

SOFTWARE-IMPLEMENTED FAULT INJECTION FOR  
DEPENDABILITY ANALYSIS OF HIGH-SPEED COMPUTER NETWORKS

BY

DAVID THOMAS STOTT

B.S., Carnegie Mellon University, 1996

THESIS

Submitted in partial fulfillment of the requirements  
for the degree of Master of Science in Electrical Engineering  
in the Graduate College of the  
University of Illinois at Urbana-Champaign, 1998

Urbana, Illinois

## ACKNOWLEDGEMENTS

I would first of all like to thank my family, without whose support I would not be a university research student. Next, I would like to thank my advisor Dr. Ravishankar K. Iyer for supporting my research.

I would also like to give thanks to the members of our CRHC research group, who provided useful suggestions for this project: Sandy Hsueh, Zbigniew Kalbarczyk, and Greg Ries.

John Beahan and Leon Alkalai from the Jet Propulsion Laboratory and Jehoshua Bruck from Caltech should also be noted for providing suggestions in developing these experiments.

I would also like to thank Bob Felderman from Myricom, Inc., for his prompt technical assistance about Myrinet.

## TABLE OF CONTENTS

	Page
1. INTRODUCTION . . . . .	1
2. RELATED WORK . . . . .	4
3. EXPERIMENTAL NETWORK TESTBED . . . . .	7
3.1 Myrinet LAN . . . . .	7
3.1.1 Myrinet host interface card . . . . .	8
3.1.2 Myrinet control program . . . . .	10
3.1.3 Myrinet links . . . . .	11
3.1.4 Myrinet switch . . . . .	11
3.2 Laboratory Network . . . . .	12
4. OVERVIEW OF EXPERIMENTS . . . . .	14
4.1 Fault Model . . . . .	14
4.2 Control Program . . . . .	15
4.3 Network Workload Application . . . . .	16
4.4 Fault Injector . . . . .	16
4.5 Fault List . . . . .	16
4.6 Experiments . . . . .	18
5. EXPERIMENTS I . . . . .	19
5.1 Fault List . . . . .	20
5.2 Workload Application . . . . .	21
5.3 Synchronizing Fault Injector and Interface . . . . .	23
5.4 Implementation of Fault Injector and Workload Application . . . . .	25
5.5 Experimental Procedure . . . . .	27
5.6 Results . . . . .	29
5.6.1 Repeatability . . . . .	31

6. EXPERIMENTS II . . . . .	34
6.1 Fault List . . . . .	35
6.2 General Experimental Parameters . . . . .	35
6.3 Experimental Procedures . . . . .	36
6.4 Results . . . . .	37
6.4.1 Effect of code selection . . . . .	39
6.4.2 Effect of workload levels . . . . .	40
6.4.3 Effect of choice of target platform . . . . .	41
7. EXPERIMENTS III . . . . .	43
7.1 Fault Model . . . . .	43
7.2 Variations on MCP . . . . .	44
7.3 Control Program . . . . .	45
7.4 Workload Application . . . . .	45
7.5 Results . . . . .	46
8. SIMULATED NETWORK FAULT INJECTION . . . . .	50
8.1 Overview of Fault Simulation . . . . .	50
8.2 Validation of Simulated Fault Injection . . . . .	52
8.2.1 Validation model . . . . .	52
8.2.2 Validation of revised MCP . . . . .	54
8.2.3 Comparison with modified MCP . . . . .	55
8.3 Discussion . . . . .	57
9. DISCUSSION . . . . .	59
9.1 Host Interface Failure . . . . .	59
9.2 Myrinet Dependability Issues . . . . .	60
9.3 Effects of Fault-Injection Parameters . . . . .	62
10. CONCLUSION . . . . .	65
10.1 Summary . . . . .	65
10.2 Future Work . . . . .	66
APPENDIX A. FAULT LOG EXAMPLE . . . . .	68
APPENDIX B. ILLUSTRATIVE SAMPLE FAULTS AND ERRORS . . . . .	71
B.1 Data Corrupt . . . . .	71
B.2 Host Computer Hang, Example 1 . . . . .	72
B.3 Host Computer Hang, Example 2 . . . . .	72
B.4 MCP Hang . . . . .	73
B.5 Multiple Manifestations . . . . .	73
B.6 Message Dropped, Example 1 . . . . .	74
B.7 Message Dropped, Example 2 . . . . .	74

REFERENCES . . . . . 76

## LIST OF TABLES

Table	Page
5.1: Results of Fault Injection Using Experimental Testbed . . . . .	29
5.2: Breakdown of Faults Resulting in Host Interface Hang . . . . .	31
6.1: Experiment Parameter Specifications . . . . .	37
6.2: Categorization of Fault-Injection Results . . . . .	38
6.3: Breakdown of Fault-Injection Results for Different Code Regions . . .	39
6.4: Breakdown of Fault-Injection Results for Different Workloads . . . . .	41
6.5: Breakdown of Fault-Injection Results for Different Platforms . . . . .	42
7.1: Regions of Data Selected for Fault Injection . . . . .	44
7.2: Effects of Faults in Data Memory . . . . .	48
8.1: Number of Errors by Category for Simulation and SWIFI in First Run . . . . .	54
8.2: Number of Errors by Fault Category for Simulation and SWIFI in Second Run . . . . .	55
8.3: Comparison of Errors without/with Fault Recovery and Avoidance Code . . . . .	56

## LIST OF FIGURES

Figure	Page
3.1: Block Diagram of Host Interface Board Including the LANai Chip . .	9
3.2: Diagram of CRHC Reliable Networks Myrinet Lab . . . . .	13
5.1: Format of Link-Level Data Messages . . . . .	21
5.2: Block Diagram of SWIFI Fault Injector . . . . .	24
5.3: Repeatability of Faults in <i>Message Dropped</i> and <i>Data Corrupted</i> Categories . . . . .	32
5.4: Repeatability of Faults in the <i>Host Interface Hang</i> Category . . . . .	33
A.1: Sample Error Log . . . . .	69

## 1. INTRODUCTION

Dependability has always been an important issue in computer systems. Since the creation of the first computer, people have studied how to assess and improve the dependability of computers. In modern computer systems, the interconnection between computers is becoming more and more important; the exponential growth of the Internet in the past decade is evidence of this trend. Despite the increased importance of the hardware that connects distributed computer nodes, relatively little work has been done to examine the dependability of computer networks themselves.

The purpose of this work is to examine issues in assessing the dependability of high-speed computer networks. To look at these issues, this thesis includes several experiments to test the dependability of a high-speed LAN by injecting faults into the network interface card and observing how errors propagate to applications running on the host computer. Though the experiments were performed entirely using a heterogeneous cluster of computers connected by a Myrinet network [1], the results may be useful in considering the dependability of other networks.

A commonly accepted method for assessing the dependability of computer systems is fault injection. Several forms of fault injection have been used (such as *pin-level fault injection* or *simulated fault injection*) for this purpose. The form of fault injection used in this study is software-implemented fault injection (SWIFI).

The Myrinet is an example of a modern network where message processing resources move from the operating system on the host computer to the interface hardware. Each Myrinet interface card has an embedded custom-VLSI RISC 32-bit microprocessor on the interface card for processing messages. For this reason, the focus of the fault injections is the host interface card. In this research, faults were injected into the on-board memory that the embedded processor uses for its code and for message buffering.

To implement the fault injection, part of this thesis included developing an environment for performing the fault injection. The program allows the user to perform fault-injection experiments with a variety of workload parameters and with several different fault options (such as a target section of data for fault injection).

This study includes the results of four sets of experiments. The first set looked at injecting faults into a small section of instruction code. The second set looked at a broader section of code and varied some experimental parameters. The third set of experiments injected faults into the data section of memory. The final set of experiments was based on the first set and was used to compare the results of the SWIFI study with simulated fault injection.

Results from exploring several factors that may affect the results of fault-injection experiments such as *workload intensity level*, *injection platform*, and *region of fault injection* show that injecting faults into different areas in memory leads to large differences in the fault activation rate [2]. The results also show that, compared to the memory location, workload intensity has less effect on the results and that the injection platform can make a large difference on the most critical set of errors, namely those that propagate to the host computer.

Comparing SWIFI experiments to the simulated fault-injection experiment was helpful in analyzing some of the issues for each form of fault injection for evaluating dependability [3]. The results from these experiments show that simulation and fault injection into the real system agreed most of the time (about 85% of the time) but that the simulation had difficulty predicting the behavior of faults leading to serious failures. For example, some faults caused the host computer to crash in the real network but had no effect on the simulated network.

The remaining chapters are organized as follows. The next chapter reviews related work in the area of network dependability. Chapter 3 describes the Myrinet and testbed network on which the experiments were performed. Chapters 4, 5, and 6 describe three fault-injection experiments performed on the testbed. Chapter 7 describes the experiment that was used to validate a simulated fault-injection experiment. Chapter 8 provides a discussion of the results of the experiments. Chapter 9 summarizes the work, and Chapter 10 provides concluding remarks.

## 2. RELATED WORK

The computer dependability community has thoroughly examined the issues involved in the dependability of processors and links connecting processors. Fault injection has been used for several decades as the primary tool to study the behavior of faults, errors, failures, and recovery [4, 5]. Several studies describe dependability experiments performed on simulated environments [6, 7, 8] when no prototype is available or when injecting faults may cause damage to the system. Fault injection is also performed on experimental testbeds [9, 10, 11, 12]. Although we have accumulated knowledge about the dependability of processors and links between processing nodes, relatively little attention has been given to the networks connecting computers. Given the recent increase in the popularity in distributed computing, this lack of attention is disturbing to the designers of highly critical systems.

Nearly all the studies that look at communication faults have a restricted fault model that is limited to a few common faults such as lost, corrupted, or delayed messages and link failures. DEFINE [6] and NEST [9] simulated the effects of connection faults

in messages. Orchestra [13] injected faults between protocol layers to study their effect on outgoing and incoming messages. DOCTOR [14] provided an environment for conducting fault-injection experiments in a distributed system; the network fault model used dropped, corrupt, and delayed messages. One study [15] measured the effect of communication faults by corrupting message headers in parallel applications.

Other studies have injected faults into computer networks to test fail-silent assumptions. For example, Arlat [16, 17] described fault-injection experiments conducted to validate the fail-silent behavior of the network interface cards in the Delta-4 project. The study used pin-level fault injection in the network interface connectors to test the fail-silent behavior of their redundant interfaces.

An issue that has not been addressed is how to determine a set of communication faults that represents those that can result from real faults. Very little has been done in the area of network dependability studies that involves the particular network components of the interface hardware, the control software, or the software that performs data transfer. We are aware of only one other study [18] where faults are injected into the network hardware to learn how those faults propagate to higher levels; that study used SWIFI to inject faults into a VME-bus-based network of M68k processors. Studies into heterogeneous networks of computers are uncommon.

A few studies have compared different fault-injection methods. For example, one study [19] compared the results obtained using three different physical fault-injection methods and a SWIFI implementation on a processor in a system area network. To

our knowledge, our experiment [3] comparing the simulation-based fault injection and real fault injection was the first paper to compare simulation and SWIFI fault-injection experiments.

### 3. EXPERIMENTAL NETWORK TESTBED

This chapter first provides background information about the Myrinet LAN needed for understanding the experiments performed in the study. The chapter then describes the configuration of the testbed Myrinet network used in the Reliable Networks Laboratory at CRHC.

