November 2000

UILU-ENG-00-2214
CRHC-00-03

University of Illinois at Urbana-Champaign

Fault-Injection-Based Assessment of Fail-Silence Provided by Process Duplication versus Internal Error Detection in Scientific-Based Applications

David T. Stott, Neil A. Speirs, Jun Xu, Saurabh Bagchi, Keith Whisnant, Zbigniew Kalbarczyk, and Ravishankar Iyer

Coordinated Science Laboratory
1308 West Main Street, Urbana, IL 61801

SF 298 MASTER COPY KEEP THIS COPY FOR REPRODUCTION PURPOSES

REPORT DOCUMENTATION PAGE

Form Approved OMB NO. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 2000	

4. TITLE AND SUBTITLE

Fault-Injection-Based Assessment of Fail-Silence Provided by Process Duplication versus Internal Error Detection in Scientific-Based Applications

5. FUNDING NUMBERS

6. AUTHOR(S)

D. Stott, N. Speirs, J. Xu, S. Bagchi, K. Whisnant, Z. Kalbarczyk, and R. K. Iyer

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(ES)

Coordinated Science Laboratory
University of Illinois
1308 W. Main Street
Urbana, IL 61801

8. PERFORMING ORGANIZATION REPORT NUMBER

UILU-ENG-00-2214

CRHC-00-03

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 uses fault injection a) to compare two software-based arthitectures for providing fail-silent processes, Voltan and Chemeleon ARMORs and b) to provide insights into the issues in using fault injection for comparative studies. Voltan uses process-level duplication and a voting procedure to provide fail-silence while Chameleon uses internal self-checking (th ough it supports additional modes, such as replication). The paper presents results from three different injection campaigns with two applications, Fast Fourier Transform and the radix sort.

Using NFTAPE, a versatile distributed fault injection tools, we evaluate the fault tolerance of each system against target-specific faults (such as corrupting message queues or corrupting fields within a message header). Results from the fault injection experiments show that each system is able to protect the target application from a significant percentage of faults (e.g., 97% for Voltan and 81% for Chameleon) without requiring any special purpose hardware.

14. SUBJECT TERMS

Software-Implemented Fault Tolerance, Fault Injection, Fail-Silent, Distributed Computing, Dependability

15. NUMBER IF PAGES

24

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

Fault-Injection-Based Assessment of Fail-Silence Provided by Process Duplication versus Internal Error Detection in Scientific-Based Applications

David T. Stott, Neil A. Speirs†, Jun Xu, Saurabh Bagchi, Keith Whisnant,
Zbigniew Kalbarczyk and Ravishankar K. Iyer

Center for Reliable and High-Performance Computing
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1308 W. Main St., Urbana, IL 61801-2307
E-mail:
{dstott,junxu,bagchi,kwhisnan,kalbar,iyer}@crhc.uiuc.edu

†Department of Computing Science
University of Newcastle upon Tyne
United Kingdom, NE1 7RU
E-mail: Neil.Speirs@ncl.ac.uk

Abstract

This paper uses fault injection a) to compare two software-based architectures for providing fail-silent processes, Voltan and Chameleon ARMORs and b) to use these experiments to provide insights into the issues in using fault injection for comparative studies. Voltan uses process-level duplication and a voting procedure to provide fail-silence while Chameleon using internal self-checking (though it supports additional modes, such as replication). The paper presents results from three different injection campaigns with two applications, Fast Fourier Transform and the radix sort.

Using NFTAPE, a versatile distributed fault injection tool, we evaluate the fault tolerance of each system against target-specific faults (such as corrupting message queues or corrupting fields within a message header). Results from the fault injection experiments show that each system is able to protect the target application from a significant percentage of faults (e.g., 97% for Voltan and 81% for Chameleon) without requiring any special purpose hardware.

Keywords: Software-Implemented Fault Tolerance, Fault Injection, Fail-Silent, Distributed Computing, Dependability

Fault-Injection-Based Assessment of Fail-Silence Provided by Process Duplication versus Internal Error Detection in Scientific-Based Applications

David T. Stott, Neil A. Speirs†, Jun Xu, Saurabh Bagchi, Keith Whisnant,
Zbigniew Kalbarczyk and Ravishankar K. Iyer

Center for Reliable and High-Performance Computing
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1308 W. Main St., Urbana, IL 61801-2307
E-mail:
{dstott,junxu,bagchi,kwhisnan,kalbar,iyer}@crhc.uiuc.edu

†Department of Computing Science
University of Newcastle upon Tyne
United Kingdom, NE1 7RU
E-mail: Neil.Speirs@ncl.ac.uk

Abstract

This paper uses fault injection a) to compare two software-based architectures for providing fail-silent processes, Voltan and Chameleon ARMORs and b) to use these experiments to provide insights into the issues in using fault injection for comparative studies. Voltan uses process-level duplication and a voting procedure to provide fail-silence while Chameleon using internal self-checking (though it supports additional modes, such as replication). The paper presents results from three different injection campaigns with two applications, Fast Fourier Transform and the radix sort.

Using NFTAPE, a versatile distributed fault injection tool, we evaluate the fault tolerance of each system against program-specific faults (such as corrupting message queues or corrupting fields within a message header). Results from the fault injection experiments show that each system is able to protect the target application from a significant percentage of faults (e.g., 97% for Voltan and 81% for Chameleon) without requiring any special purpose hardware.

1. Introduction

A common assumption made in software-implemented fault tolerance mechanisms, such as message logging, checkpointing, and process replication, is that the processing elements will suffer only crash failures, i.e., a processing element will either perform correct state transitions or will cease to function and become silent. To meet this assumption in a realistic manner, some form of self-checking facility is required within an element to detect a faulty state transition and stop the element from producing any further outputs. Field studies have also shown that in a distributed environment executing on off-the-shelf hardware components, the fail-silence assumption can be violated [21]. We define the fail-silence

property as producing the proper output, a detectably invalid output, or no output.

In this paper, we compare two approaches for providing fail-silence, Voltan [5] and Chameleon ARMORs [7]. Voltan uses duplicated nodes and a voting algorithm to prevent incorrect messages from leaving the system.

Chameleon ARMORs (Adaptive Reconfigurable Mobile Objects for Reliability) are the functional modules in the Chameleon environment, each executing as a separate process. ARMORs are designed to support a range of execution modes including replication and a variety of error detection techniques to provide node and process fail-silence. (Results from Triple Modular Redundant execution in an earlier Chameleon implementation are provided in [7].) Chameleon uses a set of internal self-checking mechanisms to detect errors. Throughout the paper, we compare these two approaches to providing fail-silence.

