An Experimental Evaluation of the REE SIFT Environment for Spaceborne Applications

K. Whisnant, R.K. Iyer, P. Jones
University of Illinois
Urbana, IL
{kwhisnan,iyer,ph-jones}@crhc.uiuc.edu

R. Some
Jet Propulsion Laboratory
Pasadena, CA
raphael.r.some@jpl.nasa.gov

D. Rennels
University of California
Los Angeles, CA
rennels@cs.ucla.edu

Abstract

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 11 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 at JPL 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 helping to smooth the transition to missions that use the REE platform exclusively for all computations.

The missions envisioned to take advantage of the SIFT environment for executing MPI-based [16] 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. Applications will execute only in simplex mode because resource constraints generally preclude replication.

This paper presents a methodology for experimentally evaluating a distributed SIFT environment executing an REE texture analysis program from the Mars Rover mission. Errors are injected so that the consequences of faults can be studied. The experiments do not attempt to analyze the cause of the errors or fault coverage. Rather, the error injections progressively stress the detection and recovery mechanisms of the SIFT environment:

1. *SIGINT/SIGSTOP injections.* Many faults are known to lead to crash and hang failures[1]. 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.* The third set of experiments further broaden the failure scenarios by injecting errors in the dynamic heap data to maximize the possibility of error propagation. The results from these experiments are especially useful in evaluating how well intraprocess self-checks limit error propagation.

REE computational model. The REE computational model consists of a trusted, radiation-hardened (rad-hard) Spacecraft Control Computer (SCC) and a cluster of COTS processors that execute the SIFT environment and 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.

[1] A crashed process terminates abnormally. A hung process ceases to make progress or becomes unresponsive to input messages, but it does not terminate.

REE testbed configuration. The experiments described in this paper were executed on a 4-node testbed consisting of PowerPC 750 processors running the Lynx real-time operating system. Nodes are connected through 100 Mbps Ethernet in the testbed, 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 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). Nonvolatile memory visible to all nodes is emulated by a remote file system residing on a Sun workstation that stores program executables, application input data, and application output data.

REE MPI application. A Mars Rover texture analysis program [5] was used as the workload application (results for executing an application from the OTIS mission in tandem with the texture analysis program can be found in [24]). Cameras on the Mars Rover take images of the Martian surface and store the images on stable storage. The program applies a series of filters to segment the image according to texture features. Three filters 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), and 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. The application executes on two nodes and analyzes one image per run for the purposes of these experiments.

2 SIFT Environment for REE

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 [12].

2.1 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 heartbeats 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.

2.2 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.

Figure 1 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.

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.

2.3 Error Detection Hierarchy

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.

Table 1: Steps for running an REE application

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.

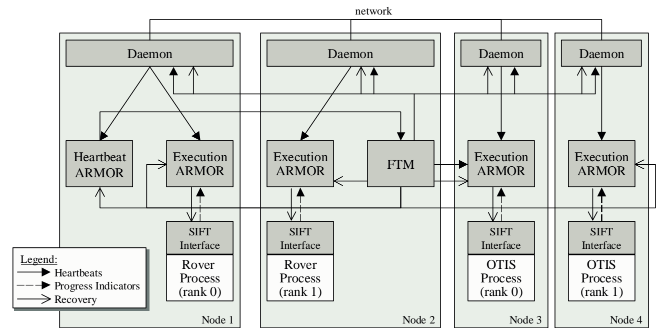


Figure 1: SIFT architecture for executing two MPI applications on a four-node network.

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. 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. 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 [10]. As with the ARMOR self-checks, the application kills itself if it cannot correct errors that are detected through internal checks.

2.4 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. This is possible because all SIFT processes share a common ARMOR architecture. 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)[2].

To protect the ARMOR state against process failures, a checkpointing technique called *microcheckpointing* [23] 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. Because each element is assigned a disjoint region within the checkpoint buffer and because an element only processes one event at a time, several threads can concurrently update the checkpoint buffer without interference.

When the ARMOR decides to make the checkpoint permanent, it copies the checkpoint buffer to the memory region that emulates local nonvolatile RAM. Data stored in the nonvolatile RAM survives node resets and is available for process recovery. To tolerate node failures, the checkpoints must be stored in a location that is independent of the failed node. In the experimental implementation, checkpoints are saved after every ARMOR message transmission to ensure that the set of ARMOR checkpoints in the system is always globally consistent; thus, only a single process must be rolled back in the event of an ARMOR failure.