#### 3.1 Myrinet LAN

The Myrinet is a high-speed, switched local-area network (LAN) [1]. Myrinet is a product of Myricom, Inc. It consists of nodes (host computers with interface cards), high-speed links, and crossbar switches. More detailed information about the Myrinet can be found at Myricom's homepage <http://www.myri.com>.

Since the Myrinet has an open architecture, we had access to the source code and compiler for its key software element (called the Myrinet Control Program or MCP). This level of access was essential for the fault injections performed in this study. It is often difficult to obtain this level of detailed information for commercial networks.

The Myrinet, in general, does not have a fault-tolerant architecture. It does, however, provide some level of error detection and fault tolerance. The data messages are protected by a CRC byte, and the network automatically remaps itself to changes in the network topology. The MCP's instruction memory is protected from the LANai processor so that programming errors do not overwrite code (unlike many processors, a store to the instruction memory is ignored instead of producing a software trap). Similarly, if the LANai attempts to execute an instruction that is not specified in the instruction set architecture (which would produce an 'illegal instruction' trap in most architectures), the instruction is treated as a `nop` instruction. These features were not tested in this thesis. We are unaware of any fault-tolerant features in the interface hardware. All of the Myrinet's fault-tolerant features assume that the processor executes error-free.

### 3.1.1 Myrinet host interface card

Message processing in a Myrinet network takes place in the host interface card. The host interface card is plugged into the host I/O bus and treated as an external I/O device. The main component of the interface card is the LANai microprocessor, a custom-VLSI 32-bit RISC microprocessor. The card also has on-board static RAM. Figure 3.1 shows a diagram of the LANai chip. The LANai processor also has a hardware DMA engine for performing DMA transactions to the host computer's main memory.

The architecture of the LANai processor is similar to typical pipelined RISC processors. The register file has 32 registers. All the registers are addressed as general-purpose

registers, but some registers have special purposes: the program counter, stack pointer, frame pointer, return address, etc. Since the card has only 128 kbytes of physical memory, virtual memory is unnecessary. The processor uses several memory-mapper registers to communicate with the DMA engine and with the physical network interface.

The Myrinet Control Program (MCP) is the software running on the LANai; it performs most of the host interface's functions. The on-board RAM holds the MCP and is used to buffer messages. In this study, the fault injector injected faults into the LANai processor by corrupting elements in the on-board SRAM.

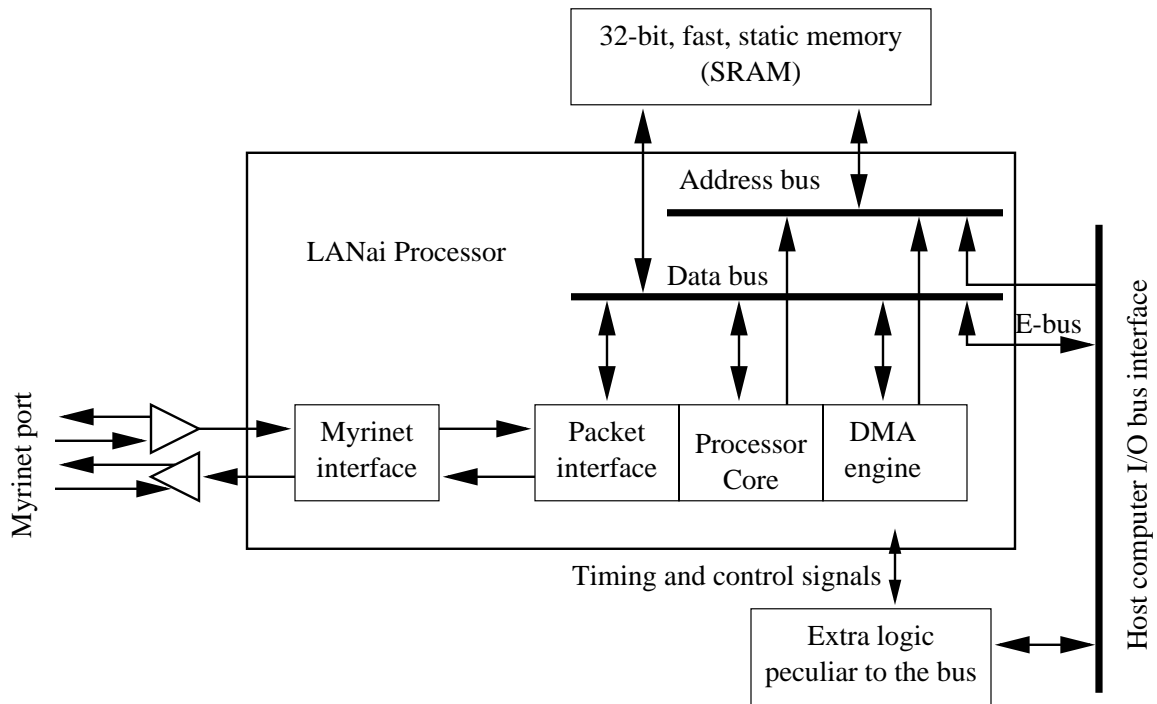


Figure 3.1: Block Diagram of Host Interface Board Including the LANai Chip

### 3.1.2 Myrinet control program

The MCP is the software program that runs on the Myrinet interface card. The program is written, for the most part, in C++. Myricom provided us with a version of GNU g++ that compiles the MCP into LANai executable code. By allowing users to rewrite the MCP, the same hardware can accommodate different protocols. By default, the Myrinet can use either TCP/IP or a more efficient protocol specific to Myrinet, which is referred to as the Myrinet API protocol since it uses API calls that Myrinet provides.

To send a message, the MCP copies the messages from a buffer in the host computer into the “send” queue on the interface card. The host computer’s drivers are responsible for ensuring that there is buffer space for the message and for including the information about the messages, such as the destination and the length of the message. The MCP uses this information to create the message header when it puts the message in the “send” queue. The MCP also assembles scatters into a single message at this point.

The MCP then finds the routing information for the message and puts the entire message and header in the network “send” queue. The message waits there until the network becomes available. At that point, the interface hardware copies the message from the queue to the network.

Upon receiving a message, the MCP copies the message into the “receive” queue. If the message is a control message (e.g., a mapping message, not a data message), then it is processed. If the queue is full, then the message is dropped. When buffer space is

available in main memory, the “host receive” function copies the message from the “host receive” queue to the host computer’s main memory.

The MCP also has functions for dynamically updating each node’s map of the network for routing. Each of these basic functions runs concurrently; the scheduling between the functions is handled through a state machine.

### 3.1.3 Myrinet links

Each node of the Myrinet connects to a switch (or directly to a second node) through a high-speed link. Each link is duplexed and has a maximum throughput of 1.28 + 1.28 Gbps. Each logical link uses 9 wires (8 are for data and 1 is to signal a control symbol). The control symbols that the Myrinet supports are STOP and GO control flow signals. Since it is expected that the data on the links will periodically be corrupted due to electromagnetic noise, the Myrinet adds a fault detection mechanism—the CRC byte. Because faults in the link will be detected, this thesis does not attempt to inject faults into this part of the network.

### 3.1.4 Myrinet switch

Connecting more than two nodes requires a switch. The Myrinet switch is a crossbar switch (which is by definition nonblocking). Today, most LAN switches, unlike the Myrinet, are store and forward switches; these switches add latency to transfer times and add state information to the switch, which may be faulty. To process a message, the switch uses the leading byte of the message to determine where to route the message.

Next, the switch removes the leading byte and routes the message. When the switch routes the message, it recalculates the CRC byte on the messages without the leading byte so that the next hop will have the correct CRC. Finally, the switch replaces the old CRC byte with the new one. These simple functions are all performed in hardware; the switch has no software and maintains no state information about the network. For these reasons and because of the simplicity of the switch, this study does not attempt to inject faults into this part of the network.

### 3.2 Laboratory Network

The experiments were performed the on the Reliable Networks Lab cluster at the Center for Reliable and High-Performance Computing (CRHC). The cluster originally consisted of two Ultra SPARC workstations (with 167-MHz Ultra SPARC processors running Solaris 2.5), two PCs (one with 200-MHz Pentium Pro and one with 133-MHz Pentium, each running Linux 2.0.0), and an 8-port Myrinet switch. (One of the Ultras was replaced with a SPARCstation 20 running Solaris 2.5). For all the experiments, version 3.07 of the Myrinet software (drivers and MCP) was used. The nodes of the network were also connected by Ethernet. A diagram of the entire network is shown in Figure 3.2.

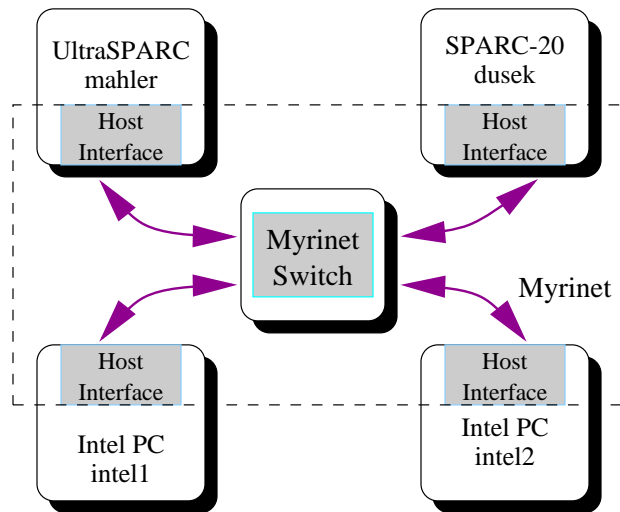


Figure 3.2: Diagram of CRHC Reliable Networks Myrinet Lab

## 4. OVERVIEW OF EXPERIMENTS

This study includes several fault-injection experiments and the Myrinet using the same fault-injection environment. The fault injector evolved from the original version during these experiments. Each experiment used a similar framework which includes

- the control program,
- a network workload application,
- a fault injector, and
- a fault list.

After describing the fault model, the following sections provide a general description of each of these components. The final section gives an overview of the experiments performed using the fault-injection environment.

### 4.1 Fault Model

Ideally, the fault injection would include real faults that might occur in the host interface card. An example of such a fault is a sub-atomic particle hitting the LANai

processor and causing it to produce an incorrect value. Faults of this type might affect data values in the register file or in memory, or they might produce incorrect calculations. It is impossible, however, to reproduce this type of fault in the CPU in a controlled experiment. Thus, an approximation of this fault model was used. The approximation was to inject faults into the instruction executing on the LANai processor. This method is capable of affecting the register file, the memory, and the ALU operands. For the remaining experiments, the fault model was a bit flip of important data structures in the MCP's on-board memory.

In each case, the fault model used in this study is a transient, single-bit fault affecting the network interface card. To inject faults, the host computer wrote the fault directly into the RAM on the interface card.

## 4.2 Control Program

The purpose of the control program is to perform a fault-injection experiment from the user's specifications (a fault list and any other experimental parameters). The control program communicates with the workload program and the fault injector to determine when to inject faults. It is also responsible for restarting interface cards when a card is hung. To do this, the control program needs feedback from the workload program. This is accomplished by having the workload send error messages to the control program. Generally, the control program runs on a remote host (not on the Myrinet network) and uses TCP/IP to communicate with the nodes on the testbed network.

