https://ntrs.nasa.gov/search.jsp?R=19970034703 2020-06-16T01:42:36+00:00Z

NASA/CR-97- 206188

NAG-1-613

October 1997

UILU-ENG-97-2230
CRHC-97-17

University of Illinois at Urbana-Champaign

IN-60-CR
067231

Hierarchical Simulation to Assess Hardware and Software Dependability

Gregory Lawrence Ries

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

REPORT DOCUMENTATION PAGE

Form Approved
OMB NO. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comment regarding this burden estimates or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

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

4. TITLE AND SUBTITLE
Hierarchical Simulation to Assess Hardware and Software Dependability

5. FUNDING NUMBERS
NASA NAG-1-613
DABT63-94-C-0045

6. AUTHOR(S)
Gregory Lawrence Ries

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(ES)
Coordinated Science Laboratory
University of Illinois
1308 W. Main St.
Urbana, IL 61801

8. PERFORMING ORGANIZATION REPORT NUMBER
UILU-ENG-97-2230 (CRHC-97-17)

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
NASA Langley Research Center
Hampton, VA 23681

DARPA/ITO
3701 N. Fairfax Dr.
Arlington, VA 22203-1714

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy or decision, unless so designated by other documentation.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12 b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This thesis presents a method for conducting hierarchical simulations to assess system hardware and software dependability. The method is intended to model embedded microprocessor systems. A key contribution of the thesis is the idea of using fault dictionaries to propagate fault effects upward from the level of abstraction where a fault model is assumed to the system level where the ultimate impact of the fault is observed. A second important contribution is the analysis of the software behavior under faults as well as the hardware behavior. The simulation method is demonstrated and validated in four case studies analyzing Myrinet, a commercial, high-speed networking system. One key result from the case studies shows that the simulation method predicts the same fault impact 87.5% of the time as is obtained by similar fault injections into a real Myrinet system. Reasons for the remaining discrepancy are examined in the thesis. A second key result shows the reduction in the number of simulations needed due to the fault dictionary method. In one case study, 500 faults were injected at the chip level, but only 255 propagated to the system level. Of these 255 faults, 110 shared identical fault dictionary entries at the system level and so did not need to be resimulated. The necessary number of system-level simulations was therefore reduced from 500 to 145. Finally, the case studies show how the simulation method can be used to improve the dependability of the target system. The simulation analysis was used to add recovery to the target software for the most common fault propagation mechanisms that would cause the software to hang. After the modification, the number of hangs was reduced by 60% for fault injections into the real system.

14. SUBJECT TERMS
fault dictionary, hierarchical simulation, hardware dependability, software dependability

15. NUMBER IF PAGES
78

16. PRICE CODE

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

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

HIERARCHICAL SIMULATION TO ASSESS HARDWARE
AND SOFTWARE DEPENDABILITY

BY

GREGORY LAWRENCE RIES

B.S., Case Western Reserve University, 1992
M.S., University of Illinois, 1995

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1997

Urbana, Illinois

HIERARCHICAL SIMULATION TO ASSESS HARDWARE AND SOFTWARE DEPENDABILITY

Gregory Lawrence Ries, Ph.D.
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, 1997
Ravishankar K. Iyer, Advisor

This thesis presents a method for conducting hierarchical simulations to assess system hardware and software dependability. The method is intended to model embedded microprocessor systems. A key contribution of the thesis is the idea of using fault dictionaries to propagate fault effects upward from the level of abstraction where a fault model is assumed to the system level where the ultimate impact of the fault is observed, and a second important contribution is the analysis of the software behavior under faults as well as the hardware behavior.

The simulation method is demonstrated and validated in four case studies that analyze a commercial, high-speed networking system called Myrinet. One key result from the case studies shows that the simulation method predicts the same fault impact 87.5% of the time, as is obtained by similar fault injections into a real Myrinet system. Reasons for the remaining discrepancy are examined in the thesis. A second key result shows the reduction in the number of simulations needed due to the fault dictionary method. In one case study, 500 faults were injected at the chip level, but only 255 propagated to the system level. Of these 255 faults, 110 shared identical fault dictionary entries at the system level and so did not need to be resimulated. The necessary number of system-level simulations was therefore reduced from 500 to 145. Finally, a third result in the case

studies shows how the simulation method can be used to improve the dependability of the target system. The simulation analysis was used to add recovery to the target software for the most common fault propagation mechanisms that would cause the software to hang. After the modification, the number of hangs was reduced by 60% for fault injections into the real system.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Ravi Iyer, for his guidance throughout this work. I would also like to acknowledge the contributions of Zbigniew Kalbarczyk, Jagdish Patel, and Myeong Lee to the hierarchical fault model case study, and those of David Stott to the validation case study, both of which are presented in this work. I would like to thank my thesis committee, Bill Sanders, Bharghavan Vaduvur, and Sharad Mehrotra, for their input into this work. Finally, I would like to thank my parents for their help and support, especially during the hectic times of my preliminary and final exams.

TABLE OF CONTENTS

Page

1. INTRODUCTION . . . 1
    1.1 Related Work and Motivation . . . 3
    1.2 Contributions . . . 5
    1.3 Additional Background . . . 6

2. MULTILEVEL SIMULATION VIA FAULT MODEL ABSTRACTION . . 9
    2.1 Fault Dictionaries . . . 10
        2.1.1 Problems with the fault-dictionary method . . . 17
    2.2 Trace-Driven Execution . . . 19
    2.3 Special Techniques . . . 20
        2.3.1 Process-interaction simulation . . . 21
        2.3.2 Object encapsulation . . . 21
        2.3.3 Operator overloading . . . 22
        2.3.4 Variable reference mapping . . . 23
        2.3.5 Custom pointer class . . . 23
        2.3.6 Backwards and forwards translation . . . 24

3. BRIEF DESCRIPTION OF MYRINET . . . 25
    3.1 Myrinet Switches . . . 26
    3.2 Myrinet Host Interfaces . . . 27
    3.3 Myrinet Control Program . . . 29