To analyze the fail-silence of each system, we used a fault injection based approach. NFTAPE [20], a tool for composing and executing fault injection experiments in a distributed environment, was used to injected faults. Besides using traditional single-event upset faults (such as memory bit-flips), the study used a set of faults specific to each system (e.g., corruptions in specific message queues of Voltan).

The experiments illustrate (a) the strengths and weaknesses of each approach (process duplication and self-checking ARMORs) and (b) the difficulties involved in applying fault injection to compare two very different systems.

The rest of the paper is organized as follows. Section 2 describes other research on fail-silent nodes and their validation. Section 3 provides background information on Voltan and Chameleon. Section 4 describes our validation methodology. Section 5 presents the results of our experiments. Section 6 discusses the issues in comparing Voltan and Chameleon and provides performance measures. Section 7 concludes the paper.

2. Related Work

A fail-silent node that uses replicated processing with comparison/voting must incorporate mechanisms to keep its replicas synchronized, to avoid the states of the replicas from diverging. Synchronization at the level of processor micro-instructions is logically the most straight forward way to achieve replica synchronism. Such hardware-based designs are increasingly expensive and difficult to imple-

ment. Hence there has been much interest in providing fail-silent nodes through software.

The task/process level synchronization approach used in Voltan was pioneered by the designers of the Software Implemented Fault Tolerance (SIFT) failure-masking node [24]. In SIFT, an application process is structured as a set of cooperative cyclic tasks. Each task performs a deterministic computation. The execution of a particular iteration of a task consists of inputting data (possibly generated by previous iteration of other tasks), processing the data, and outputting results. Fault tolerance is achieved by voting on the input data. Thus, task replicas must be synchronized at the beginning of each iteration (start of a *frame*). Because of its application-dependent design, the SIFT architecture can only be applied to a restricted range of applications. This is also the case for the VOTRICS system [22], which follows the design principles of SIFT to provide fault tolerance in a different, but still specific, class of applications (railway signaling systems).

There has always been concern over the performance of software-implemented fault-tolerant middleware caused by the overheads imposed by redundancy management protocols. In SIFT, for instance, redundancy management protocols can consume as much as 80% of the processor throughput [15]. Hybrid solutions have been proposed to circumvent this problem. MAFT [9], FTP-AP [12], and Delta-4 [16], hybrid architectures structured around a micro-instruction-synchronized hard core on top of which conventional processors are replicated. The micro-instruction-synchronized hard core is responsible for executing redundancy management functions (e.g., message voting). This improves performance; however, the hard core reintroduces the problems of associated with the hardware-implemented nodes.

The MARS architecture [18, 11] uses a combination of special-purpose hardware (e.g., comparators) and software approaches (e.g., double execution) to provide fault tolerance. Karlsson *et al.* [8] used three different physical fault injectors to assess the coverage of error detection mechanisms in this system. They found that the hardware and software mechanisms provided 98.7% coverage without duplicated execution. A software injection study for the same system [6] turned up surprisingly different results about the efficiency of the different detection techniques. It showed fail-silence of 85% without application-level detection mechanisms, with the coverage becoming perfect when application specific data consistency and other checks were added.

Fault injection has been recognized as an important tool in the dependability validation process, both for fault removal and fault forecasting [1]. Avresky [3] proposed a framework to guide the generation

of fault injection tests, to help developers remove fault from their designs. Arlat [2] used fault injection to evaluate the fail-silence of a multicast protocol in Delta-4. Karlsson [8] compared three different methods of physical fault injection. Koopman [10] compared the robustness of 13 different POSIX operating systems by testing their system calls and C library calls. Tsai [23] compared the reliability of a two generations of Tandem's TMR-based prototype machines. Madeira *et al.* [14] used physical fault injection to test the built-in detection techniques of two processors: Z80 and MC68K. No previous study, however, has employed fault injection to compare different software-implemented fail-silence or fault tolerance schemes.

The benefits of this study are twofold:

1. The experiments and results clearly expose the strengths and weaknesses of each approach: process duplication and self-checking ARMORs.
2. They also illustrate the difficulties in comparing systems via a common fault injection campaign. The insights obtained are important in developing a dependability benchmark for computer systems.

3. System Descriptions

This section describes the fundamentals of the two target systems to provide the reader with basics of the systems operate. More detailed descriptions of Voltan and Chameleon can be found in [5, 4] and [7], respectively.

3.1. Voltan

Voltan assumes that a failed process can exhibit Byzantine behavior but that each non-faulty process can sign a message it sends by affixing the message with a message-dependent, unforgeable signature. A non-faulty process is assumed to be able to authenticate any message it receives. The computation performed by a process is assumed to be deterministic. Furthermore, processes are distributed and communicate by passing messages. So, if non-faulty replicas have identical initial states, then they will produce identical output messages, provided (a) all the non-faulty replicas of a process receive identical input messages and (b) all the non-faulty replicas process the messages in an identical order.

To meet the property of fail-silence, a single logical process is formed from two replicas. As each replica forms an output message, it signs that message and passes a copy to its partner. Upon receiving

a signed output message, the replica compares it with the locally generated result. If the comparison is successful, the replica adds a second signature and outputs the message. If a comparison fails, the process signals the error and halts to prevent the error from propagating. Hence, a fail-silent Voltan process will output either correct messages or, detectably incorrect messages (these messages can be signed at most once).