### 4.3 Network Workload Application

In performing fault-injection experiments, there must be some form of network traffic. It has been shown that faults are more likely to manifest in errors with a high workload level than with lower levels [20, 21]. In these experiments, the workload application produces network traffic by continually sending messages on the network and receiving messages. To determine the application-level impact of faults, the network workload is also responsible for error detection.

### 4.4 Fault Injector

The fault injector is the compact program that performs the actual fault injections. The fault injections are triggered by sending a message to the fault injector via a UDP message (or by typing into standard input). In some experiments, the fault injector needs to synchronize itself with the MCP. In other experiments, the fault injector simply injects the fault when it receives a message (from the control program) requesting a fault injection; this version of the fault injector can also remove the fault after waiting a specified period of time.

### 4.5 Fault List

A fault list is a set of faults that are used in a fault-injection experiment. For these experiments, each entry in the fault list is defined by a memory address in the MCP and by a bit position in that instruction. The control program reads the fault list at the

beginning of an experiment and injects each fault from the list one by one, recording the results of each injection.

For an example of a fault, consider a fault that flips the 12th-least-significant bit of the instruction at address `0x4944`. This fault changes the content of the memory location `0x4944` from the value of `0xF6B72000` to the value of `0xF6B72800` – mnemonically, from `ST.H %r9, 0[%r13]` to `ST.H %r9, 2048[%r13]`. This means that the `ST.H` (i.e., store half-word) instruction, instead of writing to the memory location that register 13 points to (i.e., `0[%r13]`), would write to the memory location offset 2048 bytes from the address in register 13. The original instruction, in fact, stores the message type to `0[%r13]` and then uses the message type when adding the frame header. Since the corrupted instruction stores the message type to a different location, the packet will be formed using the message type of the last message, which might be invalid.

To determine which addresses to inject faults into, it is useful to look at two files produced by the LANai compiler (the LANai compiler is GNU g++ cross compiler and GNU binutils with the capability of producing code for the LANai processor). These files are **mcp4.s** and **mcp4.dat**. The first file, **mcp4.s**, is the assembly language file of the MCP; it uses symbolic names for all the functions and most of the variables. The second file, **mcp4.dat**, includes an ASCII version of the MCP object code and the symbol table for the MCP (including the addresses of all the functions and variables in the MCP). Myricom provides functions for looking up the addresses of variables in the file by the symbolic name of the variable.

In some experiments, the same fault is injected more than once to test the repeatability of the experiments. Since this is an uncommon procedure in dependability experiments, the terminology generally used in the fault-injection community leads to some confusion. To be clear, the following terms will be used throughout the paper with these meanings. Each entry in the fault list is called a *fault*. In some of the SWIFI experiments, there are multiple (usually ten) *fault injections* for each fault. The faults are classified in the results section by *fault* rather than by *injection*.

## 4.6 Experiments

The next three chapters describe three of the experiments performed using the fault-injection tool. The first set of experiments was chronologically the first attempt to inject faults into the Myrinet. The purpose of these experiments was to validate a simulated fault-injection experiment on the network. The next set of experiments was an extension of the first set but gained a larger coverage of the MCP and looked at parameters that affect fault injection. The third set of experiments used a variation of the fault-injection method to inject faults into the data structures to gain an even greater coverage of the MCP.

## 5. EXPERIMENTS I

This chapter describes the first fault-injection experiment performed using the fault injector. The original goal for this set of experiments was to determine whether SWIFI is a feasible method of fault injection for the Myrinet network. The early results were very promising, and a variation of this experiment was also used in validating a similar experiment using a simulated model of the network. The simulation procedure and a comparison of the results of each fault-injection method are discussed in Chapter 8. A discussion of the results of this experiment along with the results of the other experiments is provided in Chapter 9.

This chapter is organized as follows. First, two components of the experiment are given: the fault model and the network workload application. The chapter then describes a synchronization problem and how this problem was solved. Next, the chapter describes the procedures for executing the experiments. Then, the results of the experiment are given.

## 5.1 Fault List

The fault model for these experiments was a single-bit flip of the instruction executing on the LANai processor. This method of fault injection is capable of affecting the register file, the memory, and the ALU operands.

The instruction memory is protected from the MCP so that programming errors do not overwrite code. In order to upload the MCP to the LANai processor (usually done at boot time), the host computer has direct access to the memory on the interface card. Thus, only the host computer can write to the instruction memory. This facility was exploited to allow the fault injector to inject faults by corrupting the memory on the interface card.

The faults that can be injected using this implementation of the fault injector are restricted to the target section of code determined at compile time. The section of code used was part of the `hostSend()` function. There are several reasons why this section was used. First, this section represents a large share of the dynamic execution of the MCP. Second, the functionality of this code segment affects the processing of each message. Third, this was the only function that the simulated network model could execute when the experiment began.

The `hostSend()` function takes a message from a buffer in the host computer's main memory after the application issues a particular API call. It then creates a link-level header from information passed to the API call, such as the destination and length of

the message. Next, it copies the header and the message into the “send” queue in the on-board memory (on the interface card).

This section of code contains 65 assembly code instructions. The fault list contains all possible single-bit flips for the 65 instructions. Since there are 32 bits per instruction, the fault list contains a total of 2080 different faults.

## 5.2 Workload Application

The original workload application was designed to be simple and to create high-throughput network traffic. To meet these goals, a simple protocol was created. The format for protocol is shown in Figure 5.1. Each packet is encapsulated inside Myrinet’s data-link message. The first field is the protocol ID; this field identifies the protocol so that new protocols can be used in the future. The next field is a sequence number; this number is different for each message. The sequence number is used to detect dropped messages; it also detects messages arriving out of order, although the Myrinet protocol does not support out-of-order message transmission. The rest of the message is the data payload. The length of the data is set in the data-link level header.



Figure 5.1: Format of Link-Level Data Messages

The workload program uses this protocol to send data from one host to another. The data that is transmitted is a simple repeated test pattern consisting of a sequence of 64

sequential ASCII characters. Messages are transmitted as soon as there is room in the send buffer. Each data message uses 8448 bytes of data, which is the maximum transfer unit (MTU) of the network. Since the workload program that is receiving the messages knows what the data should be, it can detect errors in the data. The program uses the API calls that are distributed with the Myrinet software to send data over the network.

Since the Myrinet has a small buffer capacity relative to the the high speed at which messages are transmitted, messages are frequently dropped due to lack of buffer space. When the workload generator runs at its highest rate (about 325 Mbps when the data part of the message payload is *not* processed), the interface drops messages, often making it difficult to determine whether the messages are dropped as a result of a fault or lack of buffer space. To solve this problem, a delay between sending messages was added to slow the rate at which messages are sent. The workload was slowed until the frequency of buffer overflows was low. This data rate was determined to be approximately 100-150 Mbps along each link. A better fix, implemented in later experiments, was to check the MCP counter that records the number of messages dropped due to buffer overflow.

The workload program provides the mechanism for error detection. Since the program expects to receive data continuously, it can determine that an interface is hung (or that there was a link failure) if it does not receive data for a specified timeout period (e.g, one second). The API calls return an error value when the MCP restarts. To summarize, the errors that the workload programs detect include dropped messages, corrupt messages, interface hangs, and MCP restarts.

### 5.3 Synchronizing Fault Injector and Interface

To inject faults into the instruction that is executing on the LANai processor, the fault injector, which is running on the host computer, needs to synchronize with the LANai processor. This synchronization proved to be a challenging obstacle. To help with the synchronization, we confined the fault injection to one section of code, usually a single function that was selected when compiling the MCP. The code in the MCP is, generally speaking, very linear in that it executes relatively large blocks of code with very few control flow branches (these blocks are larger than what one would expect to find in general purpose code). Thus, if the host computer knows when the MCP is about to enter or leave the section of code, it can inject the fault anywhere in the code section and restore the correct instruction after the section finishes executing. Thus, the fault injector can inject a fault into the instruction that is executing on the LANai processor without disturbing the execution on the processor. This is how faults were injected.

To facilitate the synchronization, the host computer and the MCP communicate by setting flags in the interface memory. Each time the MCP executes the target section of code, it checks the flags to see whether there was a fault injection pending. If so, then the program signals the host computer to write the corrupt instruction into the LANai's memory space by setting one of the flags. Then, the MCP executes the code section without interruption (including the corrupt instruction if a fault was injected). Finally, the fault injector restores the correct value of the instruction after the code

section has been executed. The fault injector is implemented in this manner to minimize the disturbance on the system when injecting faults.

Figure 5.2 shows the block diagram and the flow of control in the fault injector. The flags used for the synchronization are shown below:

- **FI\_Request** *request for fault injection*: The fault injector sets this flag to notify the MCP that the user wants to inject a fault. Each code region has its own request flag.
- **FI\_Ready** *request acknowledged*: The MCP sets this flag to acknowledge the injection request before entering the code region.
- **FI\_Done** *fault injection completed*: After injecting the fault, the fault injector sets this flag to inform the MCP that the fault has been injected and that the MCP should continue to execute the code region.
- **FI\_Execised** *exercised the fault*: The MCP sets this flag when it completes the code region to inform the fault injector that the fault has been exercised.

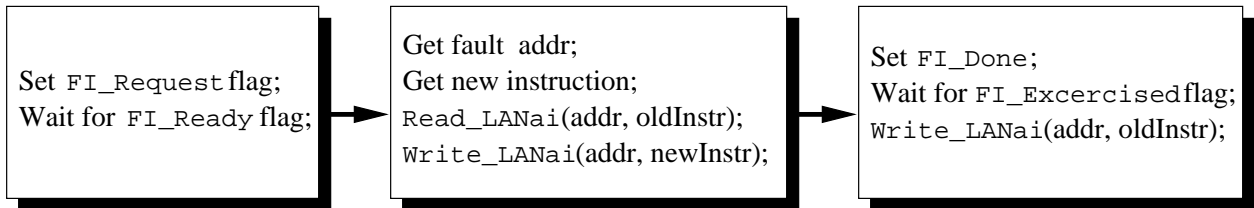


Figure 5.2: Block Diagram of SWIFI Fault Injector

Synchronizing the fault injector with the MCP requires minor modifications of the MCP. To allow fault injection into a specified region of code, these modifications include adding two functions to the MCP. The first function, inserted before the region of code, checks the **FI\_Request** flag each time the MCP calls the specified region. If the fault injector has requested a fault for that region, this function acknowledges the request by setting **FI\_Ready** and waiting for the fault injector to inject the fault and set **FI\_Done**.

The MCP then executes the region of code containing the corrupt instruction. The second function is called when the MCP finishes executing the region of code. It asserts `FI_Exercised` so that the fault injector can restore the original instruction. This sequence of events completes a single fault injection.

#### 5.4 Implementation of Fault Injector and Workload Application

Implementing the fault injector and the workload application proved to be another challenging task. By themselves, implementing each the fault injector, a process to send data, and a process to receive data is straightforward. The difficulty is in implementing all of them efficiently using the Myrinet API. The cause of the difficulties is that each task requires the program to poll the interface card frequently.

To send data using the API functions, the program needs to poll the interface card to determine when data has been sent, freeing space in the “send” buffer. To receive data using the API, the program needs to poll the interface card to determine when new data has arrived in the “receive” buffer. To inject faults, the program also needs to poll the interface card for the synchronization process. The naive solution to accomplish running all three programs together is to use a different process for each task and allow the multitasking operating system to schedule the tasks so that they each receive equal CPU cycles. The problem with this approach is in the frequency at which Solaris and Linux reschedule processes. At the maximum rate, the Myrinet can average less than

500  $\mu$ s per message (for 8448-byte messages); UNIX operating systems, however, allow processes to run 10 ms before rescheduling.