REE Applications. On detecting an application failure, the Execution ARMOR notifies the FTM to initiate recovery. The version of MPI used by JPL on the REE testbed precludes individual MPI processes from being restarted within an application; therefore, the FTM instructs all Execution ARMORs to terminate their MPI processes before restarting the application. The application executable binaries must be reloaded from the remote disk during recovery.

3 Injection Experiments

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

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

The experiments used NFTAPE [22], a software framework for conducting injection campaigns.

3.1 Error Models

The error models used the injection experiments represent a combination of those employed in several past experimental studies [9] and those proposed by JPL engineers [4].

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. [15] used the same REE testbed to examine the impact of transient errors on LynxOS.

3.2 Definitions and Measurements

System, experiment, and run. We use the term *system* to refer to the REE cluster and associated software (i.e., the SIFT

[2] 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.

environment and applications). The system does not include the radiation-hardened SCC or communication channel to the ground. An error injection *experiment* targeted a specific process (application process, FTM, Execution ARMOR, or Heartbeat ARMOR) using a particular error model. 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 and failures. 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. 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 the SCC or spacecraft integrity[3].

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 2). 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.

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 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 2 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. 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 gives the amount of overhead as seen by the application due to recovery.

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.

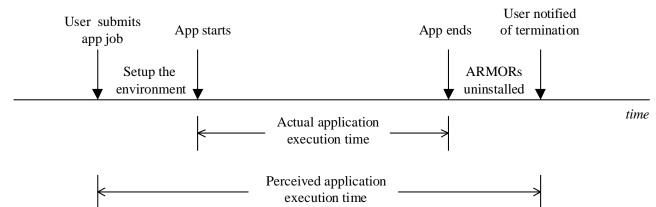


Figure 2: Perceived vs. actual execution time

Table 2: Baseline application execution time

	Perceived	Actual
Without SIFT	75.71 ± 0.65	75.71 ± 0.65
With SIFT	77.97 ± 0.48	75.74 ± 0.48

4 Crash and Hang Failures

This section presents results from SIGINT and SIGSTOP injections into the application and SIFT processes, which 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 includes 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 those cases in which the application must redo lost computation, the overhead imposed by a recovering SIFT process was insignificant.
- Correlated failures involving a SIFT process and the application were observed. In 25 cases, crash and hang failures caused a SIFT process to become unavailable, prompting the application to fail when it did not receive a timely response from the failed SIFT process. All correlated failures were successfully recovered.

Results for 100 runs per target are summarized in Table 3. 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. The row "Baseline" reports

[3] 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.

the application execution time with no fault injection[4]. One hundred runs were chosen in order to ensure that failures occurred throughout the various phases of an application's execution (including an idle SIFT environment before application execution, application submission and initialization, application execution, application termination, and subsequent cleanup of the SIFT environment).

Table 3: SIGINT/SIGSTOP injection results

Target	Failures	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

4.1 Application Recovery

Hangs are the most expensive application failures in terms of lost processing time. As discussed in section 2.3, application hangs are detected using a polling technique in which the Execution ARMOR executes a thread that wakes up every 20 seconds to check the value of a counter incremented by progress indicator messages sent by the application. Because the counter is polled at fixed intervals, the error detection latency for hangs can be up to twice the checking period[5]. This latency can be decreased by instrumenting the application with progress indicators at a finer granularity, but the unavailability of source code for some of the libraries used by the Mars Rover application preclude fine-grained instrumentation.

In addition to rollback recovery, the REE applications are expected to support forward recovery. 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.

4.2 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. 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.

An 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.

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 3 (up to 10% for hang failures) resulted from the remaining 23 cases in which the application was forced to restart.

An 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 an Execution ARMOR crash failure is detected immediately by the daemon through operating system calls, making the Execution ARMOR unavailable for only a short time. Hangs, however, are detected via a 10-second heartbeat. Although increasing the daemon-to-Execution ARMOR heartbeat frequency can reduce the detection latency, care must be taken to avoid false alarms.

5 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:

[4] Although the processing boards were reserved for our experiments, the remote disk was shared with other users. Approximately 30 baseline runs were conducted between each set of experiments for the fault model, and the average baseline measurements are reported for each fault model. When experiments across fault models were run during a timeframe in which the external workload was relatively constant, only one baseline measurement is given.

[5] Consider the case in which the application reports progress immediately after the last check by the Execution ARMOR and then hangs. Progress will appear to have been made during the next time by the Execution ARMOR check—only during the second check from the hang will the Execution ARMOR truly detect that no progress has been made.