4. CASE STUDIES . . . 31
    4.1 Modeling a Single Host Interface . . . 32
        4.1.1 System model . . . 32
        4.1.2 Fault model . . . 35
        4.1.3 Results . . . 37
    4.2 Modeling an Entire Myrinet LAN With Validation . . . 46

4.2.1 System model 47
4.2.2 Fault model 48
4.2.3 Results 51
4.2.4 Discussion 54
4.3 Inclusion of Recovery to Improve Dependability 58
4.3.1 Recovery added 58
4.3.2 Results 60
4.4 Incorporation of a Device-Level Fault Model 62
4.4.1 Fault dictionary hierarchy 63
4.4.2 Results 66

5. CONCLUSIONS 70
5.1 Summary 70
5.2 Future Work 71

REFERENCES 75

VITA 78

LIST OF TABLES

Table | Page

4.1: A sample set of fault dictionary entries. 38
4.2: Number of errors by category for simulation and SWIFI. 51
4.3: Comparison of errors before and after recovery was added. 60
4.4: Breakdown of number of errors by category. 67
4.5: Breakdown of number of errors by category and instruction type. . . 68

LIST OF FIGURES

Figure | Page

2.1: Example of hierarchy of simulation abstraction levels. 14
2.2: Picture of the trace-based method. 20
3.1: Block diagram of host interface. 28
4.1: Picture of the simulated network. 33
4.2: A block diagram of the MCP showing the "send message" module. . . 36
4.3: Number of faults leading to given number of corrupt words. 41
4.4: Effect on messages sent during fault lifetime. 45
4.5: Target system for fault injection (simulation and SWIFI). 49
4.6: Diagram of fault injection region. 49

1. INTRODUCTION

This thesis presents a hierarchical simulation method to assess hardware and software dependability that uses a detailed, low-level fault model while still producing system-level results. The method is based on the key idea of using fault dictionaries to propagate fault effects from one level of abstraction to another, from lowest to highest, and on techniques to model the system software behavior under faults as well as the hardware. Using these ideas, a fault model may be assumed at a very low level, for instance the transistor level, but the impact of faults may be evaluated at the system level, including modeling the change in the system software behavior due to the fault.

The technique is intended to help close the loop in a design-for-dependability process in computer systems. In such a process, a system design is developed to provide a given level of dependability as well as performance. Ideally, the performance and dependability of the design are evaluated before a prototype is constructed in order to correct deficiencies in the design. Without this early evaluation, costly redesign may be necessary after a prototype has been built. In order to close this design-evaluate-redesign loop, however,

analytical or simulation techniques are required that can predict system dependability in the absence of a prototype. Because the technique presented in this work is based on simulation, it can be applied to dependability evaluation in the design phase.

At this point, the meaning of the term fault dictionary as it is used in this work will be defined. *A fault dictionary details the impact of faults on the behavior of some subset of the target system in terms of the change in that subsystem's behavior due to the fault as seen from outside the subsystem.* As implied by the word dictionary, a fault dictionary contains many entries, and each entry details the impact of one fault (or of a set of faults with identical impacts) on the behavior of the chosen subsystem. The dictionary is used to model the faulty behavior of the subsystem by considering the subsystem as a black box and modifying its fault-free behavior according to the dictionary entry for the desired fault. In this way, the impact of the fault is raised from the detailed level under which it was described in the subcircuit to the more abstract level in which the subcircuit is only a component. Further details and examples of the use of fault dictionaries in this work will be given in Chapter 2.

The remainder of this chapter will discuss related work in this area and provide some motivation for the development of the method presented here. The next chapter discusses the method in more detail. Chapter 3 describes Myrinet, which is a commercial network that is used as a target system in the case studies presented in this work. Chapter 4 discusses those case studies, including demonstrating the use of fault dictionaries, validating the simulation model of the Myrinet, and showing how the analysis provided

by the tool can be used to improve the target system. And, finally, the last chapter presents some conclusions and future work.

1.1 Related Work and Motivation

There are a number of ways to analyze the dependability of a system. A general overview of measurement techniques for dependability evaluation, including simulated fault injection, physical fault injection, and measurement-based analysis, is given in [1]. More detail on physical fault injection can be found in [2], while software techniques for fault injection are described along with the FERRARI project in [3].

Both physical fault injection and software implemented techniques, however, require at least a prototype system to analyze. These methods are therefore not well suited to the design phase of a product, where such a prototype is not available. Instead, simulation or analytical techniques are used in this phase.

There are some studies that concentrate primarily on modeling the systems at a software level. Models at this level are often graph-based, and may use stochastic activity nets (SANs), similar to [4], or are analytic in nature and are used to predict software reliability under software faults based on testing data [5]. The difficulty with these methods is that some high-level fault model must be assumed. For the analytic methods to predict software reliability from testing data, these fault models come from failures observed during the test phase. In other cases, these techniques are used after a prototype

of the system is available, and so fault models can be taken from observation of the system in the field. Such data is not available during the design phase of the system, however.

Both simulation and analytical techniques have their place in dependability analysis. In this study, however, we concentrate on simulation techniques because of their capability to handle high levels of complexity, to employ generalized distributions for fault arrival or other processes, and to express hardware designs and software protocols very naturally. It should be noted, however, that employing simulated fault injection to study dependability, as opposed to analytical techniques, means confronting the problems of repeated experiments, fault coverage, and confidence intervals.

There are several system-level simulation tools that can model the hardware of a system. REACT is an example of one such tool that operates only at high levels of abstraction, modeling CPU's and memories as either good or faulty [6]. ADEPT and MEFISTO are examples of simulation tools that use VHDL to allow analysis from the logic level to the system level [7, 8]. Some of these tools can also model the software executing on the system in an abstract way. None of these tools, however, models the change in the software behavior due to a fault.