Since it is not possible to run these tasks in multiple processes, they must run as a single process. To accomplish this, a new process combines these functions and provides a user interface through TCP/IP and standard input. The application is basically a scheduler (like a simplified multitasking operating system) that allows the programmer to write tasks and add them to the set of tasks that the application runs. No task is allowed to block, but it may return control to the scheduler when polling. The application also provides a primitive form of intertask communication to allow I/O between a process that checks for user input and other processes. The programmer is responsible for determining where the task might block (e.g., poll on the interface card) and to ensure that the task returns control to the scheduler upon reaching those points.

One task, for example, sends data. As mentioned above, there is a point in the code at which the task needs to poll the interface card and wait for buffer space to become available. The code was redesigned so that it could save all of its important state information (e.g., the fact there is a message still waiting to be sent and the fact that it is waiting for space in the send buffer) and return control to the scheduler when it needs to wait for a free buffer. At that point, the scheduler chooses the next task by round robin.

A more complicated example is the fault injector. The fault injector can block at any of several points: waiting for the control program to request the next fault to inject,

waiting for the MCP to enter the target code region, or waiting for the MCP to execute the target code region. The fault-injector task needs to record which of these states it is in when it releases control to the scheduler. This can be done by simply enumerating all the possible rescheduling points, saving that number, and executing from the point corresponding to that number the next time the task is scheduled.

Note that because the scheduling is handled by `return` statements in the high-level language (C++), information such as the PC and the values in the register file do not need to be saved.

## 5.5 Experimental Procedure

For these experiments, all four network hosts were used. The workload program ran on each node such that every node sent and received data to and from a different node (for example, the second node sent data to the third node and received data from the first node). This maximized the network throughput by having all four machines sending data at the same time. The fault injector ran on only one of the nodes (for this experiment, that node was one of the SPARC Ultras).

Since the entire state of the hardware is too complex to be controlled in real time, the same fault is not guaranteed to have the same result for each injection. For example, the contents of dead registers cannot be controlled while the processor is operating. Thus, to determine the repeatability of the effects due to the injected faults, each fault was injected ten times (or more if there was reason to believe that the result was incorrect,

such as another user running a program during the experiment). Using the terminology described in Section 4.5, the fault list contains 2080 *faults*, and for each fault there are ten *fault injections*, or 20,800 *fault injections* for the whole experiment. When describing the results of the experiment, the effect of each fault is classified by *fault* rather than by *fault injection*.

For each fault, the control program creates a log with the error messages from each node. An example of one of these logs is given in Appendix A. The log for each fault is used to categorize each fault in one of the following ways:

- **No Impact** – The fault does not cause errors or a failure in any of the 10 injections.
- **Message Dropped** – The fault causes a message to be dropped.
- **Data Corrupted** – The fault causes the sending of a message with incorrect data. (Generally, this means that an old message was sent in place of the current message.)
- **Restart MCP** – The fault causes the MCP to restart itself (the application continues to function).
- **Host Interface Hung** – For at least some of the injections, the fault causes a host interface to fail to operate properly until it is reset. Host interface hangs can be further classified as (1) host MCP unable to send or receive messages, (2) host MCP able to send messages but not receive messages, (3) host MCP able to receive messages but not send messages, (4) MCP on remote interface hangs, (5) MCP sends corrupt message before hanging, or (6) MCP repeatedly restarts itself.
- **Host Computer Crash** – The fault crashes the host system.
- **Varied Errors** – The fault results in different kinds of errors for different injections.
- **Other Error** – The fault cannot be classified into one of these error categories.

## 5.6 Results

Table 5.1 summarizes the results of the experiment. The table shows the number of faults that fell into each category, the percentage of all faults that fell into each category, and the percentage of error-producing faults (i.e., those that caused detectable errors in this study) that fell into each category.

Table 5.1: Results of Fault Injection Using Experimental Testbed

Fault-Result Category	Count	% of Injections	% of Errors
Message Dropped	176	8.5	20
Data Corrupted	88	4.2	10
Restart MCP	65	3.1	7.4
Host Interface Hung	514	24.6	42.7
Host Computer Crash	9	.43	1.0
Varied Errors	22	1.1	2.5
Other Error	1	.05	.11
No Impact	1205	57.9	n.a.
Total	2080	100	100

From the table, we can see that 875 of the 2080 faults (42%) resulted in some type of error, and about 59% of those faults causing errors were some type of host interface failures. Only about 1% of the faults fell into the *Varied Errors* or *Other Error* category. The only fault representing the *Other Error* category caused the faulty node to resend a set of four messages several times; this is the result of a fault corrupting the message queue data structure.

Fault containment is critical for fault-tolerant systems. In this study, faults propagated from the host interface in a few ways. As shown in the table, nine of the faults

resulted in a crash of the host computer. Thirty of the faults propagated from the node receiving the fault injection to a different node and hung the interface on the remote node.

The 65 faults that caused the interface to reset had only a minor effect on the application; other than the lost throughput while the MCP reinitialized (a small fraction of a second), the application was unaffected. The 176 faults causing dropped messages and the 88 faults causing corrupt messages also had a negligible effect on the performance of the network.

Table 5.2 shows several variations of the category *Host Interface Hung* and how often each occurred. The largest of these variations (*Silent MCP Hang*) was the “normal” host interface hang, in which the MCP stopped communicating with the host computer and the network until the host reset the MCP. Some cases (31% of the hangs) affected only some functionality of the MCP. For example, in 146 of these faults, the MCP continued to process incoming messages but ceased to send outgoing messages. In 13% of all hangs, the fault caused a persistent change to the MCP software that caused the MCP to restart repeatedly until it was manually reloaded. The other faults displayed some kind of propagation: 30 faults hung an interface after receiving a message from the node where the fault was injected, and in 25 faults, the interface sent a corrupt message before hanging.

Table 5.2: Breakdown of Faults Resulting in Host Interface Hang

Fault-Result Category	Count	% of Hangs
Silent MCP hang	229	44.6
MCP stops sending data	146	28.4
MCP stops receiving data	15	2.9
Remote MCP hang	30	5.8
Corruption before hang	25	4.9
MCP loops on restart	69	13.4
Total	514	100

### 5.6.1 Repeatability

Recall that, due to uncertainties in controlling the exact state of the interface when injecting a fault, repeated injections of the same fault are not guaranteed to have the same effect. In order to evaluate repeatability, each fault was injected ten times. The majority of faults displayed a high degree of repeatability. Sixty-eight percent of the faults produced identical results for all ten injections. Moreover, 77% of the faults produced the same result in at least 8 of the 10 injections. Of the injections that caused an error, only the faults in the *Varied Errors*, *Host Computer Crash*, and *Other Error* categories caused different errors. These faults include only 32 faults (1.5% of the faults).

Figure 5.3 gives a breakdown of the faults in the *Message Dropped* and *Data Corrupted* categories to show the repeatability of fault effects from different injections. In order to quantify repeatability, each fault in the category was subdivided by the number of injections that activated the fault. The  $x$ -axis depicts the number of message drops or message corruptions that resulted from the ten injections; the  $y$ -axis is the number of faults that produced the given error for  $x$  of ten injections. For example, the two bars

above number 7 in the figure show that 40 faults caused dropped messages for 7 of 10 injections and 14 faults caused corrupt messages for 7 of 10 injections. The histogram shows that if a fault causes a message drop or data corruption, then it is likely to cause the same result for most or all of the injections.

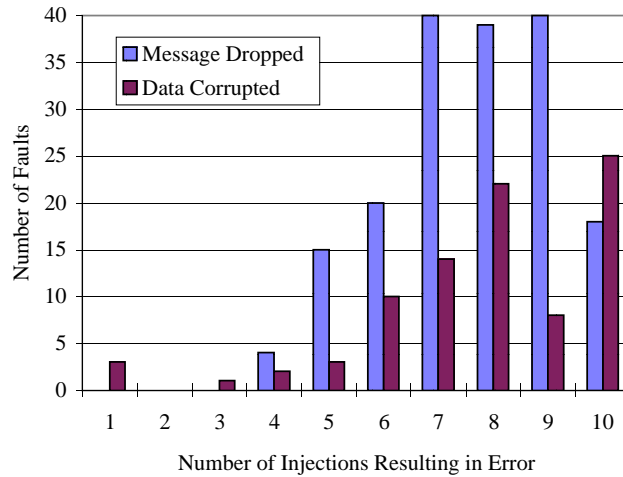


Figure 5.3: Repeatability of Faults in *Message Dropped* and *Data Corrupted* Categories

Figure 5.4 is similar to Figure 5.3, but it uses the category *Host Interface Hang* instead. This histogram shows that interface hangs are not as repeatable as message drops or corruption. Some faults hang the MCP every time they are injected, and other faults only hang the MCP about half the time.

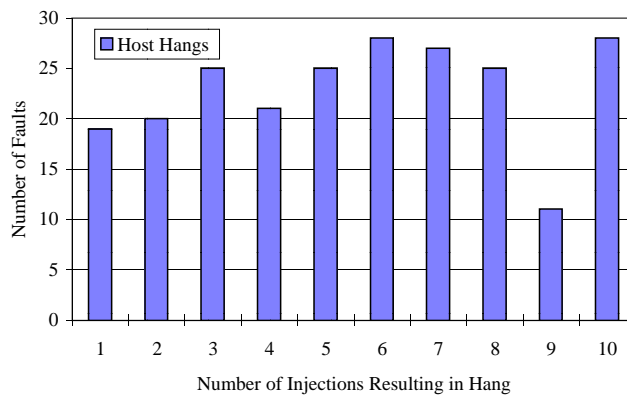


Figure 5.4: Repeatability of Faults in the *Host Interface Hang* Category

## 6. EXPERIMENTS II

The chapter describes the second set of fault-injection experiments. These experiments are similar to those in the Chapter 5, but the effects of certain experimental parameters are tested. These parameters include target section of code, choice of target host platform, and workload intensity level. A discussion of these results along with the results of the other experiments is provided in Chapter 9.

The control program and fault injector used in for these experiments is identical to the one used in the first set of experiments. The workload application is also identical to the one in the first set of experiments, except that the intensity level of the workload is varied. The network contains the two Ultra SPARC workstations and two Intel PCs, as described in Section 3.2.

## 6.1 Fault List

To gain better code coverage, faults are injected into the entire `hostSend()` function and into the `hostReceive()` function. The `hostReceive()` function copies messages from the receive buffer on the interface card to the host computer, the reverse of `hostSend()`.

One fault list contains 500 faults randomly selected from `hostSend()`, and another contains 500 faults randomly selected from `hostReceive()`. As in the previous section, each fault is an address in the given section of code and the bit position of the bit that will be flipped.

## 6.2 General Experimental Parameters

The experiment described this chapter includes six trials. The trials differ in the value of various experimental parameters. The following paragraphs give a specific definition of each parameter as it applies to this study and the values it may assume.

The **network workload** specifies the sequence of messages and the destination node for each node in the network during one experiment. This parameter has only two values for the experiments, high and low. The data and size of messages is the same for each workload level, but the rate that messages are sent varies. The difference between the two workloads is that the high workload sends messages at the maximum rate the receiving node can accept (the sending interface's message buffers are kept full), while the

low workload specifies a much lower rate, which typically uses only one of the message buffers of the sending interface at a time.