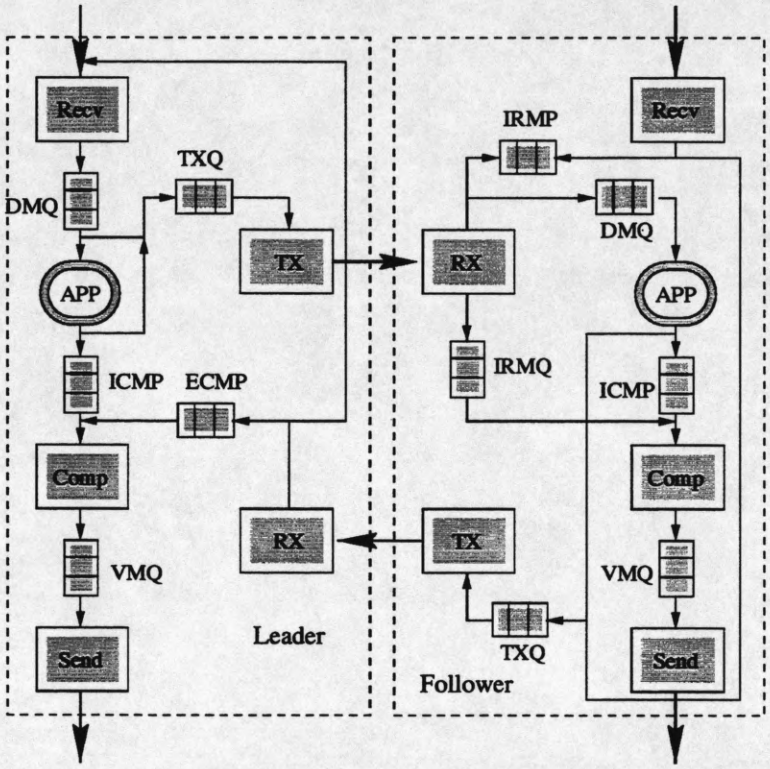

Figure 1. The Structure of a Fail-Silent Voltan Process

The method of operation for a fail-silent process pair is shown in Figure 1. The 'Voltan system' consists of five cooperating threads on each node (Recv., Comp., Send, RX, and TX). The processes in the process pair are called the *Leader* and the *Follower*. The leader's receiver thread (Recv) accepts authentic double-signed messages and places them into the application's Delivered Message Queue (DMQ) while sending a copy to the follower. The application threads select a message from DMQ, process the message, and form an output message. A copy of this message is signed and transmitted to the other replica. The unsigned message is stored locally in the Internal Candidate Message Pool (ICMP). When the Reception thread (RX) receives a singly-signed message it places it in the External Candidate Message Pool (ECMP). The Comparison thread (Comp) compares messages from ICMP and ECMP. If the

comparison succeeds, the message from the ECMP is signed again and the doubly-signed message is placed into the Voted Message Queue (VMQ); otherwise the replica terminates to prevent any error from propagating. Finally, the Send thread picks up messages from the VMQ and dispatches them to their destinations.

The follower process also contains a mechanism to ensure that if a correctly functioning leader misses receiving a valid message for processing but the follower does receive that message then the message gets ordered and processed by the pair.

3.2. Chameleon ARMORs

The Chameleon environment provides a means for constructing reliable distributed application around *ARMOR* (Adaptive Reconfigurable Mobile Objects for Reliability) processes. ARMORs communicate through message passing and can be installed on any node in the Chameleon network. They are built from replaceable components called *elements* and *compounds*.

Elements constitute the most basic functional unit of the ARMOR and can be replaced during runtime, thus allowing the ARMOR process to adapt to changing application requirements. Elements are passive objects that are invoked to perform specific operations (such as taking a checkpoint) by messages; each incoming message spawns a new thread of execution within the ARMOR process. Messages contain one or more message operations, each operation invoking a specific function in one or more elements within the ARMOR.

An ARMOR can be constructed to function as an application by designing elements to perform the functions of the application. When implementing a distributed application as an ARMOR, communication between the nodes uses the normal Chameleon infrastructure.

In addition to the application ARMORs, the Chameleon environment consists of several other ARMORs that help ensure system availability. Briefly, daemon ARMORs are installed on each node to launch and manage other ARMORs on the node, to oversee them, and to route messages between them. Managers (e.g., the Fault Tolerance Manager (FTM)) oversee specific ARMORs and are responsible for recovering from failures in their subordinates. In the context of applications, a manager installs the required number of application ARMORs on various nodes. The manager only intercedes when the process needs to be stopped, such as when migrating an application ARMOR to a different node, while

a cooperating ARMOR undergoes reconfiguration, or when recovering from a checkpoint.

Chameleon divides its error detection mechanisms into several levels [25]. A level is denoted *lower* than another if it is implemented more closely to the ARMOR being monitored. Level 1, the lowest level, consists of detecting errors within the ARMOR. Level 2 consists of detection by the daemon installed on the same node as the ARMOR. The same daemon is responsible for monitoring all locally installed ARMORs. Levels 1 and 2 interact to provide error containment within the local node and to ensure the abstraction of fail-silent processes and nodes, respectively. Levels 3 and 4 involve distributed detection protocols among replicas or pseudo-replicas of ARMORs, possibly executing on different nodes; these levels are not considered for this paper. Table 1 lists the available techniques. It should be noted that the ARMOR architecture is designed to support both control flow and data signatures. There are two types of data checks in Chameleon: data signatures, which are based on replicated information being present in multiple elements, and data audits, which have been used by Liu [13] in the design of a high-availability mobile telephone network controller. However, these checks were not implemented for the application environment under test in this paper.

Table 1. Error Detection Mechanisms in Chameleon ARMORs

Detection Mechanism	Level	Description
Livelock Detection	1	Checks timeout on mutex lock
Coarse-grained I/O Signature	1	Checks pattern of I/O message types against prescribed set of valid sequences
Fine-grained Control Signature	1	Checks runtime control flow against valid control flow paths
Text-segment Signature	1	Periodically checks signature of text segment pages
Crash Failure Detection	2	Detects abnormal process termination
Timing Failure Detection	2	Timeout on messages

4. Dependability Assessment

To validate the fail-silent properties of the two target systems (Voltan and Chameleon), we have run several different pairs of fault injection campaigns (each campaign pair includes one campaign using Voltan and one using Chameleon). The goals of these campaigns vary. The first pair of campaigns was used to verify that the systems properly handle faults they were designed to handle (for example, Voltan should still operate correctly when a message is dropped). The second pair of campaign campaigns was meant to expose the vulnerable parts of the system's design (for example, data corruption in Chameleon).

The last pair of campaigns tested the fail-silence of the processes in a more general sense by injecting random faults, such as random memory corruptions into the process stack and heap.

4.1. Target Applications

The first issue to be addressed is the mode in which each environment (Voltan and Chameleon ARMORs) is to be executed. In Voltan, the application needs to be modified by replacing communication functions and adding initialization functions to execute as a replicated process. A Voltan process called Nizam oversees the process's initialization.