Finally, many different algorithms for simulating the impact of faults at the gate level have been developed. These tools are typically designed to determine the fault detection coverage of a set of test vectors. Some of the strategies employed in these algorithms include concurrent fault simulation [9], parallel fault simulation [10], deductive fault simulation [11], or a combination of the three [12]. While these strategies have been

shown to be very effective in accelerating fault simulation at the gate and switch levels, they are not easily applied to system-level simulations, particularly when modeling the detailed behavior of the software as is done in this thesis. However, this thesis does make use of multiple simulation levels, and where gate-level simulations are done, techniques such as concurrent or parallel simulation could be applied to that part of the overall method. An example of combining concurrent fault simulation with a fault dictionary analysis is described in [13].

The difficulty in modeling software under faults is that the software behavior must be tied to the hardware architecture. Very abstract views of the software lack these connections and require the use of abstract fault models. Low-level views of the software, which allow more detailed fault models, are costly to simulate, however. Thus, a tool is needed that can simulate a high-level view of the hardware and software in a system and yet still maintain the connections between that behavior and views of the system at lower abstraction levels. Through these connections, a detailed fault model that changes the state of the hardware can be mapped to a change in the state of the software.

1.2 Contributions

Four key contributions to the state of the art are made in this thesis. The first is the idea of using fault dictionaries to raise fault abstraction levels from the level where a fault model is assumed to the system level where the impact of the fault is observed. The second contribution is a technique to assess the behavior of the system software

under faults as well as assessing the hardware dependability. Third, four case studies are presented to demonstrate the use of the hierarchical simulation method and particularly how the fault dictionary idea is applied to a real system. Finally, the fourth contribution is a validation of the simulation method by comparison to software-implemented fault injections into a real system.

1.3 Additional Background

This work builds upon the previous research in the areas of system-level simulation, multiple simulation levels, and fault dictionaries to provide such a tool that can connect hardware and software behavior in the presence of low-level faults. Some of the related work on these basic methods is presented in the following paragraphs.

The system-level simulation environment this thesis makes use of is called DEPEND [14]. This simulation environment makes use of a process-oriented simulation strategy and the power of C++ to allow a very natural description for computer systems and software. DEPEND has a library of high-level system components from which simulations can be constructed, such as CPUs or disks.

There is support within DEPEND for an abstract model of software behavior represented by a probabilistic control flow graph [15]. Faults are modeled abstractly according to the corruption they cause in memory, but only the good or faulty states of memory words are tracked, not their actual values. Such a flow-graph model is therefore best suited to abstract simulation of highly reliable systems for long periods of time as opposed

to the detailed software behavior that is analyzed in this thesis. This thesis therefore extends DEPEND with a new technique for simulating software behavior in detail, as well as providing the fault dictionary mechanism by which the impact of detailed, low-level faults may be propagated up to the system level for evaluation.

The use of multiple simulation levels has been explored for fault simulation at the logic level. One technique, called mixed-mode simulation, represents a subset of a circuit where faults will be injected at the electrical level while the outputs from that subset are propagated at the gate level [16]. Another more efficient technique dynamically switches the subcircuit where faults are being injected from the logic level to the electrical level for only the period surrounding fault injection [17].

The term *fault dictionary* was first coined in the context of raising fault abstraction levels in a description of a tool called FOCUS [13, 16]. In FOCUS, a fault dictionary was used to reduce the resource requirements of multilevel simulation and to propagate fault effects from transistor-level models of CPU subcircuits up to a gate-level model of the entire CPU. A similar use of fault dictionaries to go between the transistor and gate levels was made in [18].

One of the major contributions of this work is the extension of the idea of fault dictionaries to a generalized mechanism to propagate fault effects between any two neighboring levels of abstraction of a system. In particular, the application of fault dictionaries to modeling software behavior and the representation of fault dictionaries at the system level as a list of corrupted software variables and the corrupted program control flow due

to a fault in a given program module is new to this work. Also defined by this thesis are four simulation levels where the fault dictionary technique can be effectively applied, allowing fault effects to be propagated upwards from the device-physics level to the transistor level. Finally, the structure of the fault dictionary between each of the four levels is also described.

2. MULTILEVEL SIMULATION VIA FAULT MODEL ABSTRACTION

The simulated fault injection method presented here is designed to achieve two key goals. First, it should simulate the system at a high level of abstraction to provide a very fast simulation of the system hardware and software at the same time that it injects detailed faults and obtains the proper response of the system to those faults. This goal is achieved by conducting multiple levels of simulation and propagating fault effects up through each simulation level through the use of fault dictionaries. The second goal is to avoid simulating the system when it is fault free. This goal is met by performing the simulation in a trace-driven manner where uninteresting portions of system execution can be skipped by jumping ahead to a new state taken from a trace of the fault-free system. Both of these aspects of the system will be described in their own sections below.

In addition, a number of techniques are used to allow the real code of the system software to be used in the highest of the simulation levels. Using the real code allows the software behavior to be described in a very natural way and ensures that the simulation takes into account the specific implementation details of the software. It also allows the

software being modeled to go through several versions without painstaking redevelopment of a new model or extensive modification of the old one. However, because the system software may rely on direct communication with special-purpose hardware in the real system (as the software in our case studies does), it is necessary to use a number of techniques to provide a similar environment in the simulation so that this code can be used with minimal modification. These techniques are described in the section following the fault dictionaries and trace-driven execution.

Finally, although the exposition of the fault dictionary method is a major contribution of this thesis, the real test of the method described here is its application to the analysis of a real system. For that reason, another important contribution of this thesis is the presentation of the case studies in chapter four that show the hierarchical simulation method works on a real system and validate it against fault injections done on real hardware.

2.1 Fault Dictionaries

A definition for fault dictionary has already been given in the introduction chapter, but for convenience, that definition is repeated here. *A fault dictionary details the impact of faults on the behavior of some subset of the target system in terms of the change in that subsystem's behavior due to the fault as seen from outside the subsystem.*

The key function of the fault dictionaries is to raise the abstraction level of faults so that their impact at the system-level can be determined. The abstraction level of a

fault is raised by carefully choosing the subsystems for which a fault dictionary will be generated. The boundaries of the subsystem are placed so that information specific to the abstraction level of the subsystem is naturally contained within that subsystem while the outputs of the subsystem can be easily represented at a higher abstraction level.