The **target platform** parameter refers to the type of host computer connected to the interface undergoing fault injection. The host platform parameter can take one of two values: an interface is either connected to a SPARC Ultra or to an Intel PC.

The **code selection** parameter defines the region of code in the MCP at which faults are injected. This parameter takes two values: first, the “broad send” uses the fault list from the *host send* code (317 instructions); second, the “broad receive” uses the fault list from the *host receive* code (922 instructions). The entire MCP contains about 18,400 instructions, but these target regions of code represent a larger share of the dynamic execution of the code. In the results and discussion, the code section from the experiment in Chapter 5 is referred to as the “focused send,” since its fault list was restricted to the first part of the `hostSend()` code.

### 6.3 Experimental Procedures

This experiment uses six new experiments as well as the results from the first experiment to meet our objectives of comparing the effect of the experiment parameters. These experiments are represented by letters in Table 6.1. For example, Experiment A injected 400 faults (4000 injections) into the “broad send” region of the MCP on an Ultra while running a high workload. Since the “focused send” experiment G is included in

the “broad send,” this experiment was only performed for the base conditions (using a high workload on a SPARC platform).

Table 6.1: Experiment Parameter Specifications

Experiment	Platform	W.L. intensity	Code region	No. faults	No. inject.
A	Ultra	High	Broad send	400	4000
B	Ultra	High	Broad recv	400	4000
C	Ultra	Low	Broad send	400	4000
D	Ultra	Low	Broad recv	400	4000
E	PC	High	Broad send	400	4000
F	PC	High	Broad recv	400	4000
G	Ultra	High	Focused send	500	5000

## 6.4 Results

Table 6.2 shows the categories used to classify the results of each fault injection. The first five categories are considered severe. *MCP restart* is somewhat less severe, since it requires no action from the application to recover (device drivers can hide the effects of a restart from the application) and only the current messages in the buffer are lost. Categories such as *message dropped* and *data corrupt* can be easily remedied by network protocols (by resending corrupt messages). The fault injection on the simulation, unlike the real system, is deterministic (each unique fault produces the same results for every injection); it does not have the categories of *hang sometimes* and *multiple manifestations*.

Faults can cause *no error* for several reasons. First, the fault injector cannot guarantee that all corrupt instructions execute, since the fault address is determined without knowledge of branching. Second, since the corrupt instruction differs from the good one

Table 6.2: Categorization of Fault-Injection Results

Fault-injection result	Characterization
MCP hang	The host interface hung in at least half the experiments.
MCP hang sometimes	The host interface hung in some, but less than half, of the experiments.
Hang remote MCP	A fault on one host interface caused a different host interface to hang.
Host computer hang	The host computer either hung or rebooted itself.
Multiple manifestations	Some combination of the above results was seen.
MCP restart	The MCP on the host interface restarted.
Message dropped	Some message was not received.
Data corrupt	A valid message was received, but its contents had been modified.
Other errors	Some other error occurred.
No error	No error was detected.

in only one bit, the difference in the operation may be unimportant. For example, many instructions have a field that specifies whether to set a condition flag; if a condition flag is set before the next time it is used, then the fault is masked. Also, some instructions have unused fields. Third, many faults change a `nop` instruction into another instruction that also performs no operation. Furthermore, many instructions can perform similar functions even when corrupted. For example, if an instruction compares a register value to some other value but the fault causes a different register to be used, there is a good chance that the same condition code may be set with or without the fault. Similarly, if an opcode makes a comparison with a random register instead of register 0 (which is hardwired to 0), there is a good chance that the corrupt register is also set to 0.

Appendix B illustrates several detailed examples of faults causing many of these error result categories.

## 6.4.1 Effect of code selection

Table 6.3 gives the combined results for injecting into different regions of code from the SWIFI experiments. The table is divided into three sets of data, one for each code-selection parameter value used in the experiment. For each parameter, the table shows (a) the number of times fault results fell into each category and (b) the ratio of the number in (a) to the number of injections resulting in errors.

Table 6.3: Breakdown of Fault-Injection Results for Different Code Regions

Fault Injection Result	Send Code		Receive Code		Focused Send	
	Count	Error %	Count	Error %	Count	Error %
MCP hang	14	33%	9	31%	79	32%
MCP hang sometimes	5	12%	5	17%	60	24%
Host computer hang	3	7%	3	10%	1	1%
Hang remote MCP	1	2%	0	0%	4	2%
Multiple manifest.	3	7%	0	0%	12	5%
MCP restart	4	9%	3	10%	19	8%
Message dropped	8	19%	5	17%	55	22%
Data corrupt	5	12%	3	10%	19	8%
Other errors	0	0%	1	3%	0	0%
No error	357	-	371	-	251	-
Total	400	100%	400	100%	500	100%

There are two major points from this data that should be noted. First, the activation rate for the focused region is much greater than that of the broad region. For the focused region, nearly half of the faults produced an error, but in the broad region only 43 out of 400 (11%) of the faults produced an error. One reason for this difference in activation rates is that in the focused region each corrupt instruction is executed; while in the broad region, due to control flow branches, the corrupt instruction might not be executed.

The second point is that between the two broad regions, the distribution of the results of faults that caused errors is very similar. The percentage of *MCP hang sometimes* is a bit higher in the focused region, but in the remaining categories, either the numbers are too small or the differences are too small to be statistically significant.

The results indicate that despite the similarities, different scopes of injections should be used in dependability analyses.

#### 6.4.2 Effect of workload levels

In Table 6.4, the number of fault results for each category is given for both workload levels. The right column shows, for each category, the total number of faults that had the same result for each workload level. The heavy workload sends messages at an average rate of 150 Mbps; the rate for the low workload is about 2-3 Mbps. The table shows that the results from injecting faults into the interface of an Ultra with a very heavy workload are similar to the same injections with a light workload for the less severe categories.

Although we expected more hangs using the heavy workload, the results for *MCP hangs* are in fact the opposite. We were unable to find the reason for this pattern. However, the *MCP hang sometimes* category offers some insight. The *MCP hang sometimes* category almost disappears for the light workload. The light workload does not exercise certain parts of the program (e.g., code to handle a full buffer might never execute). Apparently, the conditions needed for such a fault to cause an error only occur when the queue lengths are large or when parts of the MCP compete for resources. Since there

are some errors that only appear in the heavy workload and others that only appear in the light workload, multiple workloads should be considered when high fault coverage is needed.

Table 6.4: Breakdown of Fault-Injection Results for Different Workloads

Result	Heavy Workload	Light Workload	Number of Matches
MCP hang	23	27	22
MCP hang sometimes	10	1	1
Host computer hang	6	4	4
Hang remote MCP	1	1	1
Multiple manifestations	3	4	3
MCP restart	7	7	7
Message dropped	13	13	13
Data corrupt	8	8	8
Other errors	1	0	0
No error	728	735	724
Total	800	800	783

#### 6.4.3 Effect of choice of target platform

The results of the fault injection to each platform are given in Table 6.5. The right-most column gives the number of faults for each category that matched. Since the faults are injected to the LANai processor independently of the host CPU, we expect most faults to produce the same result on each platform. Each platform, however, has its own I/O bus and operating system, which could produce different results.

The network needs one node, which is automatically selected, to perform extra functions to maintain routing information. In our experiments, a PC was selected to perform these functions. These extra functions may be another source of differences in the results.

Table 6.5: Breakdown of Fault-Injection Results for Different Platforms

Result	Ultra	PC	Matches
MCP hang	23	20	18
MCP hang sometimes	10	11	5
Host computer hang	6	12	5
Hang remote MCP	1	1	1
Multiple manifestations	3	5	3
MCP restart	7	8	7
Message dropped	13	13	12
Data corrupt	8	8	7
Other errors	1	3	1
No error	728	719	714
Total	800	800	773

As in the previous cases, the number of errors is small relative to the number of injections. The relative number of matches is lower than in previous cases. For example, taking the *MCP hang* and *MCP hang sometimes* categories together, the percentage of matches is less than 70%. It appears from our results that the type of host platform makes a significant difference in the dependability behavior of the network. Also, and perhaps more important, obtaining results for one platform type does not necessarily predict the behavior on another platform type. Further studies would be needed to verify this result.

## 7. EXPERIMENTS III

This chapter describes a third set of fault-injection experiments. The basic difference between the experiments described in this chapter and previous ones is that faults described in this chapter corrupt the data memory in the interface, not instruction memory.

### 7.1 Fault Model

The fault model used in this set of experiments is a transient single-bit flip error in the on-board memory on the interface card. This is a common fault model for dependability studies [4]. Faults are injected by writing the new value into memory and then, after a specified period of time, restoring the data value unless the processor had updated the data during that period of time.

The fault lists used in the experiment trials in this set of experiments were created by finding interesting data structures in the symbol table of the MCP. Table 7.1 lists six sets of data regions used to create fault lists.

Table 7.1: Regions of Data Selected for Fault Injection

Data Section	Description
map table	lookup table for address to node
route table	lookup table for route to node
host send variables	set of data structures used by host send function
host receive variables	set of data structures used by host receive function
net send variables	set of data structures used by net send function
net receive variables	set of data structures used by net receive function

Each of these six areas of memory uses a different fault list. Each fault list contains one entry for the least significant bit, the second least significant bit, and the most significant bit. The fault lasts for one second before the fault injector restores the good value of the memory.

## 7.2 Variations on MCP

Three versions of the MCP were used in this experiment. The first version is the default version that was used in the previous experiments. The second version uses the same code but is compiled with the “DEBUG” flag, which includes several run-time assertion checks. The third version uses three error recovery and avoidance routines based on errors detected in the earlier experiments.

One of these routines adds a check to incoming messages for a bug that caused any node to hang upon receiving a message with the length field in the header set to 0. The other two routines add timeouts to limit the length of time the MCP waits for (1) the DMA to become available and (2) the send buffer to become available when messages are waiting.

### 7.3 Control Program

The control program used in these experiments is similar to the one used in the first set of experiments. Some features were added to make the interface easier to use. The framework of the control program described for the other experiments also describes this program.

### 7.4 Workload Application

This version of the fault-injection environment allows the fault injector and the workload to run independently. Thus, any workload program can be used. In fact, this fault-injection environment has been used to inject faults into benchmark programs using two variations of the Myrinet API (the API and BPI are two interfaces that Myricom provides) and a few TCP/IP programs including a video server application running on an array of two redundant Myrinet networks demonstrating fault resilient data encoding using X-Codes [22].

Besides providing network traffic, the workload program is also responsible for monitoring errors. Most TCP/IP programs do not record error information (such as dropped messages), since the protocol should recover from any errors. For this reason, they are not used in this study.

For this set of experiments, the workload application continually sends data to randomly chosen destinations nodes and reads any incoming messages. The data message

includes a copy of the data structure that maintains link information (including the address of each node, the message sequence number, the time since message was sent and received, and the current guess of the link status). Some of these fields (message sequence number, message length, and destination address) are used for application-level error detection. For each logical link, if either node does not receive any data for some timeout period (set to 1.5 seconds), the node logs a message saying that the link is down.

The network contains one Ultra SPARC workstation, one SPARCstation20, and one Intel Pentium PC, as described in Section 3.2.