In Chameleon, the ARMORs can support applications in three different ways:

1. The application runs in a replicated, checkpoint-recovery, or any other application-dependent mode. In this mode, Chameleon has little role to play in error detection or recovery in the application.
2. The application runs as an Embedded ARMOR, wherein the ARMORs provide error detection and recovery to the application, but the application messages are not routed through the ARMORs.
3. The application is modified to run as a full-fledged ARMOR. This is the most intrusive mode of operation.

Recall that our goal is to compare self-checking ARMORs and process duplication in Voltan. The third mode of operation, is the only one in which Chameleon operates in a configuration that is reasonably comparable with the Voltan configuration.

To test the systems, two simple and well-known target applications *Fast Fourier Transform* (FFT) and *radix sort* are run on a pair of nodes. Each application is parallelized into exactly two tasks and executes on a distributed network of workstations or PCs. The choice of these applications was primarily based on what was available "off-the-shelf" and could easily be implemented on both the systems so that the evaluation could be based on the same workloads.

The first program is a version of the FFT, a common algorithm for several scientific applications, such as signal processing. In our implementation (based on [17]), the algorithm runs in $\log_2(N)$ iterations, where each iteration performs $N/2$ parallel operations (the transform operates on an array of complex numbers with N elements). The master node reads the input file from disk. For each iteration after the first one, (a) a copy of the array is sent to the slave node, (b) the operations are divided evenly among

the two nodes, and (c) the slave node sends a copy of the array back to the master. For the experiments, we used a 4096 element input array.

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

4.2. Fault Injectors

In this set of experiments, we use two basic classes of fault injectors. The first class injects traditional single event upsets to the target process's memory (heap (data), stack, and text (code) segments). Because this class uses a simple, easy-to-understand fault model that applies to any system, it is a good choice from which to build the basis of our comparison. The second class uses what is called target-specific fault injectors. This class of injectors includes those that inject faults from functions within the target program. Using such an injector, faults can be tailored toward the specific implementation (e.g., perform message corruptions in Voltan when the messages are in the Delivered Message Queue or corrupt the control fields in the ARMOR messages). The specific faults that were injected are described in Sections 5.1 and 5.2.

The message queues in Voltan appear to be a vulnerable component, since most of the work in Voltan for providing fail-silence is processed in them. The effect of corrupting the queues will resemble timing errors or event ordering errors between nodes, which are common causes of errors in agreement protocols. Thus, we are interested in the effect of corrupting the queues with faults such as dropping queued messages, reordering messages, inserting duplicate messages, and corrupting messages. In particular, the DMQ and the VMQ queue (shown Figure 1) may be most vulnerable, since they send messages directly to the application or to the other node.

To support these corruptions, only a few simple modifications to the Voltan application library were needed. First, methods were added to the Queue class: FiDrop(), FiReorder(), FiDuplicate(), and FiModify() (which respectively popped a message form the queue, reordered two adjacent messages in the

queue, duplicated a message in the queue, and corrupted the contents of a message). Next, a function was added that takes a string describing a fault (the name of a queue and the type of fault) as input and calls the appropriate fault method (from the four faults above). Next, in order to access this function, a simple class was added that opens a named pipe (e.g., a Unix socket) to wait for trigger events. At this point, any program can inject a fault by writing a message to the named pipe.

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

4.3. NFTAPE

NFTAPE [20] is a tool for conducting automated fault injection experiments. It provides an API for writing simple fault injectors. These fault injectors are called light-weight fault injectors (LWFI) because they do not need to include code for triggering faults or for logging results, as NFTAPE provides these services. Simple triggers wait for events and then use API calls to send a trigger event to a LWFI or to another trigger (to support a cascade of triggers). NFTAPE can compose new fault injection experiments by interchanging LWFIs and triggers that use the NFTAPE API.

NFTAPE contains two main components: the Control Host and the Target Nodes. The Control Host, which generally resides on a safe node, processes a file called a Campaign Script. This file provides information about the global sequencing of events in a campaign (or set of runs), what processes need to be execute, what parameters they take, and when to run or terminate the processes. These processes can be target applications, monitors, triggers, fault injectors, acceptance tests, or other processes on the node. The Control Host also logs the results of the experiments (including output from NFTAPE and all processes it runs) for off-line analysis.

Each Target Nodes (i.e., any node other than the Control Host), runs a program called the Process

Manager. This program executes all processes, monitors when they terminate, captures any I/O and processes NFTAPE commands between any nodes or processes.

Four important features of NFTAPE facilitated these experiments:

1. the LWFI API (new fault injectors such as the target-specific fault injectors were easy to write),
2. process management (NFTAPE can start all the processes used in the experiment and clean them up if the program terminates abnormally),
3. logging (the outputs from each process, including fault injector, trigger, and application, are logged using the same format), and
4. automated experiment sequencing (the order of events such as the order in which processes run).

4.3.1. Configuring NFTAPE for Target-Specific Fault Injectors

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

To trigger the Voltan fault injection methods through the named pipe interface, a small program was needed to receive NFTAPE triggers (a function provided in the NFTAPE API library). Figure 2 show the complete Voltan fault injector. To inject memory faults, NFTAPE executes and triggers a preexisting driver-based fault injector [20].

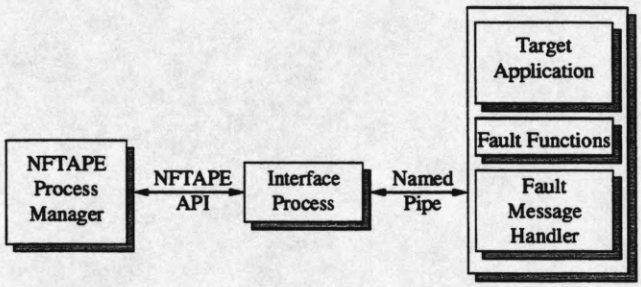

Figure 2. Target-Specific Fault Injector Example for Voltan

To trigger the fault injection elements in Chameleon through the named pipe interface, it uses an interface program just like the one for Voltan. This small interface program to convert the NFTAPE API format to Chameleon's named pipe interface.

5. Experiments

The experimental evaluation of the fail-silence provided by Voltan and Chameleon was done through three different (pairs of) campaigns.

1. Campaign A stresses the available detection mechanisms in each system and assesses the effectiveness of the specific techniques. For this campaign, directed message control flow and text segment corruptions are performed in Chameleon, and message corruptions, reordering, drop, and duplication are done in Voltan.