- 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 4 summarizes the results of approximately 6,000 register injections and 3,000 text-segment injections into both the application and ARMOR processes. Failures are classified into four categories: segmentation faults, illegal instructions, hangs, and errors detected via assertions. The second column in Table 4 gives the number of successful recoveries vs. the number of failures for each set of experiments. Errors that were not successfully recovered led to system failures (4 due to FTM failures, 5 due to Execution ARMOR failures, and 2 due to Heartbeat ARMOR failures).

Table 4: Register and text-segment injection results

Target	Recoveries/ Failures	Failure Classification				App. Exec. Time (s)		Recoverry Time (s)
		Seg. fault	Illegal instr.	Hang	Assert-ion	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	77 / 80	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	84 / 88	53	28	5	2	76.47 ± 2.87	71.00 ± 2.31	0.51 ± 0.05
Execution ARMOR	93/ 95	45	31	11	8	77.48 ± 1.93	74.83 ± 1.86	0.43 ± 0.04
Heartbeat ARMOR	95 / 97	53	33	11	0	73.23 ± 0.37	71.21 ± 0.36	0.30 ± 0.01

FTM recovery. Table 4 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 6.

Table 4 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 the application recovered from all seven correlated failures.

Execution ARMOR recovery. Three register injections and two text-segment injections into the Execution ARMOR led to system failure. In each of these cases, the error propagated to other ARMOR processes or to the Execution ARMOR's checkpoint.

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.

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.

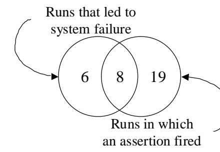
Heartbeat ARMOR recovery. The Heartbeat ARMOR recovered from all register errors, while text-segment injections brought about two system failures. Although no corrupted state escaped the Heartbeat ARMOR, the error prevented the Heartbeat ARMOR from receiving incoming messages. Thus, the Heartbeat ARMOR 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 (1) instructing the FTM's daemon to reinstall the FTM process, and (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. 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. However, 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

Table 5: System failures observed through heap injections

Element	Effect on System				System Failures			#4
	A	B	C	D	Total	#2	#3	
`mgr_armor_info`. Stores information about subordinate ARMORs such as location and element composition.	4	1	5	4	14	6	8	19
`exec_armor_info`. Stores information about each Execution ARMOR such as status of subordinate application.	0	0	5	4	9	4	5	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	0	0	2
`agr_app_detect`. Used to detect that all processes for MPI application have terminated and to initiate recovery if necessary.	0	0	0	0	0	0	0	4
`node_mgmt`. Stores information about the nodes, including the resident daemon and hostname.	0	14	0	0	14	0	14	3
TOTAL	4	15	10	8	37	10	27	37



Legend (Effect on system):
(A) Unable to register daemons.
(B) Unable to install Execution ARMORs.
(C) Unable to start applications.
(D) Unable to uninstall Execution ARMORs after application completes.

Legend (System failure/assertion check classification):
(2) System failure without assertion firing.
(3) System failure with assertion firing.
(4) Successful recoveries after assertion fired.

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 4, 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.

6 Heap Injections

Careful examination of the register injection experiments showed that crash failures were most often caused by segmentation faults raised from dereferencing a corrupted pointer. To maximize the chances for error propagation, only data (not pointers) were injected on the heap. Results from targeted injections into FTM heap memory were grouped by the element into which the error was injected. Table 5 shows the number of system failures observed from 100 error injections per element, classified as to the their effect on the system. One hundred targeted injections were sufficient to observe either escaped or detected errors given the amount of state in each element; overall, 500 heap injections were conducted on the FTM.

Many data errors were detectable through assertions within the FTM, but not all assertions were effective in preventing system failures. One of four scenarios resulted after a data error was injected (the last three columns in Table 5 are numbered to refer to scenarios 2-4):

1. The data error was not detected by an assertion and had no effect on the system. The application completed successfully as if there were no error.
2. The data error was not detected by an assertion but led to a system failure. None of the system failures impacted the application while it was executing.
3. The data error was detected by an assertion check, but only 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, and system failures resulted from which the SIFT environment could not recover. These cases show that error latency is a factor when attempting to recover from errors in a distributed environment.
4. 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.

The injection results in Table 5 show that some state information was more sensitive to error propagation than other. The least sensitive elements (`app_param` and `mgr_app_detect`) 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 during the initialization phases of a run.