## 7.5 Results

The results of the experiments are given first by fault list (one for each data section) and then by MCP version (where “default” is the unmodified MCP, “debug” is the unmodified MCP compiled with the debugging flag, and “modified” is the version of the MCP with the error avoidance schemes added).

The errors categories that the workload and control program detect are as follows:

- **Temporary Hang** – The fault caused the MCP to hang; when the fault is removed the MCP resumes normal execution.
- **Hang** – The fault caused the MCP to hang; the interface needed to be reset to resume execution.
- **Reset** – The fault caused the MCP to restart.
- **Dropped Messages** – The fault causes some messages to be dropped.
- **Assert** – The fault triggers a run-time assertion statement in the MCP.
- **No Error** – No error is detected for the fault.

The results of these runs are shown in Table 7.2. The leftmost column gives the version of the MCP for that row of results. The next four columns give the number of faults that resulted for each of the error categories. The fifth column shows the number of faults that caused a run-time assertion failure. The last column shows the number of fault injections in that trial. Since different versions of the code were used, some fault lists have a different number of faults for different MCP versions on the same set of data structures. The last set of results is the sum over the first six sets.

For the “net send” section, some faults (three per run) corrupted the counters that are used by the error detector. These values are marked with the “†” symbol. Similarly, for the “net receive” section, 18 faults per run corrupted the counters; these values are marked with the “†” symbol. The values in the table for these data represent the number of errors that actually caused an error, and not the faults that only affected the counter directly.

From these results, it is apparent that the fault-activation level is dependent on the target section of memory. For the “net receive” section, 138 of the 1497 faults (9.2%) produced an error. But for the “host receive” section, only 13 of the 2241 faults (0.58%) produced any error.

The data in the table indicate that the run-time assertions were useful in detecting some of the errors that caused the MCP to hang or drop messages, but several more errors escaped the assertions. The run-time checks detected 14 of the errors but allowed

Table 7.2: Effects of Faults in Data Memory

MCP version	Temp. Hang	Hang	Reset	Drops	Assert	Num. Injections
host table						
default	19	0	0	7	0	407
debug	1	0	0	19	0	918
modified	1	0	0	1	0	1125
route table						
default	10	0	0	7	0	1207
debug	0	1	3	0	2	903
modified	1	0	0	0	0	903
host send						
default	2	4	2	1	0	441
debug	3	3	0	0	3	441
modified	1	5	2	0	0	450
host receive						
default	3	0	2	0	0	741
debug	0	0	1	0	0	741
modified	0	7	0	0	0	759
net send						
default	8	14	0	12 <sup>†</sup>	0	786
debug	18	3	5	3 <sup>†</sup>	2	789
modified	9	9	1	5 <sup>†</sup>	0	813
net receive						
default	22	14	4	5 <sup>‡</sup>	0	495
debug	16	17	10	2 <sup>‡</sup>	7	495
modified	19	14	5	3 <sup>‡</sup>	0	507
total						
default	64	32	8	32	0	4077
debug	38	17	19	24	14	4287
modified	31	37	8	26	0	3503

<sup>†</sup>Does not include the 3 faults that corrupted a counter used to detect the error.

<sup>‡</sup>Does not include the 18 faults that corrupted the counters used to detect errors.

98 errors to pass (thus the coverage of these checks is 12.5%). It is possible that the fault injector injected faults into data added for the run-time checks.

The modified version of the MCP was able to avoid many of the faults that hung the original MCP. This version avoided errors for more than half of the faults (33 of 64) in the “temporary hang” category. But there was no improvement for those faults that lead to permanent MCP hangs. We are not sure why this is.

## 8. SIMULATED NETWORK FAULT INJECTION

Part of the original motivation for this thesis was to use fault injection on a real Myrinet network to validate the results of a simulated fault-injection study of the same network. The simulated fault-injection experiments served as the basis of Greg Ries's Ph.D. thesis [23]. Details of the simulated fault-injection setup can also be found in [8, 2].

This chapter first provides an overview of the Myrinet simulation and the simulated fault-injection experiment. The next section describes the comparison of the results of the experiment comparing the simulation and SWIFI fault injections and the differences between the SWIFI experiment in Chapter 5 and the one used in this comparison.

### 8.1 Overview of Fault Simulation

To understand the comparison of between the two fault-injection methods (simulated fault-injection and SWIFI), a basic overview of the simulated fault-injection experiment is provided here.

The simulation fault-injection program includes three major components. First, it includes a cycle-accurate simulation of the LANai processor on a single Myrinet host interface card. Second, it includes a program that simulates a workload on the interface card and injects faults into the cycle-accurate simulation. Third, it includes a simulation of the four-node Myrinet network using the fault-simulation tool DEPEND [24, 25].

Myricom, Inc., supplied the cycle-accurate simulator. It accurately simulates the LANai processor running the actual MCP program. This simulation, however, includes neither the interaction between the Myrinet interface card and the host computer nor the traffic between the simulated node and the rest of the network.

The cycle-accurate simulation was extended to add the effect of a network workload and to allow fault injection. To simulate the effect of a workload, the state of the simulated MCP was initialized to appear to have messages in its message buffers. Faults were injected to be as similar as possible to the fault injections in the real network. To inject a fault, the simulation used the same fault list used in the SWIFI experiment; that is, the simulation used the same version of the MCP and injected faults by flipping the same bits. The effect of the faults was recorded as a “fault dictionary entry.” The fault dictionary includes a list of all the variables in memory (or register file) that are different between the faulty run and a “gold run” (i.e., a run without faults).

Since the cycle-accurate simulation is slow (about ten seconds per fault injection) and since it only simulates a single interface, a second level of simulation was also used. This level of the simulation uses the dependability simulation tool DEPEND. The model

includes all four nodes of the network and the switch connecting the networks. It takes a fault dictionary entry as an input to determine the effect of each fault on the network.

## 8.2 Validation of Simulated Fault Injection

The SWIFI experiments used for this comparison were carried out in nearly the same manner as those in Chapter 5. The major difference is in the fault list. This experiment used only 500 faults randomly chosen from the 2080 entry fault list used in the previous experiment. One difference between the two experiments is that the laboratory network contained a second Sparc Ultra in place of the SPARCstation20.

As mentioned above, both fault injectors (simulated and SWIFI) injected the same set of faults into the network and recorded the application-level results for each fault. But since simulation is completely deterministic, faults were only injected once, whereas the SWIFI method made 10 independent fault injections for each fault to test repeatability.

This experiment is divided into three sections. The first describes the first validation experiment. The second shows the results of the same experiments after some modifications were made to the simulator. The last shows the same comparison using a modified version of the MCP.

### 8.2.1 Validation model

In the simulation experiment, some of these injections (48 injections) had to be discarded because they caused the simulator to crash (e.g., the simulator had a segmentation

fault). Thirty-six (73%) of the faults that crashed the simulator also caused *MCP hang*, *MCP hang sometimes*, or *MCP restart* in the SWIFI method. Also, 67 injections resulted in *MCP hang sometimes* or *multiple manifestations* for the SWIFI and had to be discarded because the simulation did not include these categories. The breakdown for the remaining 385 faults is presented in Table 8.1.

The leftmost column of Table 8.1 shows the fault-injection results (based on the categories in Table 6.2). The next column gives the number of faults resulting in each category observed in the simulations. The SWIFI column shows the number of faults resulting in each selected category in the real system. Next, the match column shows the number of times the simulation and the real fault-injection results were identical and the percentage of matches. The last column show the percentage of matches. In computing the value for the match columns, the SWIFI method was considered to give the “gold” result. If, for a given injection, the simulation result agreed with the SWIFI result, a match was said to occur. The value in the match column is the number of matches that fell into a given category divided by the total number of SWIFI results in that category. For example, the simulator and SWIFI results matched for 51 injections in the *message dropped* category. The maximum possible number of matches for this case would be 54 (100%). Thus, for the *message dropped* category the simulator accuracy was 94.4% (51/54); for the remaining seven faults in which the simulator detected a drop, the SWIFI determined a different error category.

Table 8.1: Number of Errors by Category for Simulation and SWIFI in First Run

Fault-injection result	SWIFI	Simulation	Matches	% Match
MCP hang	52	14	10	19.2%
Hang remote MCP	4	0	0	0.0%
MCP restart	15	14	5	33.3%
Message dropped	54	58	51	94.4%
Data corrupt	19	29	15	78.9%
No error	241	270	239	99.2%
Total	385	385	320	83.1%

Table 8.1 shows that the simulation does extremely well at detecting when no error will occur (matching SWIFI for over 99% of the faults) and reasonably well at predicting the less severe injection results (e.g., the simulation correctly identified a dropped message for about 95% of the faults for which SWIFI determined that result). The simulation, however, has relatively low accuracy in predicting severe fault results, such as host interface hang where about 20% of the injections match. One reason for the low level of accuracy is that the simulation does not fully model the interaction between the host and the interface.

### 8.2.2 Validation of revised MCP

After making some improvements to the simulation (such as fixing problems that caused the simulation to crash and improving the accuracy of simulating functions that modified pointers in the high-level simulation), the experiment was repeated. This time, only 4 faults needed to be discarded because they caused the simulator to crash; and 73 faults were discarded because the effect of the fault varied in the SWIFI experiment.

The results of this trial are given in Table 8.2 using the same format used in Table 8.1. Since this trial included some of the faults that were discarded in the first trial, some of the values in the SWIFI column have changed.

Table 8.2: Number of Errors by Fault Category for Simulation and SWIFI in Second Run

Fault-injection result	SWIFI	Simulation	Matches	% Matches
MCP hang	78	61	51	78.2%
Hang remote MCP	4	0	0	0.0%
MCP restart	16	6	6	37.5%
Message dropped	55	58	52	94.5%
Data corrupt	19	19	16	84.2%
No error	251	279	245	97.6%
Total	423	423	370	87.5%

### 8.2.3 Comparison with modified MCP

After observing the cause for some of the errors, a few error recovery or avoidance mechanisms were added to the MCP. The first mechanism added a check to the MCP that discarded incoming messages with the length field in its header set to zero. (This bug was determined to be the cause of the errors that hung the remote MCP.) The second mechanism recovered from errors where the MCP hung. Several of the hangs were caused when the MCP erroneously considered the send buffer to be full (when it was, in fact, empty). In this case, the MCP would hang waiting for room to be added in the send buffer. This was fixed by limiting how long the MCP waits for a buffer to become available. The third mechanism also limited how long the MCP waits for the state machine to change state. This timeout prevents the MCP from waiting indefinitely

for the `DMA_BUSY` signal to be reset when the DMA is not in use (but due to a fault the MCP erroneously asserts the `DMA_BUSY` signal).

After the MCP code was changed to add the three types of recovery mentioned above, both the SWIFI and the simulated fault injectors reran the experiment. The results of the experiments are shown in Table 8.3. As before, the number of faults causing a given impact are listed for each of the five error categories for the simulation and real system. This time, however, there are two numbers separated by a slash. The first number is the impact of the fault in the previous trial, the second number is the impact of the fault with the new recovery routines added.