2. Campaign B injects faults for which there is no direct available detection mechanism in the system. This is meant to expose the worst-case scenarios in both system. For Chameleon ARMORs, this campaign consists of injections after the self-checks have been done, and for Voltan, it consists of injections where the faults are aliased by the signatures.

3. Campaign C injects random memory faults are injected instead of selecting faults based on features of the system being examined. Faults are injected into the text, heap and stack of the executing processes.

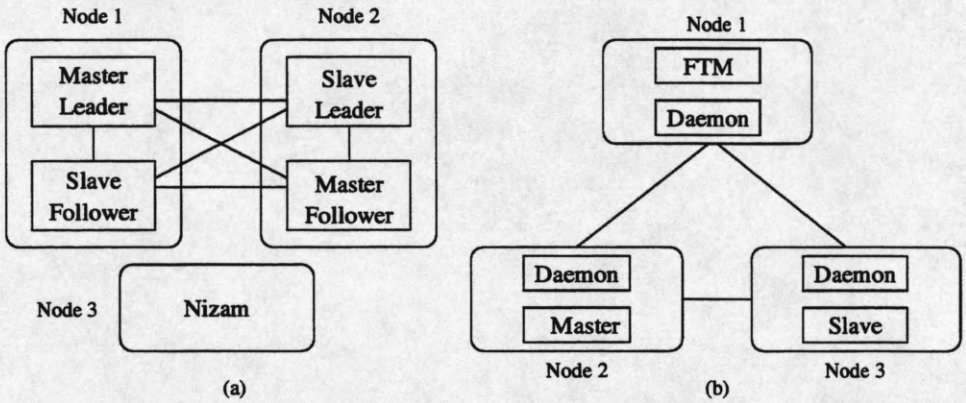

Figure 3. Experimental Configuration for (a) Voltan and (b) Chameleon

Figure 3 shows the configurations in which systems run. For both FFT and the radix sort, the application is parallelized into two tasks, denoted as Master and Slave. In Voltan, each task runs as a duplicated process: a Leader and a Follower. Thus, there are four application processes: Master Leader, Master Follower, Slave Leader, and Slave Follower. In Chameleon, for the purpose of this study, duplication of processes was not used, and hence, there were only two application processes, Master and Slave, corresponding to the two application tasks. In both systems, fault injection targets the application processes.

5.1. Campaign A: Assessment of Specific Detection Mechanisms

In this campaign, directed fault injection is done to stress and evaluate the effectiveness of the detection techniques implemented in each system.

5.1.1. Voltan

As described in Section 4.2, we added the capability of injecting specific errors into the queues used by Voltan to process messages. The faults injected were the modification of a message waiting in a queue, the removal of a message from a queue, the addition of a duplicate of a message held in a queue, and reordering pairs of messages held in a queue.

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

The queues selected for injection were the Delivered Message Queue (DMQ) and the Voted Message Queue (VMQ) of the slave processes of the radix sort and FFT applications. Errors in these queues represent faults arising when messages arrive and depart from the Voltan nodes. The results in Table 2 show the effect of injecting the faults listed above into the given Voltan queues. No variations in behavior were displayed between the 25 runs with 6 faults each while running the radix sort, and the same behavior was observed for FFT.

Table 2. Results from Campaign A for Voltan

Correct means that the program ran to completion and produced the expected results. Fail-Silent means that the program stopped and failed to produce any results.

Queue	Reorder	Drop	Modify	Duplicate
Leader DMQ	Correct	Fail-Silent	Fail-Silent	Correct
Follower DMQ	Correct	Fail-Silent	Fail-Silent	Correct
Leader VMQ	Correct	Correct	Fail-Silent	Correct
Follower VMQ	Correct	Correct	Fail-Silent	Correct

The results show that Voltan provides 100% fail-silence coverage for this type of message injection. Injections into the Leader and the Follower produced identical results. Nizam, the process that oversees

the application process's initialization, is the only unprotected process in Voltan. It is only invoked at the beginning and at the end of the application run. Faults are not injected into Nizam because it is assumed such a fault will crash the application.

5.1.2. Chameleon Self-Checking ARMORs

The results from Campaign A of the directed message injections into ARMORs are shown in Table 3. In this campaign, target-specific fault injections are conducted. The injection is targeted at the control fields of the ARMOR's messages. This type of injection is meant to stress the coarse-grained signature mechanism of ARMORs. The injection occurs after the message has been generated by the application element but before the coarse-grained signature element has checked the message. This type of fault injection cannot be time-triggered exactly, rather it is a hybrid of message-rate-based and time-based triggering. In our implementation, the fault injection element attempted to inject a fault once per second; if no message was present, the fault injector would set a flag to inject the next message(s).

Table 3. Results from Directed Message Control Field Injections in Chameleon

	Control-field Corruption		Delay		Drop	
Result	FFT	rsort	FFT	rsort	FFT	rsort
Good	0	0	0	20	0	2
Hang	3	2	0	0	1	0
Coarse-Grain Sig.	11	18	0	0	0	0
Crash Failure	15	10	30	10	29	28
Non-Fail-Silent	1	0	0	0	0	0
Total	30	30	30	30	30	30

The results from Table 3 show that the coarse-grained signature is effective in all but one case of message corruption. The important point is that in 35-60% of cases, the error is detected without a crash of the process. The single case of non-fail-silent behavior observed with FFT resulted from the corruption of the pointer field of one element of the linked list of message operations. As a result, the FFT application skipped an operation on one element of the input data, and the output result differed from the golden run for one data point. The message delay and drop either cause a good result or a process crash. Thus, all the cases are fail-silent. Radix sort (rsort) shows greater resilience to delayed and dropped messages because of the native implementation.

Another set of injections was targeted at the text segment of the FFT and radix sort applications (only

results for FFT are presented). Random single-bit flips were injected into the text segment, which is protected through the text segment signature in Chameleon. The results of these injections are shown in Table 4.