Table 5 also shows the efficiency of assertion checks in preventing system failures. The rightmost two columns in the table represent the total number of runs in which assertions detected errors. For example, assertions in the `mgr_armor_info` element detected 27 errors, and 19 of those errors were successfully recovered. The Venn diagram to the right of the first row depicts the relationship between the set of runs experiencing system failure and the set of runs in which an assertion fired.

The data also 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.

On the other hand, 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 at the time of these experiments it did not check to make sure that the returned daemon ID is nonzero. If the translation failed because of an error, the FTM's daemon detected that the message destination ID was invalid. The detection occurred too late, however, since the error already propagated outside the FTM. This problem was rectified by adding checks to the translation results before sending the message.

7 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 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[6] 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 11 of the 10,000 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.

8 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) [13]. MARS achieved fail silence in these experiments through process duplication across nodes. A real-time control program was used as the test application for these experiments. A later study compared software-implemented fault injection to the three physical injection approaches [9].

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 [6]. 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.

In addition to the three environments presented above, there are several other projects involved in providing software-implemented fault tolerance. AQuA [7] and Eternal [17] replicate CORBA objects. Arjuna [20] achieves reliability through transactions and replication. GUARDS [19] provides a generic framework for fault tolerance and integrity management. CoCheck [21], FT-MPI [8], and MPI/FT [3] cater specifically to MPI applications by offering synchronous checkpointing, extended MPI function semantics for error handling, and replication, respectively. Finally, FTCT [11]

[6] SIGINT, SIGSTOP, register, and text-segment injections caused 100, 98, 80, and 95 failures, respectively.

adds fault tolerance to cluster management by replicating a central manager.

None of these environments 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 since the application process interactions—including those with other application processes and with the SIFT processes—are simple and infrequent.

Finally, few of these SIFT solutions 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 are required to see correlated failures, error propagation, corrupted checkpoints, and system failures.

9 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 10,000 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.
2. Even though the probability for correlated failures is small, its potential impact on application availability is significant.
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.
4. Targeted injections into dynamic data on the heap were useful in further investigating system failures brought about by error propagation. 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.

Acknowledgments

This work was supported in part by a NASA/JPL contract 961345 and by NSF grants CCR 00-86096 ITR and CCR 99-02026.

References

[1] J. Arlat, et al., "Experimental evaluation of the fault tolerance of an atomic multicast system," in *IEEE Trans. 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] F. Chen, et al., "Demonstration of the Remote Exploration and Experimentation (REE) fault-tolerant parallel-processing supercomputer for spacecraft onboard scientific data processing," in *DSN-00*, pp. 367-372, 2000.

[6] 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 *DSN-01,* pp. 304-313, 2001.

[7] M. Cukier, et al., "AQuA: An adaptive architecture that provides dependable distributed objects," in *SRDS-17*, pp. 245-253, 1998.

[8] 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.

[9] E. Fuchs, "Validating the fail-silence assumption of the MARS architecture," in *DCCA-6*, pp. 225-247, 1998.

[10] J. Gunnels, D. Katz, E. Quintana-Ortí, and R. van de Geijn, "Fault-tolerant high-performance matrix multiplication: theory and practice," in *DSN-01,* pp. 47-56, 2001.

[11] 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.

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

[13] J. Karlsson, J. Arlat, and G. Leber, "Application of three physical fault injection techniques to the experimental assessment of the MARS architecture," in *DCCA-5,* pp. 150-161, 1995.

[14] 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.

[15] H. Maderia, R. Some, F. Moereira, D. Costa, D. Rennels, "Experimental evaluation of a COTS system for space applications," in *DSN-02,* 2002.

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

[17] L. Moser, P. Melliar-Smith, and P. Narasimhan, "A fault tolerance framework for CORBA," in *FTCS-29*, pp. 150-157, 1999.

[18] D. Powell, P. Verissimo, G. Bonn, F. Waeselynck, and D. Seaton, "The Delta-4 approach to dependability in open distributed computing systems," in *FTCS-18*, pp. 246-251, 1988.

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

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

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

[22] D. Stott, B. Floering, Z. Kalbarczyk, and R. Iyer, "Dependability assessment in distributed systems with lightweight fault injectors in NFTAPE," in *IPDS-00*, pp. 91-100, 2000.

[23] 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.

[24] K. Whisnant, R. Iyer, Z. Kalbarczyk, P. Jones, "An Experimental Evaluation of the ARMOR-based REE Software-Implemented Fault Tolerance Environment," pending technical report, University of Illinois, Urbana, IL, 2001.