As an example, consider a stage in a microprocessor pipeline. The stage consists, basically, of some latches at the input that contain the outputs of the previous pipe stage, a block of logic that computes the outputs for the current stage, and latches at the stage outputs to hold those outputs for use in the next stage. This pipeline stage could be used to generate a fault dictionary for use between the transistor and gate levels. Faults would be injected into a transistor-level model of the pipe stage. Within one clock cycle, the faults would either have made a difference in the value of the output latches or have died out. The fault dictionary entry for each fault would store which outputs were different. Because of the latches at the outputs, the timing, voltage, and current details of the pipeline stage logic block are contained within the subsystem. Outside the subsystem, only the latched values are important, and these values can be easily converted to digital logic values and propagated through a gate-level model of the rest of the system (which is fault free).

There is a second advantage to using a fault dictionary. A fault dictionary can also help to reduce the number of simulations that must be done. Many of the faults injected into the transistor-level pipeline stage mentioned above may not propagate to the outputs and cause a latch error. All of these faults will produce no fault dictionary entry and will

proceed no further up the hierarchy of simulations. As well, many faults may produce identical patterns of latch errors and, therefore, identical fault dictionary entries. In this case, it is only necessary to note the number of faults represented by the dictionary entry. Only one simulation will have to be run at the higher simulation levels in the hierarchy to determine the impact of all of the represented faults.

An ideal fault dictionary wouldn't loose information in the transition from one abstraction level to another higher one. In practice, however, there is some information lost. Consider again the pipeline stage mentioned above. While the latch filters out *most* of the timing and voltage information in the transistor-level model, it doesn't filter all of it. In most cases, the latch will end the cycle with a solid one or zero and no information will be lost. In some cases, however, the latch may come to some intermediate value or continue to settle to a one or zero after the latch has closed. In these cases, there is still some information in the voltage or timing of the latch value that will be lost by considering the latch voltage as a digital logic value that stays constant for the entire cycle as is assumed in the gate-level model.

The fault dictionaries are used by conducting multiple independent simulations at the different abstraction levels. One fault dictionary is generated through repeated simulation of the chosen subsystem for many different faults. The dictionary is then used to represent the subsystem's faulty behavior in a simulation at the next higher abstraction level. In the pipeline example, for instance, assume the microprocessor had five pipeline stages. Each stage could be considered as a subsystem, and a fault dictionary could be generated

for each, beginning with transistor-level fault models and ending with gate-level latch errors. These fault dictionaries would then be used in a gate-level simulation of the entire pipeline. When a fault occurred inside one of the pipe stages, the inputs of the pipe stage would be examined and a suitable dictionary entry would be chosen representing the gate-level impact of one fault in that pipe stage. The latch outputs of that pipe stage would be modified according to the dictionary entry, and the gate-level simulation would continue. Thus, the simulations that generate a fault dictionary and the simulations that use it run independently.

In this work, four simulation levels have been defined where the concept of a fault dictionary can be effectively applied to raise the abstraction level of faults. A diagram of the hierarchy of these simulation levels is given in Figure 2.1. In the figure, fault models are represented by the rounded rectangles. The rounded rectangle in the upper left, labeled "Heavy Ion Particle Impact," represents the assumed fault model of a heavy-ion particle striking a transistor junction in a microprocessor. The remaining three rounded rectangles represent fault dictionaries which can be viewed as abstractions of the heavy ion fault model at higher simulation levels. Each of the three fault dictionaries represents a translation of fault effects between two simulation levels as denoted by the arrows. For instance, the "Output Current Surge" dictionary translates between the device-physics-level representation of faults and the transistor-level representation.

There are four basic simulation steps depicted in the figure. In the first step, two simulations, one at the device-physics level and one at the transistor level, are combined