Table 4. Results from Chameleon Injection into the Text Segment for the FFT Application
Rand. means anywhere in the text segment, including elements that are used and unused; Elem. means into into the text segment of the elements that are used, including the FFT element and a few other Chameleon infrastructure elements; FFT means into the FFT element

	With Text Signature				W/O Text Signature			
Result	Rand.	Elem.	FFT	Tot.	Rand.	Elem.	FFT	Tot.
Good	0	0	0	0	16	5	7	28
Hang	0	1	1	2	0	0	0	0
Text Segment Sig.	24	20	22	66	0	0	0	0
Crash Failure	6	9	7	22	13	21	20	54
Non-Fail-Silent	0	0	0	0	1	4	3	8
Total	30	30	30	90	30	30	30	90

All the elements in the Chameleon library (about 40) are statically linked into each ARMOR, though the ARMOR may be using just a few elements from the library (e.g., FFT only uses three elements, including the core FFT element). When randomly injecting over the entire text segment, the text segment signature check always detected the error unless the fault caused the program to abort first. Since there is such a large amount of dead code compiled into the text segment, targeting injections to code that is more likely to execute provides a better idea of how effective the signature check is. To test this, we targeted faults into the text segment's pages that contained elements in use and into those of the application element. The results from these injections, shown in Table 4 (columns "Elem." and "FFT"), demonstrate that the text segment signature is able to capture most of the faults and again not only prevent fail-silence violations but also capture the error before a process crash. It can be seen that text segment signature results in some false positives; that is, it signals an error when it detects a bit flip in the text segment, though that fault may not otherwise have caused any error. As a result, there are no good runs with the text signature turned on. To reduce the number of false positives, we have proposed a modification of the text signature technique to flag an error only when the fault is in a page of the text segment that is being used or that is going to be used shortly.

5.2. Campaign B: Assessment of System Vulnerabilities

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

5.2.1. Voltan

The message injections described in Section 5.1.1 do not investigate the situation where a modified message produces the same signature as the original message. This possibility can be made arbitrarily small by using digital signature techniques (e.g., [19]). However, the existing implementation uses only a simple checksum in the range 0-255 to authenticate data. We investigated the effect of injecting an error into messages that leave the signature unaltered. If corruption occurs within a message, there is a 1/256 chance of its being accepted as authentic by Voltan. We injected faults into the Delivered Message Queue of the leader slave node. This represents the worst-case fault scenario in a Voltan node, where corrupt messages have been processed by both leader and follower nodes. The results obtained are shown in Table 5.

Table 5. Results of Injecting Message Faults into Voltan that Bypass Signature Checking
Correct Result means that the application completed and produced the correct results. Fail-Silent Behavior includes any run that did not produce an output (either hanged or crashed due to an error). Non-Fail-Silent includes runs that produced an incorrect output.

Result	radix sort	FFT	Total
Correct Results	6	8	14
Fail-Silent Behavior	12	0	12
Non-Fail-Silent	12	22	34
Total	30	30	60

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

5.2.2. Chameleon Self-Checking ARMORs

The message injections in Section 5.1.2 do not investigate the situation in which the data field of a message is corrupted. The purpose of Campaign B is to look at how ARMORs react to different kinds of faults in the messages. In this case, faults were randomly injected into the entire message, both control and data fields. In addition, duplicate messages were sent.

Table 6. Results of Injecting Message Faults into Chameleon that Bypass Signature Checking
All the detection techniques in Levels 1 and 2 were active.

	Duplicate		Modify		
Result	FFT	rsort	FFT	rsort	Total
Good	0	4	0	2	6
Hang	2	0	0	0	2
L1 Detected	0	0	0	0	0
Crash Failure	9	14	16	8	47
Non-Fail-Silent	19	12	14	20	65
Total	30	30	30	30	120

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

Another set of injections is done to the application process's heap data. Since the heap data is not protected in the current Chameleon implementation, this class of injections also falls in the category of exposing the system vulnerabilities. Results from this injection are presented in the column titled Heap in Table 8.

5.3. Campaign C: Random Memory Faults

In this campaign, random memory faults are injected into the heap, stack, and text segments of the application processes.

5.3.1. Voltan

The campaign tested the Voltan system against random memory faults (memory bit flips into the text, stack, and heap sections of each of the test applications). Each combination of fault location and application included 25 runs, with a fault rate of one fault per second. The results of each run are shown in Table 7.

Table 7. Results from Random Memory Bit Flips in Voltan

Heap MF runs injected faults into the follower copy of the master process and Heap SL runs injected faults into the leader copy of the slave process

	Heap		Heap MF		Heap SL		Text		Stack		
Result	FFT	rsort	FFT	rsort	FFT	rsort	FFT	rsort	FFT	rsort	Total
Good	13	13	11	8	23	24	11	16	19	18	156
Hang	9	11	14	16	2	1	2	0	6	7	68
Seg. Fault	0	0	0	0	0	0	12	9	0	0	21
Non-Fail-Silent	3	1	0	1	0	0	0	0	0	0	5
Total	25	25	25	25	25	25	25	25	25	25	250

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

The heap injections were repeated on different nodes. Initially, faults were injected into the Master-Leader. The results of these injections are given in Table 7 in columns marked Heap, Text and Stack. To explore the impact of varying the fault injection target (i.e., whether it is the Leader or the Follower), we performed two additional sets of fault injections, which targeted (1) the Master-Follower and (2) the Slave-Leader. The results are given in Table 7 in columns marked Heap MF and Heap SL, respectively. We observed that the choice among copies (Leader or Follower) had little effect–the number of incorrect outputs was slightly lower for the Follower. However, faults injected into the Slave were much less likely to manifest. A possible reason for this may be that the data on the Slave are only valid between the time it receives the input and the time it sends the results; there are no live data while the Slave is waiting to receive a message, as is the case for the Master.

In all but 5 of the 250 (2.0%) runs, the program either produced the correct output (62%), hung (27%), or aborted on a segmentation fault (8.4%). In each of these cases, the run was fail-silent because it either produced the correct output or no output.

Process duplication in Voltan protected an application from producing an error in 89 of the 94 runs where a fault manifested by either hanging a node or crashing silently. (Presumably, the cases where a node crashed would have also crashed if Voltan was not used.) Only 5.3% of these runs had errors that escaped Voltan.

5.3.2. Chameleon Self-Checking ARMORs

The results from Campaign C for Chameleon are shown in Table 8. In this campaign, we examined the effects of injecting random bit flips into the heap, stack, and text segment of a Chameleon ARMOR. In addition to the core application element, there are also a few Chameleon-specific elements in the ARMOR, e.g., the named pipe management element for establishing communication with the local daemon, the coarse and fine-grained signature elements, and the text segment signature element. In the experiments, the heap injection is done to the entire ARMOR's heap, the stack injection is done to every thread's stack, and the text injection is done to the text segment of the application element only.