Table 8.3: Comparison of Errors without/with Fault Recovery and Avoidance Code

Fault-injection result	SWIFI	Simulation
MCP hang	78/32	61/36
Hang remote MCP	4/0	0/0
MCP restart	16/16	6/6
Message dropped	55/67	58/63
Data corrupt	19/19	19/19
No error	251/289	279/299
Total	423	423

Experiments were run on the simulated system first to evaluate the effectiveness of the recovery mechanisms before implementing them on the real system. The results of the simulated fault injections showed a decrease from 61 to 36, or 41%, in the number of faults that caused hangs. Note that the added recovery was completely outside the fault-injection region so that the injected faults could be made exactly the same as in the

verification study. Of those same faults, 25 no longer caused a hang after the addition of the recovery code.

### 8.3 Discussion

Several factors affecting the accuracy level of this study are addressed in [3]. A summary of those factors is given below.

Three limitations of the cycle-accurate simulation should be noted. First, the simulator does not model memory-mapped I/O. As a result, only some of the MCP can be simulated using the simulated fault-injection approach. A second limitation is that certain error conditions are not produced by the simulator. One particular difference we noted is that the simulator did not implement memory protection of the low-memory segment, where the MCP is stored, as can be done in the real host interface. A final limitation is that the simulation runs several orders of magnitude slower than the real system.

Further, the simulation is only as accurate as the technical specification of the system. One example of unspecified behavior is the response to an unaligned memory access: a 32-bit memory access has to be aligned to a 32-bit boundary (the memory address must be evenly divisible by 4) in the LANai device. Another example of this specification problem involves a particular memory-mapped register. In this example, a fault changed the DMA\_DIR register, which holds a one-bit value specifying the direction of DMA transfers between the host (workstation) and LANai chip. In our simulations, we considered only

the lowest bit of this register to be valid, so writing a 5 or a 1 to this register would give the same result. Experiments on the real device showed, however, that values other than 0 or 1 written to this register can cause the interface to hang. The SWIFI fault injector verified that this is the true behavior of the hardware.

## 9. DISCUSSION

This study prompts three areas of discussion: (1) the relatively large number of host interface failures as compared to other failure types, (2) the dependability issues that arose from the experiments, and (3) the factors that affect the experiments.

### 9.1 Host Interface Failure

By far the most common error found is some form of host interface hang. Several errors can cause the MCP to hang. For example, disrupting the state machine in the MCP so that it waits for events that cannot occur causes a deadlock condition. In another example, writing zero into the stack pointer appears to cause some error that prevents the MCP from correctly initializing itself in some cases.

The MCP can hang in several ways. In one situation, the MCP continually resets itself and continually reports to the device driver that it has been reset. Some corrupt instructions cause the MCP to use a dead register value to update a variable, thus affecting different parts of the program. The MCP's state machine can be affected such

that it is partially hung (i.e., it either continues to send messages but no longer accepts incoming messages or vice versa). Partial hangs can make it difficult to implement a watchdog because they are not fail-silent.

The following is an example of a fault that causes a remote interface to hang. The fault changes the content of the memory location `0x4960` from `0x87320090` to `0x87322090`. Mnemonically, the instruction changed from `LOAD 0090[%r12], %r14` to `LOAD 2090[%r12], %r14`. This new `LOAD` instruction first adds `0x2090` to the contents of register 12 to determine a memory address and then copies the contents of memory at that address to register 14. At that point in the code, register 14 is used to hold the length of the message being sent. Note that this length value is eventually written into the outgoing message's header. With the fault, the instruction reads the length value from an incorrect address. Typically, the erroneous value read is zero. The message is then sent over the network with the incorrect length value in the message header. When the remote node receives the apparently valid message with the message-length field set to 0 in the header, it hangs. This behavior was verified by having the MCP produce messages with 0 in the header length field.

## 9.2 Myrinet Dependability Issues

This experiment underscores three major issues in Myrinet dependability. First, our results indicate that the probability of host interface failures as a result of instruction or data corruptions due to faults can be very high. We should point out that the section of

code we injected faults into is only a small piece of the MCP and that other sections of code are likely to respond differently to faults. Second, the host computer can hang as a result of faults injected into the Myrinet. Third, the existence of faults that propagate from one node to another is an important issue.

With a relatively high number of faults resulting in hangs, our results suggest that Myrinet may not be suitable for critical applications. It should be noted, however, that the fault injections caused an application to terminate only when the host computer failed. This suggests that the Myrinet may support a level of dependability sufficient for most applications.

Another issue is raised by those faults that caused the host computer to hang. Although the root cause of these hangs is still unknown, we were able to reproduce this effect for at least some of the faults when injected into the PC. In these cases, it appears that the fault causes the interface to write to illegal addresses in the host computer memory, perhaps overwriting the kernel. We have verified that the host interface is capable of overwriting physical address 0 on both platforms (SPARC and PC) and this action results in hanging each platform. In other words, the host is not protected from its external devices. If this is the case, it is not only a dependability issue, but also a security problem. It is possible that adding hardware to protect certain memory ranges would prevent faults from causing this error. Choosing a platform that does not allow external devices on its I/O bus to overwrite its own kernel might also fix the problem. The reason

that the fault propagates from the Myrinet to the host computer may be a design flaw in the computer.

Another issue is the propagation of a fault from one node to another. In 30 faults, we saw a fault injected in one node hanging another node. By investigating some of these faults, we discovered that when a node sends a valid message with the message length field in the header set to zero, the receiving node hangs while processing the message. We suspect that all of the errors which propagated from one interface to another result from this behavior. If so, the problem can be resolved by modifying the MCP to check for zero-length data messages and to drop any such message as an illegal message. Regardless, such propagation of faults in a network is an issue that needs to be considered in network dependability studies.

### 9.3 Effects of Fault-Injection Parameters

Several of the experiments in this work examined the parameters affecting the fault injection environment. These parameters include the workload level, the platform of the target host, and the target region of memory where faults are injected.

The parameter that had the greatest effect on the results of the experiment is the target region of memory where faults were injected. Some regions, such as the host table data structure, produced very few errors (only 2 of 1125 faults lead to errors for one version of the MCP). Another region of data structures, the net receive data structures, had a fault-activation level of 9.2%. Injecting faults into the instruction memory usually

had a higher fault-activation level. For example, the `hostSend()` function had a high fault-activation level (42% of 2080 faults caused some error). Despite the difference in the fault-activation rate, the sets of errors produced from each data region are very similar.

These results suggest that some regions of data are much more critical than others. Assessing the dependability of systems efficiently requires injecting faults into these critical regions. One approach to finding critical regions is random fault injection. Random fault injection is bound to eventually find the critical regions. Once critical regions are found, injections can focus on those regions. Another approach to finding critical regions is to examine the code and select those regions that seem to be important in carrying out the purpose of the code. The later approach was used in this study.

The SWIFI experiments also examined the effect of workload and host platform on the manifestation of errors. The workload level had very little bearing on the effect of the faults. The number of faults resulting in the *MCP hang sometimes* category (where the injected fault caused the MCP to hang about half the time) was much higher with the heavy workload (10 compared to 1). The number of faults in each of the other error categories was similar between the two runs. The number of faults causing errors was only slightly higher for the high workload (72 of 800 faults caused an error in the heavy workload compared with 65 of 800 in the light workload). These results suggest that different workload levels should be considered when assessing the dependability of systems.

The choice of host platform had little bearing on the effect of common errors such as dropped messages. This is not surprising since there is no interaction between the interface card and the host computer for these faults and the MCP is the same for each platform. But there was a difference in the results of faults that propagated to the system bus, particularly for catastrophic errors. From the experiment, twice as many faults caused a host computer to crash on the PC than on the SPARC Ultra. Differences in the I/O bus (the Ultra uses an S-BUS and the PC uses a PCI bus) and in the host kernel account for these differences.

## 10. CONCLUSION

### 10.1 Summary

This thesis provides an in-depth dependability study of a Myrinet LAN. The study includes the creation of a SWIFI fault-injection environment and performance of several experiments on the Myrinet testbed using that environment. The experiments looked at the application-level impact of the faults in terms such as dropped messages, interface hangs, or host system hangs. The fault model was a transient single-bit flip in either the instruction executing on the MCP or a single-bit flip anywhere in memory (on the Myrinet interface card).

By running experiments under different conditions, such as workload intensity level, the study examined the effects of these parameters. The results show that the target region of memory has a larger impact on the fault-activation level than any other parameter tested. Also, the host platform had an effect on those faults causing critical errors, such as crashing the host computer.

The largest dependability issue for the Myrinet LAN that was uncovered was that the host computers have no memory protection from the interface cards. Some faults caused the LANai processor to write incorrect memory addresses in the host computer's main memory; the DMA hardware lacks circuitry to protect the host computer from this type of error. In some cases, a single-bit error lead to this sort of error causing the host computer to crash.

The fault-injection environment was also used to validate a simulation of the fault injection experiments. The results of the validation show that the simulation is useful for determining the effect of many common faults (such as dropped messages). But the results also expose limitations in the simulated fault injected that affect more critical errors.

## 10.2 Future Work

Some limitation of the SWIFI fault injector should be noted. First, as with any fault injection study, the fault injector is only as useful as the fault model. Here, the fault model of a transient single-bit flip may be unrealistic for the study. If the memory chip is encoded using ECC logic, then all single-bit flips in memory would be corrected. The Myrinet does not, at this point, incorporate any such memory protection. Also, the fault model of flipping a single bit of the executing instruction may be biased. This fault model was useful since (1) it represents a fault that could occur and (2) it has the potential

for affecting the register file, memory, or the results of the ALU. Certainly, however, the instruction set architecture has a large impact on the set of possible errors.

Other fault-injection experiments might produce informative results. Simple random fault-injection experiments might show a very low fault activation-rate and a distribution of errors similar to that of the other experiments. Another example would be to inject multiple faults into different nodes of the system at the same time. The fault injector has been used to do this into the RAIN testbed to show that it can be done. Total failures seemed to increase, but there is no reason to believe that any of the faults were correlated.

## APPENDIX A. FAULT LOG EXAMPLE

Figure A.1 shows a sample of a log file entry for one fault (fault number 0128). The first three lines give information about the fault: the fault number, the address of the MCP instruction in which the fault is being injected, and the corrupt instruction. The log then shows the results from each of the ten times that the fault was injected.

The next six lines show the first injection. The first of these lines shows that fault is being injected into *mahler's* interface at time 21:44:21. The next line shows that the injection completed successfully. In the next line, an error message from *dusek*, the node receiving messages from *mahler*, states that while waiting to receive message 8245, message 8241 was received. This error message is interpreted as a “message resent” error. The next message (message 8246) then arrives, but since message 8245 never arrived, another error message from *dusek* states that a message was dropped. That error message is interpreted as a “message dropped” error. For this injection, what really happened is that the contents of message 8245 were replaced with the contents of message

Inj. #	FAULT: fault.0128 FAULT ADDRESS: 0x48bc Fault changes instruction from 0xF6863DB0 to 0xF6863DB4
1.	mahler 21:44:21 Fault injection number 1 mahler 21:44:21 Fault injection complete dusek 21:44:21 message_id=8241 (expects 8245) ERROR: message resent dusek 21:44:21 message id=8246 (expects 8245) ERROR: message dropped
2.	mahler 21:44:23 Fault injection number 2 mahler 21:44:23 Fault injection complete dusek 21:44:23 message_id=10708 (expects 10712) ERROR: message resent dusek 21:44:23 message_id=10713 (expects 10712) ERROR: message dropped
3.	(Injection number 3 resulted in corrupt message)
4.	mahler 21:44:29 Fault injection number 4 mahler 21:44:29 Fault injection complete dusek 21:44:30 time out waiting for message dusek 21:44:31 time out waiting for message dusek 21:44:31 time out waiting for message ERROR: MCP HANG detected
5.	(Injection number 5 resulted in corrupt message)
6.	mahler 21:44:40 Fault injection number 6 mahler 21:44:40 Fault injection complete dusek 21:44:41 time out waiting for message dusek 21:44:42 time out waiting for message dusek 21:44:43 time out waiting for message ERROR: MCP HANG detected
7.	(Injection number 7 resulted in corrupt message)
8.	(Injection number 8 resulted in corrupt message)
9.	(Injection number 9 resulted in MCP hang)
10.	(Injection number 10 resulted in corrupt message)

Figure A.1: Sample Error Log

8241. Hence, the result of this injection is a corrupt message. Seven of the log entries were similar to this one. The figure shows only the first two in detail.