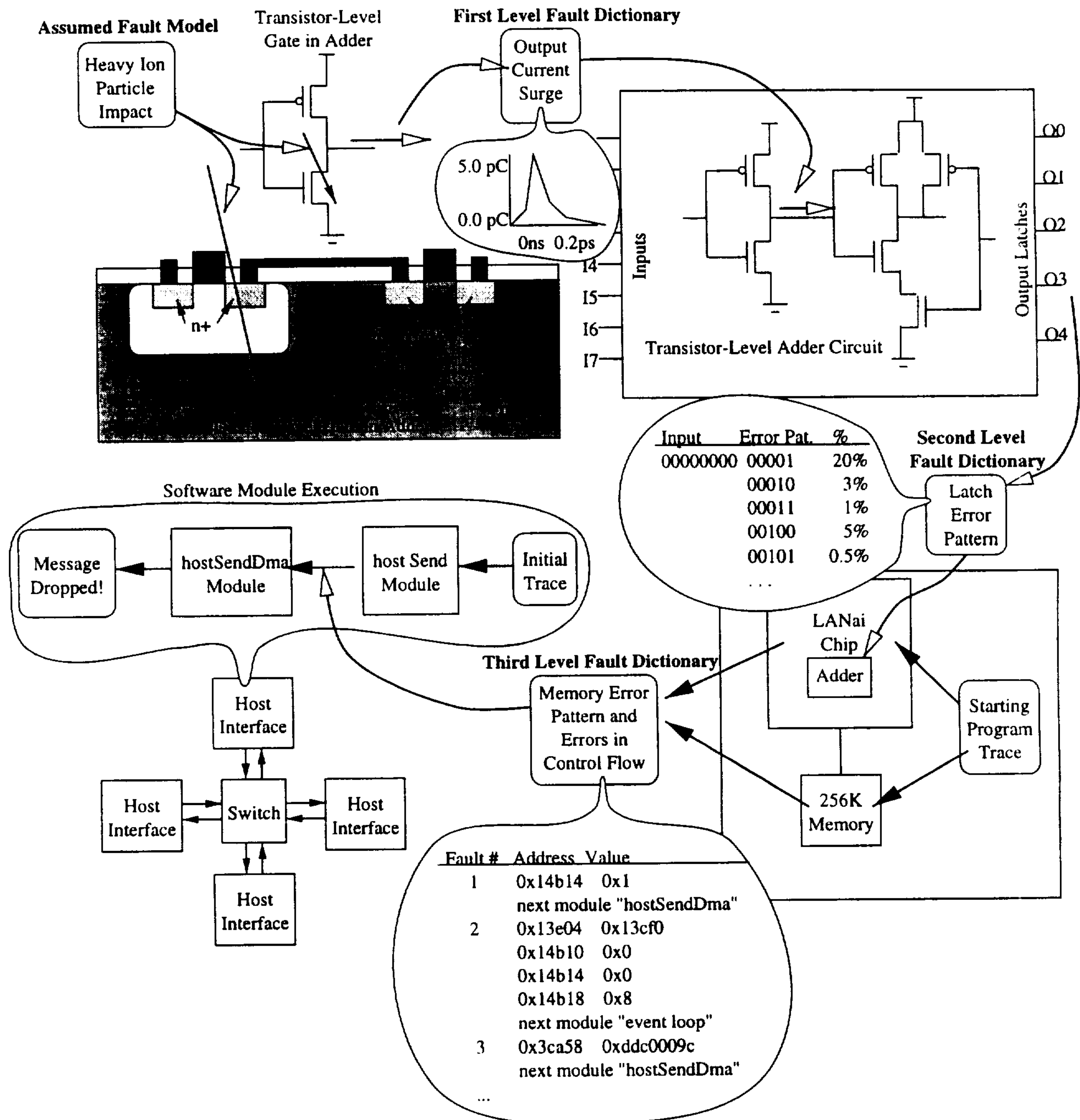

Figure 2.1: Example of hierarchy of simulation abstraction levels.

to determine the appropriate current burst at a gate output due to a heavy ion particle impact. The simulations at this level produce the "Output Current Surge" fault dictionary representing current surges due to ion impacts in different types or sizes of gates. Each dictionary entry models one ion impact as a piecewise-linear analog current waveform recorded at periodic timesteps throughout the fault's lifetime.

The current burst fault dictionary becomes the fault model for the next step, a transistor-level simulation of a subsystem of the processor, perhaps the adder. In this second step, the operation of the adder is simulated for one cycle while the current burst propagates. During this time, the current burst will either die out or affect one or more output latches of the circuit. Any errors on the output latches at the end of the cycle are recorded as the new gate-level fault model for the original particle impact. The resulting fault dictionary, represented by the "Latch Error Pattern" box in the figure, actually represents the impact of faults in the subcircuit probabilistically. For a given input combination, the probability of observing a given latch error pattern will be recorded in the dictionary. Functional decomposition is used to prevent the number of inputs from overwhelming the simulation effort, and any symmetry present in the subcircuit may be used as well.

The latch error fault dictionary is used in the third step where the entire microprocessor is simulated running one module of its software in a cycle-accurate simulation. During the execution of the module, the operation of a microprocessor subcircuit, an adder in the example thus far, is corrupted by modifying its outputs according to the

latch error pattern selected probabilistically from the appropriate dictionary entry. At the end of the module's execution, the memory and processor state are then examined for changes in the software state or control flow. The corrupted memory and control flow are recorded in the third fault dictionary, represented by "Memory Error Pattern and Errors in Control Flow" in the figure. In this fault dictionary, the memory locations corrupted due to the fault are enumerated along with their faulty values, and the next module that will be executed is also listed in case the control flow between software modules was changed by the fault.

The corrupted memory and control flow become the fault model for the particle impact at the system-level. During the simulation of the software execution at the system level, the fault model is injected by modifying the software state and control flow at the end of the appropriate module (the same module simulated when the dictionary was computed in the previous step). The errors are then propagated throughout the software by simulating its execution until the ultimate impact of the fault is known.

While it makes sense to follow the path of a single fault as was done in the description above, the actual simulation proceeds differently. Instead of each simulation level being invoked one at a time, the simulation levels operate concurrently but on different faults. For example, imagine that one simulation has been conducted at the device-physics level, resulting in a current burst fault model. That fault model is stored as the first entry in a fault dictionary between the device-physics-level step and the transistor-level step. Next, the device-physics simulation may pick up a new fault and begin a second simulation.

At the same time, the transistor-level simulation may pick up that current burst in the first fault dictionary entry and begin the corresponding transistor-level simulation.

Note that it is not necessary to always begin with a device-level fault model. If less accuracy is desired, fault injection could begin at levels above the device level, such as a bit-flip fault model at the logic level. If this is the case, however, then the impact of implementation details such as supply voltage levels, accurate circuit timing, and transistor sizing will not be considered in the final results.

2.1.1 Problems with the fault-dictionary method

There is one major problem that must be overcome in applying the fault dictionary method. That problem is the fact that the entries in the fault dictionary depend not only on the fault that is being injected but also on the inputs to the chosen subsystem. For example, the impact of a fault in the instruction decode unit may depend on the instruction being decoded as well as the fault injected.

At the transistor and gate levels, the solution to this problem is to carefully choose the subsystem for which a fault dictionary will be generated so as to keep the number of inputs down to a manageable level. For example, rather than choosing the entire instruction decode stage of a pipeline as a block for a fault dictionary, it may be more manageable to look at the instruction decode stage as being a combination of several circuits. One examines the highest opcode bits to decode the instruction type. Another examines the middle bits to decode source registers. Finally, a third examines the lowest

bits to determine the result register. Symmetry may be used in addition to the functional decomposition to further reduce the number of inputs that must be considered.