Table 8. Results from Random Memory Bit Flips in Chameleon
All Level 1 detection techniques were active except in column 5 where text signature was turned off.

	Heap		Stack		Text		Text w/o Sig.		Total	
Result	FFT	rsort	FFT	rsort	FFT	rsort	FFT	rsort	FFT	rsort
Good	9	7	5	5	0	0	7	4	21	16
Hang	0	4	0	0	1	0	0	0	1	4
L1 Detection	1	2	0	0	22	24	0	0	23	26
Crash Failure	2	2	24	25	7	6	20	26	53	59
Non-Fail-Silent	18	15	1	0	0	0	3	0	22	15
Total	30	30	30	30	30	30	30	30	120	120

The results show a fail-silence coverage of 84.6% averaged over the two applications and using both text segment injections (individually 87.5% and 81.7% for the radix sort and FFT respectively). A large contribution of the fail-silence violations is from the heap injections. Removing the results of the heap injections, the fail-silence coverage becomes 98.3%. This is understandable, since most of the data of both FFT and the radix sort are allocated dynamically and reside in the heap. The signature mechanisms

in the target implementation protect the text segment and control flow of the ARMORs, but they do not provide any detection for the data.

As mentioned earlier, the application-supported data audits provide protection against data errors. In addition, the ARMOR architecture supports data signature. The overall effectiveness of the two approaches is yet to be evaluated experimentally. The basis behind the data signature is that multiple copies of the internal state of ARMORs are available. Either there is explicit replication of elements within an ARMOR or there is naturally information in multiple elements within an ARMOR. For example, the set of daemons in Chameleon is a subset of the set of ARMORs. These are maintained in separate elements in the manager. The data signature is formed by a executing a hashing function on the shared internal state and comparing it with the hashed value of the state at the alternate location(s). In the case of FFT or the radix sort, the application element's data is its state and will be protected by such a Data Signature. Another observation is that the fine-grained signature did not detected an control-flow errors which may have resulted from the text injections.

6. Discussion

6.1. Difficulties Encountered Comparing the Two Systems

This study is the first we are aware of to compare the fail-silence achieved by two dissimilar software-based approaches. We needed to resolve several issues in order to build a meaningful comparison.

1. *Different Data Conventions.* We needed to address the fact that each system had different data conventions. We assumed that a simple fault model like random memory bit flips to the text, heap, and stack regions would be a fair basis for measuring fails silence, but there are architectural differences for each of these. The text segment in Chameleon includes code for a large library of elements that were not used in this experiment. Thus, only a small fraction of the text faults affected code that could be executed. This becomes a problem for injecting text segment faults because the text segment checker is more likely to catch random text faults before they manifest if they are in dead code.

2. *Platform Differences.* Since both systems are multi-threaded, they contain several stacks instead of just one. For each, we selected a thread and then an address within that thread's stack. Using

this method, stack faults in Solaris's thread implementation appeared to manifest more frequently than in Linux's. Since the choice of platform may affect the dependability of the programs running on it, it makes sense to first understand the dependability issues for each platform. Future studies need to look at the dependability of each of these platforms running the same application.

3. *Fault Triggers.* The fault trigger criterion is another issue that may lead to problems in fault injection comparison studies. The trigger in these experiments was time-based (one fault per second). It is questionable whether a simple rate is always appropriate, since execution time in different systems may differ greatly. For example, injecting one fault per second in a run that lasts for one second will differ from doing so in a run that lasts for five seconds. When injecting message faults, the timer trigger performed poorly because the fault injector often attempted to inject faults into an empty message queue. It is not clear how to trigger message faults in different systems when the systems have different message patterns.

4. *Use of Multi-Threaded and Multi-Process Applications.* As mentioned above, the stack fault model is more complicated for these cases than in a single-threaded process. Both systems are multi-process: Voltan uses a process called Nizam to locate services; Chameleon uses the Fault Tolerance Manager (FTM) to oversee the daemons. While Nizam runs as a single unprotected process, the FTM is an ARMOR with a variety of detection techniques. We did not inject faults into either of these processes in this study since neither process performs many actions while the application is running. A more complete study may need to address how to test the reliability of these processes.

5. *Target-specific Fault Injection.* Since many types of faults are very unlikely to be injected through random fault injection, it is often necessary to accelerate faults injections. One way to do this is to inject faults when the system is in a particular state (e.g., when the system is under high stress, while components are recovering, while other nodes are sending messages). Another approach is to inject specific faults which are likely to manifest, e.g., reordering messages on a queue. While these approaches are good for learning about system, they complicate comparing different systems, since the faults and triggers are specific to the program.

6.2. Performance Overheads

An important consideration for using software fault tolerance is the overhead one pays for the fault resilience. With a hardware solution, an application can run close to or at full speed, but software solutions incur overheads in performing checks and in synchronization between nodes. Table 9 compares the performance of the test applications running on the two systems and a baseline, non-fault-tolerant version using TCP/IP. The TCP/IP version and the Chameleon version ran on a Sparc-Solaris platform (140MHz Ultra processor), while the Voltan version ran on a Linux platform (233MHz Pentium processor). Each version removed debugging information (which was present in the other experiments). The execution time includes the time spent in the main computational and communication loop, but it does not include the time to input or output the data set. The overhead of the Chameleon execution results from the inefficiency of its message operations when exchanging large messages (which was the case with both applications) and from all the level 1 and level 2 detection techniques being active. Note that the Chameleon architecture supports the activation of specific detection techniques on an as-needed basis.

Table 9. Performance Comparison

Version	FFT	Radix Sort
Voltan	2.8 sec.	3.5 sec.
Chameleon	3.5 sec.	5.7 sec.
TCP/IP	1.4 sec.	1.8 sec.

7. Conclusion

In this paper we examined two approaches for providing fail-silent nodes: (1) process replication with voting, as used by Voltan and (2) internal self-checking mechanisms, as used by Chameleon ARMORs. The goal of this comparison was to see if the self-checking method (without explicate duplication) can achieve the same level of fault tolerance as process replication.

We used NFTAPE, a distributed fault injection, to assess the fail-silence of these two systems while executing one of two test applications, Fast Fourier Transform and the radix sort. The analysis was divided into three campaigns:

- The first campaign validated specific detection techniques in the systems and demonstrated almost perfect coverage for both system.
- The second campaign measured the need for protection by injecting faults into areas which are not directly protected by detection techniques. In both systems, about 45% of the fault injection runs produced non-fail-silent behavior.
- The third campaign, used random memory bit flips into the heap, stack, and text segments of the application processes. Voltan maintained fail-silence in 97.5% of the time, and Chameleon ARMORs in 84.6% of the time.