In the fourth injection, *dusek* stops receiving messages from *mahler* and reports that it timed out waiting for new messages. After receiving three such messages, the monitor declares an MCP hang and resets the interface on *mahler*. After the reset, *dusek* begins to receive messages again. Note also that there are no timeout errors from *mahler*. This indicates that either the interface is only partially hung (it can receive messages but not send messages) or the fault propagated to the receiving node (*dusek*). The first of these is in fact the case, since resetting *mahler* (and not *dusek*) remedies the problem. The log entries for injections 3, 5, and 9 were similar to this one; only the first two are shown in detail in the figure.

## APPENDIX B. ILLUSTRATIVE SAMPLE FAULTS AND ERRORS

This appendix includes detailed explanations of some specific faults that were used in the experiment and that produced different errors. These results were determined using the experiments described in Chapter 6.

### B.1 Data Corrupt

In the “receive code,” one fault changed `ld M[127fc], %r14` to `ld M[127cc], %r14`. The instruction should load the contents of memory address `0x127fc` to register 14. The data structure `HostReceiveBuffer` is stored in memory location `0x127fc`. This is supposed to be a pointer to the beginning of an incoming message. Changing the address of the pointer causes the data to appear corrupted. In particular, the application receiving the message finds that the beginning of the message does not match the expected header.

## B.2 Host Computer Hang, Example 1

In the “receive code,” one fault changed `ld M[16b30], %r11` to `ld M[17b30], %r11`. The contents of memory location `0x16b30` is a pointer to the address in the DMA (using the host computer’s physical address space) where the incoming message is supposed to be written. The contents of the new memory location `0x17b30` is apparently to be unallocated and set to 0. The instruction following the faulty one stores the value of register 11 into `0xffffffff3c` (in the LANai’s memory space, the upper bits are ignored), which is the memory-mapped register EAR containing the base address in the host computer’s physical address space of the destination of the memory transfer. The DMA engine on the interface will then write to the memory location with physical address equal to EAR. Writing to this memory location resulted in a system crash on both the Ultra and the PC. The error can be recreated by writing 0 into the EAR and performing a DMA write.

## B.3 Host Computer Hang, Example 2

In the “send code,” one fault changed `st [%r12], %r9` to `st [%r12+152], %r9; %r12←%r12+152`. Register 12 is supposed to hold the HostSendItem structure. In subsequent instructions, the value of register 12 is used to find the buffer holding the data that is about to be sent. The DMA engine attempts to read the physical address given by the faulty address. This causes a system failure on both platforms.

## B.4 MCP Hang

In the “receive code,” one fault changed `bt 22c4` to `bt 2244`, where `bt` is the mnemonic for “branch true” (or unconditional branch). The address of this instruction is `0x22b4`. Thus, this fault changes the instruction from a forward branch to a backwards branch. In this case, the fault creates a loop of 16 instructions in the program’s control flow. Since the fault injector is waiting for the MCP to leave the target segment of code before reverting the faulty instruction, the program is caught in a loop. Hence, the fault becomes a permanent fault, and the MCP is stuck in the loop. This is considered an *MCP hang* by the categories above, since the MCP appears to the API to have hung.

## B.5 Multiple Manifestations

In the “send code,” one fault changed `ld [%r14+24], %r14` to `ld [%r14+88], %r14` (the offsets for this example are given by decimal values instead of hexadecimal). This instruction is called inside the `NetSendQueue.putPeek()` call. The value of `24 + register 14` should be `*buffer[putNext]`. Instead, the faulty instruction gets the buffer address from beyond the bounds of the buffer array. The result of such an operation depends on which buffer in the circular queue is being used. This fault could result in MCP hangs, MCP resets, or dropped or corrupted messages.

## B.6 Message Dropped, Example 1

In the “receive code,” one fault changed `sub.f %r14, 0, %r0` to `sub.f %r14, 128, %r0`. The original instruction subtracts (compares) `r14 - 0`, setting condition flags, and ignores the results (`r0` is specified as the destination, but it is hardwired to 0). At this point in the code, register 14 holds the `HostReceiveChannel->receiveAck.full()`, which is true only if the receive queue is full (which should not happen in the lab setup since the program continually empties the receive queue). The next instruction (branch not equal, `bne 2170`) after the faulty one branches if the `ZERO` is clear over a segment of code that handles a full receive queue. The statement should return 0 since the queue rarely fills in the experimental setup. Thus, the compare statement should set the `ZERO` (or `EQUAL`) condition flag. Then, the branch statement should skip past the code to handle a full receive queue. Instead, the faulted compare statement becomes `0 - 128`, which will not set the `ZERO` flag, and the branch is not executed. The MCP then assumes the receive queue is full and drops the incoming message. This fault produced a dropped message in the real system.

## B.7 Message Dropped, Example 2

In the “send code,” one fault changed `ld M[%r12 + 140], %r14` to `ld M[%r12+140], %r30`. Register 30 is not used in supervisor mode (which we use exclusively), so we are not concerned about the contents of register 30. Here, register 12 holds the `HostSendItem` structure, and `HostSendItem+0x140` is the channel number of the current message as set

by the API. The value of register 14 is supposed to have been used by the MCP to set the message channel. Since register 14 is not set with the faulty instruction, the MCP either drops the message or sends it to the wrong channel, where it is dropped. The SWIFI detected a dropped message for this fault.

## REFERENCES

- [1] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W.-K. Su, "Myrinet: A gigabit-per-second local-area network," *IEEE Micro*, vol. 15, no. 1, pp. 29–36, 1995.
- [2] D. T. Stott, M.-C. Hsueh, G. Ries, and R. K. Iyer, "Dependability analysis of a commercial high-speed network," in *Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS-27)*, June 1997.
- [3] D. T. Stott, G. Ries, M.-C. Hsueh, and R. K. Iyer, "Dependability analysis of a high speed network using software implemented fault injection and simulated fault injection," *IEEE Transactions on Computers Special Issue on Dependable Computing*, vol. 47, pp. 109–119, January 1998.
- [4] R. K. Iyer and D. Tang, "Experimental analysis of computer system dependability," in *Fault-Tolerant Computer System Design* (D. K. Pradhan, ed.), ch. 5, Prentice Hall PTR, 1996.
- [5] J. A. Abraham, "Challenges in fault detection," in *Proceedings of the 25th International Symposium on Fault-Tolerant Computing (FTCS-25)*, pp. 96–114, June 1995.
- [6] W. Kao and R. K. Iyer, "DEFINE: A distributed fault injection and monitoring environment," *IEEE Workshop on Fault-Tolerant Parallel and Distributed System*, June 1994.
- [7] K. K. Goswami and R. K. Iyer, "Simulation of software behavior under hardware faults," in *Proceedings of the 23th International Symposium on Fault-Tolerant Computing (FTCS-23)*, pp. 218–277, June 1993.
- [8] G. Ries and R. K. Iyer, "Evaluating the impact of transient faults on software behavior: Case study of a commercial high-speed network," in *Proceedings of the 6th IFIP International Working Conference of Dependable Computers for Critical Applications (DCCA-6)*, March 1997.

- [9] A. Dupuy, J. Schwartz, Y. Yemini, and D. Bacon, "NEST: A network simulation and prototyping testbed," *Communications of the AMC*, vol. 33, no. 10, pp. 64–74, 1990.
- [10] J. H. Barton, E. W. Czeck, Z. Z. Segall, and D. P. Siewiorek, "Fault injection experiments using FIAT," *IEEE Transactions on Computers*, vol. 39, pp. 575–582, April 1990.
- [11] G. Kanawati, N. Kanawati, and J. Abraham, "FERRARI: A tool for the validation of system dependability properties," in *Proceedings 22nd International Symposium on Fault-Tolerant Computing*, pp. 336–344, July 1992.
- [12] M. Z. Rela, H. Madeira, and J. G. Silva, "Experimental evaluation of the fail-silent behavior in program with consistency checks," in *Proceedings of the 26th International Symposium on Fault-Tolerant Computing (FTCS-26)*, pp. 394–403, July 1996.
- [13] S. Dawson, F. Jahanian, T. Mitton, and T.-L. Tung, "Testing of fault-tolerant and real-time distributed systems via protocol fault injection," in *Proceedings of the 26th International Symposium on Fault-Tolerant Computing (FTCS-26)*, pp. 404–414, June 1996.
- [14] S. Han, K. G. Shin, and H. A. Rosenberg, "DOCTOR: An integrated sOftware fault injeCTiOn enviRonment for distributed real-time systems," in *IEEE International Computer Performance and Dependability Symposium (IPDS'95)*, pp. 204–213, March 1995.
- [15] J. Carreira, H. Madeira, and J. G. Silva, "Assessing the effect of communication faults on parallel applications," in *IEEE International Computer Performance and Dependability Symposium (IPDS'95)*, pp. 214–223, March 1995.
- [16] 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 Engineering*, vol. 16, no. 2, pp. 166–182, 1990.
- [17] 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 Transactions of Reliability*, vol. 39, no. 4, pp. 455–467, 1990.
- [18] S. Han and K. G. Shin, "Experimental evaluation of failure-detection schemes in real-time communication networks," in *Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS-27)*, June 1997.

- [19] E. Fuchs, "Validating the fail-silence of the MARS architecture," in *Proceedings of the 6th IFIP International Working Conference on Dependable Computing for Critical Applications (DCCA-6)*, March 1997.
- [20] X. Castillo and D. P. Siewiorek, "Workload, performance and reliability of digital computing systems," in *Proceedings of the the 11th Annual IEEE International Symposium on Fault-Tolerant Computing (FTCS-11)*, pp. 84–89, July 1981.
- [21] R. K. Iyer, D. J. Rossetti, and M.-C. Hsueh, "Measurement and modeling of computer reliability as affected by system activity," *ACM Transactions on Computer Systems*, vol. 4, pp. 214–237, August 1988.
- [22] L. Xu and J. Bruck, "X-code: Mds array codes with optimal encoding," tech. rep., California Institute of Technology, 1997.
- [23] G. Ries, *Hierarchical Simulation to Assess Hardware and Software Dependability*. Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1997.
- [24] K. K. Goswami and R. K. Iyer, "DEPEND: A design environment for prediction and evaluation of system dependability," in *9th Digital Avionics Systems Conference Proceedings*, pp. 87–92, October 1990.
- [25] K. K. Goswami, R. K. Iyer, and L. Young, "DEPEND: A simulation-based environment for system level dependability analysis," *IEEE Trans. on Computers*, vol. 46, no. 1, pp. 60–74, 1997.