At the upper simulation levels, though, the different input states that must be considered are the different states of the software when it enters the chosen module for which a fault dictionary will be developed. In this case, it is not so clear how to reduce the number of software states that will be considered in forming the fault dictionary. A claim is made here that, for the types of systems for which this technique was developed, embedded microprocessor systems, the state of the software upon entering a block is not as important in determining the impact of a fault as is the particular fault itself or the design of the module. In other words, the impact of many faults within the module will be independent of the software state when entering the module or will be dependent on only a couple variables which are major inputs to the module. If such is really the case, then only a sampling of the software states upon entering a module need be considered when developing a fault dictionary for that module. This claim will be substantiated in the second case study.

Finally, note that all fault injection methods, including measurement, must deal with this problem of the impact of different software states during fault injection. The problem is not further exacerbated by the use of fault dictionaries. Thus, similar techniques can be applied with fault dictionaries as are applied elsewhere.

2.2 Trace-Driven Execution

A trace-driven model of the software is used at the system-level to allow the simulation to skip over uninteresting periods of the software's execution (when the system is fault free). The trace consists of a series of snapshots of the software state at different points throughout its workload. In the case studies that are presented later, these snapshots are taken from an actual fault-free run of the software for the network workload that was used.

The trace is used in the following way. When a fault is to be injected, the state of the software is initialized to the latest point that the state is known before the fault occurs. The software code is then simulated from that point up until the fault injection. The fault is injected by corrupting the software state as dictated by a fault dictionary entry for the module of the software which is being executed when the fault occurs. Execution of the program code then continues until the outcome of the fault injection is known and the software can be said to be in a good state. This may mean that the software is simulated until it crashes and is reloaded in a good state, or it may mean the simulation continues until it is feasible to compare the state to that of the good program, and those states are the same. In either case, the next step is to determine the time of the next fault and repeat the process of initializing the program state. A picture of the trace-based simulation method is given in Figure 2.2.

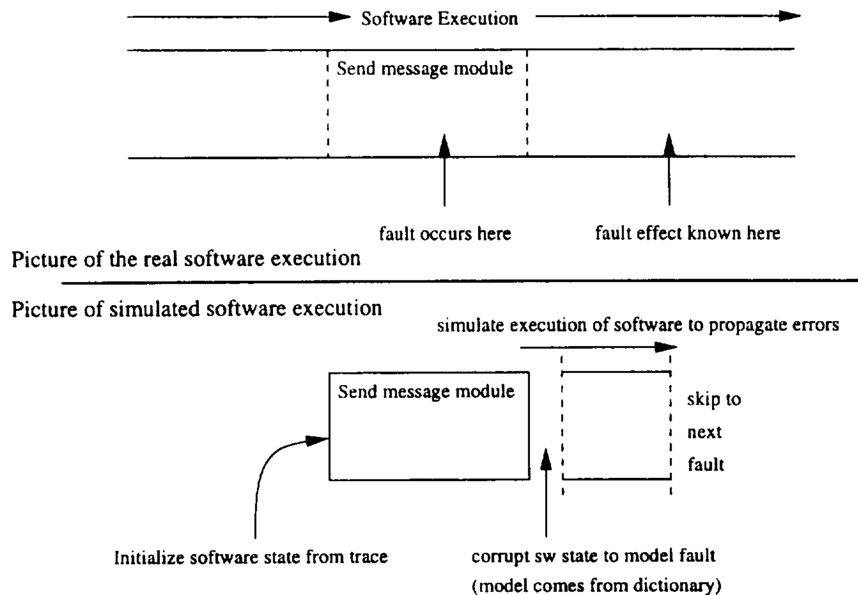

Figure 2.2: Picture of the trace-based method.

2.3 Special Techniques

All of the special techniques that are described in this section are used only in the system-level simulation to allow the real code of the software to be used in modeling the system's software behavior. It is not necessary to use these techniques in any of the lower simulation levels that were described in the fault dictionary section. The first of these techniques is provided by using the DEPEND simulation engine and environment. The middle four techniques are all provided by using an advanced C++ compiler (and thus are only directly applicable to system software written in C or C++, although it may be possible to modify them for software in other languages). Finally, the last technique is is provided outside of the simulation environment to transfer system state between the system-level simulation and the next lower level (behavioral level in the case studies presented later).

2.3.1 Process-interaction simulation

Process-interaction is a strategy for representing a discrete event simulation [19]. In this strategy, the discrete events are grouped together in a sequence according to the process they are part of. For example, a database search could be considered as a process containing a sequence of many discrete events each of which access a database to read some keys. All of the processes in a simulation are executed together, much as a multitasking operating system executes many program tasks together. However, when a process is waiting for the simulation time that its next event should be executed, it is temporarily blocked.

This type of representation allows a very natural representation of computer systems and is one of the core features of the DEPEND simulation engine and environment used in this work. The strategy was key to allowing the use of the real software code in modeling the software behavior of the target system. With the process-interaction strategy, the software behavior could be simply represented as one process which executed the real program code instrumented with occasional time delay statements (representing the passage of time as the code was executed).

2.3.2 Object encapsulation

The process-interaction strategy allowed the easy representation of one copy of the software, but it was not sufficient by itself to allow the representation of multiple copies. This is because the software may make use of global variables which would be incorrectly

shared by all of the copies of the software executing in the simulation. In order to avoid this collision for global variables without extensive modification of the software, the object-oriented nature of C++ was used. A C++ object was created to represent one copy of the software, and all of the global variables used by the software were allocated in that object. In this way, each copy of the software had its own copy of the global variables, and the C++ compiler would automatically ensure that the code associated with one particular object would access the variables also associated with that object.

2.3.3 Operator overloading

After using the above two techniques, it is possible to simulate the execution of multiple copies of the system software. One further problem with using the real code, however, is the fact that the simulation environment may be different than the environment in which the code would normally execute. In the real environment, there may be memory-mapped I/O or other special purpose hardware. In order to provide a similar environment in the simulation, extra simulation processes were created to perform the functions of the missing hardware, and C++ operator overloads were used to direct the C++ compiler to automatically generate the necessary communication between the real code and the new processes that simulate the special hardware, avoiding any necessary modification of the real code.