Since both applications used in this experiment are data-centric, heap injections are very likely to corrupt program data. Voltan provides a high coverage for this type of errors. It is likely that Chameleon ARMORs would demonstrate a higher error coverage executing a control-centric application (such as real-time control software). The fail-silence coverage obtained from the third campaign rose to 97.8% by considering only stack and text injections.

The study also provided insight into the issues that need to be considered when comparing the dependability of different systems. These include different data conventions, platform differences, different fault triggers, and use of multi-threaded and multi-process applications. Despite these unresolved issues, the experiments in this study were able to compare the two systems in terms of their fail-silence coverage.

References

[1] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, "Fault injection for dependability validation: A methodology and some applications," *IEEE Trans. Software Eng.*, vol. 16, no. 2, pp. 166–182, 1990.

[2] 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," *IEEE Trans. Reliability*, vol. 39, no. 4, pp. 455–467, 1990.

[3] D. Avresky and et. al., "Fault injection for the formal testing of fault tolerance," in *Proc. of the 22nd Int'l Symp. on Fault-Tolerant Computing (FTCS-22)*, pp. 345–354, June 1992.

[4] D. Black, C. Low, and S. K. Shrivastava, "The voltan application programming environment for fail-silent processes," *Distributed Systems Engineering*, vol. 5, pp. 66–77, June 1998.

[5] F. V. Brasileiro, P. D. Ezhilchelvan, S. K. Shrivastava, N. A. Speirs, and S. Tao, "Implementing fail-silent nodes for distributed systems," *IEEE Trans. Computers*, vol. 45, pp. 1226–1238, Nov. 1996.

[6] E. Fuchs, "Validating the fail-silence of the MARS architecture," in *Proc. of the 6th IFIP Int'l Working Conf. Dependable Computing for Critical Applications (DCCA-6)*, Mar. 1997.

[7] Z. Kalbarczyk, R. K. Iyer, S. Bagchi, and K. Whisnant, "Chameleon: A software infrastructure for adaptive fault tolerance," *IEEE Trans. Parallel and Distributed Systems*, June 1999.

[8] J. Karlsson, P. Folkesson, J. Arlat, and G. Leber, "Application of three physical fault injection techniques to experimental assessment of the MARS architecture," in *Proc. of the 5th IFIP Int'l Working Conf. Dependable Computing for Critical Applications (DCCA-5)*, pp. 267–287, Mar. 1996.

[9] R. M. Kieckhafer, C. J. Walter, A. M. Finn, and P. M. Thambidurai, "The MAFT architecture for distributed fault tolerance," *IEEE Trans. Computers*, vol. 37, pp. 398–405, Apr. 1988.

[10] P. Koopman and J. DeVale, "Comparing the robustness of posix operating systems," in *Proc. of the 29th Int'l Symp. on Fault-Tolerant Computing (FTCS-29)*, pp. 30–39, June 1999.

[11] H. Kopetz, H. Kantz, G. Grunsteidl, P. Puschner, and J. Reisinger, "Tolerating transient faults in MARS," in *Proc. of the 20th Int'l Symp. on Fault-Tolerant Computing (FTCS-20)*, pp. 466–473, June 1990.

[12] J. H. Lala and L. S. Alger, "Hardware and software fault tolerance: A unified architecture approach," in *Proc. of the 18th Int'l Symp. on Fault-Tolerant Computing (FTCS-18)*, pp. 240–245, June 1988.

[13] Y. Liu, Z. Kalbarczyk, R. K. Iyer, and I. H. Levendel, "Designing a high-availability mobile telephone network controller: A case study," in *Proc. of the IEEE Int'l Computer Performance and Dependability Symp. (IPDS'2K)*, May 2000.

[14] H. Madeira and J. G. Silva, "Experimental evaluation of the fail-silent behavior in computers without error masking," in *Proc. of the 24th Int'l Symp. on Fault-Tolerant Computing (FTCS-24)*, pp. 350–359, June 1994.

[15] D. L. Palumbo and R. W. Butler, "Measurements of SIFT operating system overhead," Tech. Memo. NASA Tech. Mem. 86322, NASA, 1985.

[16] D. Powell, ed., *DELTA-4: A Generic Architecture for Dependable Distributed Systems*. Spring-Verlag, Oct. 1991.

[17] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. O. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 2nd ed., 1992.

[18] J. Reisinger and A. Steiniger, "The design of a fail-silent processing node for the predictable hard real-time system mars," *Distributed System Eng. Journal*, vol. 1, no. 2, pp. 104–111, 1993.

[19] R. L. Rivest, A. Shamir, and L. M. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Comm. of the ACM*, vol. 21, pp. 120–126, Feb. 1979.

[20] D. T. Stott, Z. Kalbarczyk, and R. K. Iyer, "Using NFTAPE for rapid development of automated fault injection experiments," in *Digest of FastAbstracts of 29th Int'l Symp. on Fault-Tolerant Computing (FTCS-29)*, pp. 39–40, June 1999.

[21] A. Thakur, "Measurement and analysis of failures in computer systems," Master's thesis, University of Illinois at Urbana-Champaign, 1997. Advisor R. K. Iyer.

[22] N. Theuretzbacher, "VOTRICS: Voting triple modular computing system," in *Proc. of the 16th Int'l Symp. on Fault-Tolerant Computing (FTCS-16)*, pp. 144–150, July 1986.

[23] T. K. Tsai and R. K. Iyer, "An approach to benchmarking of fault-tolerant commercial systems," in *Proc. of the 26th Int'l Symp. on Fault-Tolerant Computing (FTCS-26)*, pp. 314–323, June 1996.

[24] J. H. Wensley *et al.*, "SIFT: Design and analysis of a fault tolerant computer for aircraft control," *IEEE*, vol. 66, pp. 1240–1255, Oct. 1978.

[25] K. Whisnant, S. Bagchi, B. Srinivasan, Z. Kalbarczyk, and R. K. Iyer, "Incorporating reconfigurability, error detection and recovery into the chameleon armor architecture," Tech. Rep. UILU-ENG-98-2227, Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, Dec. 1998.