The Ensemble system," Ph.D. thesis, Cornell University, Ithaca, NY, 1988.

[16] J. He, M. Rajagopalan, M. Hiltunen, R. Schlichting, "Providing QoS customization in distributed object systems," in *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms*, pp. 351-372, 2001.

[17] K. Huang and J. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Transactions on Computers*, vol. 33, no. 6, pp. 518-528, 1984.

[18] M. Li, D. Goldberg, W. Tao, and Y. Tamir, "Fault-tolerant cluster management for reliable high-performance computing," in *Proceedings of the 13th Conference on Parallel and Distributed Computing and Systems*, pp. 480-485, 2001.

[19] Z. Kalbarczyk, S. Bagchi, K. Whisnant, and R. Iyer, "Chameleon: A software infrastructure for adaptive fault tolerance," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 6, pp. 560-579, 1999.

[20] J. Karlsson, J. Arlat, and G. Leber, "Application of three physical fault injection techniques to the experimental assessment of the MARS architecture," in *Proceedings of the 5th Dependable Computing for Critical Applications Conference*, pp. 150-161, 1995.

[21] S. Kerns, et al., "The design of radiation-hardened ICs for space: A compendium of approaches," *Proceedings of the IEEE*, vol. 76, no. 11, pp. 1470-1509, November 1988.

[22] H. Maderia, R. Some, F. Moereira, D. Costa, D. Rennels, "Experimental evaluation of a COTS system for space applications," in *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, 2002.

[23] Message Passing Interface Forum, "MPI-2: Extensions to the Message Passing Interface," http://www.mpi-forum.org/docs/mpi-20.ps.

[24] L. Moser, P. Melliar-Smith, D. Agarwala, R. Budhia, and C. Lingley-Papadopoulos, "Totem: A fault-tolerant multicast group communication system," Communications of the ACM, vol. 39, pp. 54-63, April 1996.

[25] L. Moser, P. Melliar-Smith, and P. Narasimhan, "A fault tolerance framework for CORBA," in *Proceedings of the 29th Symposium on Fault-Tolerant Computing*, pp. 150-157, 1999.

[26] P. Narasimhan, L. Moser, and P. Melliar-Smith, "State synchronization and recovery for strongly consistent replicated CORBA objects," in *Proceedigns of the 2001 International Conference on Dependable Systems and Networks*, pp. 261-270, 2001.

[27] D. Powell, P. Verissimo, G. Bonn, F. Waeselynck, and D. Seaton, "The Delta-4 approach to dependability in open distributed computing systems," in *Proceedings of the 18th International Symposium on Fault-Tolerant Computing*, pp. 246-251, 1988.

[28] D. Powell, et al., "GUARDS: A Generic upgradable architecture for real-time dependable systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 6, pp. 580-599, 1999.

[29] J. Ren, "AQuA: A framework for providing adaptive fault tolerance to distributed applications," Ph.D. Thesis, University of Illinois, Urbana, IL, 2001.

[30] S. Shrivastava, "Lessons learned from building and using the Arjuna distributed programming system," *Lecture Notes in Computer Science*, vol. 938, Springer-Verilag, Berlin, 1995.

[31] G. Stellner, "CoCheck: Checkpointing and process migration for MPI," in *Proceedings of the 10th International Parallel Processing Symposium*, pp. 526-531, 1996.

[32] D. Stott, B. Floering, Z. Kalbarczyk, and R. Iyer, "Dependability assessment in distributed systems with lightweight fault injectors in NFTAPE," in *Proceedings of the 4th International Computer Performance and Dependability Symposium*, pp. 91-100, 2000.

[33] D. Tang and R. Iyer, "Analysis and modeling of correlated failures in multicomputer systems," *IEEE Transactions on Computers*, vol. 41, no. 5, pp. 567-577, May 1992.

[34] R. van Renesse, K. Birman, and S. Maffeis, "Horus: A flexible group communication system," *Communications of the ACM*, vol. 39, pp. 76-83, April 1996.

[35] J. Wensley, "SIFT: Software implemented fault tolerance," in *Proceedings of AIPS*, vol. 41, pp. 243-253, 1971.

[36] K. Whisnant, Z. Kalbarczyk, and R. Iyer, "Micro-checkpointing: Checkpointing for multithreaded applications," in *Proceedings of the 6th International On-Line Testing Workshop*, July 2000.