2.3.4 Variable reference mapping

The remaining techniques are all used to help model the behavior of the software under a fault. Variable reference mapping is used to arrange the software variables for one simulation copy of the system software into a memory map that matches the arrangement for the real system. This arrangement is done by using a feature in C++ called references. A reference associates a new variable name with an already existing variable. This feature was used to create the desired memory map by first allocating a large array to model the system memory. Then, the normal global variables of the program are changed to references which are mapped to the appropriate locations in the already existing array. In this way, a duplicate of the real system's memory map is created in the array modeling the system memory.

2.3.5 Custom pointer class

Another technique that was used to provide more accurate software execution in the presence of a fault and an easier translation for fault models was the creation of a custom class to represent pointers in the simulated system. This custom pointer class acted, as much as possible, the same as a standard C++ pointer with the exception that it always pointed within the memory array allocated for the simulated system. This custom pointer class provided two functions. It helped to prevent the simulated system from overwriting regions of memory in the simulator that were not part of the memory of the simulated system. It also allowed the custom pointers to have the same value as the pointer in the

real system would have. For example, if the array modeling the system memory were allocated at address 500, the value of a normal pointer for the first word of the memory would be 500, not 0 as it would have been in the real system. By using a custom pointer class, the class could take care of dealing with this offset of 500, so the custom pointer value could be stored as 0.

2.3.6 Backwards and forwards translation

The last technique, backwards and forwards translation, is used to translate the software state from the system-level simulation to the next lower level (backwards translation) or from the lower level to the system level (forwards translation). Providing this translation allows a state from the system-level simulation to be used for fault dictionary generation in the next lower level (behavioral level in the case studies). The forward translation then allows the fault model coming out of that fault dictionary to be included back in the system-level simulation. Because of all the techniques described above that are used to make the simulation execution environment and the real execution environment for the system code to be the same, the two translation steps for the case studies were as simple as copying the memory array from one model to the other.

3. BRIEF DESCRIPTION OF MYRINET

All of the case studies in the following chapter that demonstrate the simulation method discussed in this thesis use a Myrinet as the target system. For this reason, a description of the Myrinet hardware and software is provided below. Additional details can be found in the references or at Myricom's web site (http://www.myricom.com) [20].

Myrinet is a commercial, high-speed, local area network. It is based on packet-switched, point-to-point communication technology that was first developed for deployment in system area networks, such as in the Mosaic multicomputer. Information can flow along the links in a Myrinet at 1.2 Gb/s in both directions, and the total peak bandwidth available in a Myrinet scales upward with the number of hosts connected to the network.

A Myrinet is made up of combinations of three key components. The first, a host interface, is an expansion board that connects a host computer to the network. Second, the Myrinet Control Program (MCP) is the control software that runs on each host interface board and performs the network control functions, routing, and message transfer.

The third component, a switch, is used to connect multiple host interfaces (and thus host computers) together for topographies other than two directly connected hosts. Each of these components will be described in further detail in the sections below, beginning with the switches.

3.1 Myrinet Switches

Each of the switches in a Myrinet is a perfect crossbar. At the time of this writing, there exist Myrinet switches with from 4 to 10 bidirectional ports. That means a 10-port Myrinet switch really has 10 input ports and 10 output ports and can form any permutation of connections from input to output with no more than one input connected to each output.

Switches can be connected to other hosts or other switches in an arbitrary, multilevel topography. There may be only one route between each pair of hosts, or redundant routes may be provided by connecting more than the minimally necessary number of switches. Connections may be added or removed from the Myrinet at any time, and the network will automatically adapt to the new configuration.

Information flows through a switch in atomic packets. When a packet enters a switch, the first byte of the packet designates the output port through which the packet should leave. The switch will attempt to make that connection and will hold it until the entire packet travels through the switch. After the packet leaves, the switch tears down that connection and can allow some other input port to access that same output port.

Each switch provides only enough buffering to store information in transit on an input port should the selected output port for that packet be blocked. If such is the case, the switch will send a flow control message back along the reverse channel for the input that is blocked to inform the transmitting host to stop transmission. Once the selected output is no longer blocked, another flow control signal will be sent to the transmitting host to notify it to continue transmission. Messages are therefore buffered in the host interfaces, not in the switches.

3.2 Myrinet Host Interfaces

Anywhere a host computer is to be connected to the Myrinet, a host interface board is inserted to form the connection. The host interface board sits on the host computer's expansion bus (such as a PCI bus) and connects to another host computer or network switch through a multiconductor cable link. A block diagram of the host interface is shown in Figure 3.1.

Each host interface board is an embedded computer system. It contains a custom, 32-bit microprocessor, 256K of static RAM, and additional hardware that connects it to the host computer expansion bus and the network links. The Myrinet Control Program is executed on this custom processor on each interface board, and its binary image is stored in the static RAM. The static RAM is also used to buffer incoming and outgoing messages.

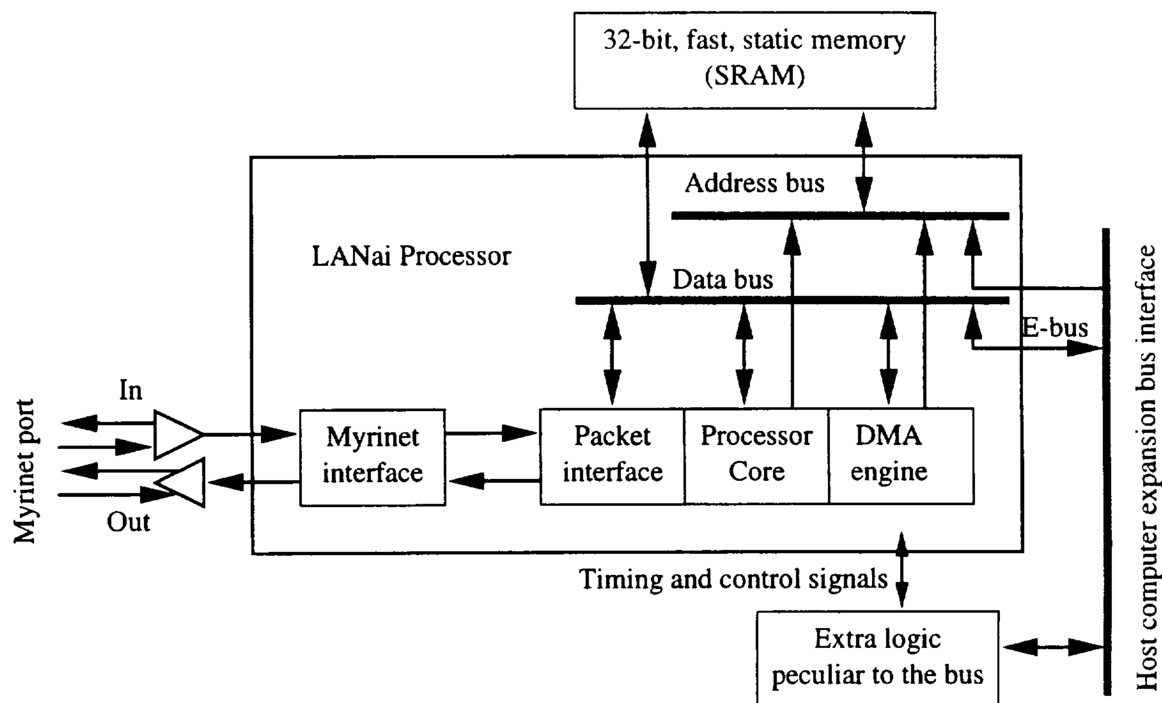

Figure 3.1: Block diagram of host interface.

The custom processor on each interface board is called the LANai processor and was designed by Myricom. It is a RISC processor with multiple general-purpose registers and a load/store instruction set. In addition, the processor makes use of memory-mapped I/O to provide hardware interfaces to the host computer expansion bus, the outgoing network link, and the incoming network link. Each of these interfaces can operate concurrently with the processor, allowing the generation of up to five simultaneous memory accesses (instruction, data, expansion bus, incoming link, outgoing link) of which two can be satisfied by the memory in one cycle. The processor operates at a speed of 40 MHz, and each of the hardware interfaces can transmit one 32-bit word on each cycle.

The memory on the host interface board is accessible by the host computer as one contiguous chunk. This feature is used both to allow the host computer to download the

MCP software to the interface board and to allow the host computer to communicate with the interface by writing signals into the interface's memory.

3.3 Myrinet Control Program

The MCP is the brains of a Myrinet. It performs nearly all of the network control functions–only simple flow control is performed by the switches. In particular, the MCP's functions can be broken down into the following three classes: sending messages out onto the network, receiving messages from the network, and cooperating to map the network and determine proper routes.

The MCP sends a message by programming the memory-mapped I/O interfaces of the LANai processor. When the host informs the interface board that a message is ready to be sent, the MCP will perform the following steps. First, it will allocate a buffer in its static RAM to hold the message. Next, it will construct the header of the message, including such details as the message type, length, and destination. The MCP then initiates a DMA transfer to get the message data from the host computer into its buffer in static RAM through the host's expansion bus. The correct route will then be prepended to the message from the routing table. Finally, the memory-mapped interface for the outgoing network link will be programmed to initiate the transfer of the message onto the network.

A receive operation contains many similar steps to a send, performed in the reverse order. The incoming network link is first programmed to accept a message from the

network into a buffer in the static RAM. Next, the message header and CRC are examined to be sure the message is valid and without error. (Invalid or erroneous messages are dropped.) A message buffer on the host is then allocated for the received message, and the expansion bus memory-mapped interface is programmed to transfer the message from the interface's static RAM to the buffer in the host computer.

The last function of the MCP, cooperating to map the network, is slightly more complicated and will only be summarized here. Each interface has a unique ID, and in mapping, this ID is used to select one interface as the mapper. This interface will initiate network mapping periodically, send mapping requests, collect replies, and distribute completed maps to all other interfaces. The mapping interface forms its map by sending "are you there" type messages exhaustively to the possible ports in the network. Unconnected ports are detected by a lack of response to the inquiry for some timeout period. The other interfaces in the network will only respond to mapping requests and periodically compute routing tables when a completed map is received. In the case that a timeout period occurs with no response from the mapping interface, a new mapper will be chosen from among the remaining MCPs. With this mapping protocol, a Myrinet can dynamically adjust to changes in the network topography.

4. CASE STUDIES

Four case studies were done to demonstrate the simulated fault injection method presented in this thesis. All four of the case studies used a Myrinet as the target system, but each study had a different focus. In the initial case study, the main goal was to demonstrate how the fault injection method could be applied to the Myrinet system. This case study was therefore limited to modeling only a single host interface in detail, and a behavioral fault model was used. The second study built upon the first by extending the system model to an entire Myrinet LAN and doing some verification against a real Myrinet. The third study added some recovery code to the Myrinet MCP software, based on the results of the second study, in an attempt to reduce the tendency of the host interface to hang under fault injection. Finally, in the fourth case study, the focus was on the use of the fault dictionaries to raise the abstraction level of an input fault model from the device level to the system level.

4.1 Modeling a Single Host Interface

This study was done as an initial demonstration of the fault-dictionary-based simulated fault injection method. It was therefore limited to modeling only a single interface in detail and using a behavioral-level fault model to reduce the implementation time of the study. As a result, the special system-level simulation techniques of object encapsulation and custom pointer class were not necessary in this study. This case study was first presented in [21].

In this study, the simulated hardware consists of the LANai processor and 256K of memory, and the simulated software consists of the majority of the MCP program. One function of the MCP, the network mapping, is not simulated in this case study because a full Myrinet is not simulated–a predefined network workload is used rather than simulating the behavior of multiple nodes on the network in addition to the node undergoing fault injection. A pictorial representation of the simulated system is given in Figure 4.1.

4.1.1 System model

There are two simulation levels in this case study. First, a cycle-accurate simulation of the LANai chip and memory is used to do detailed simulation of one module of the MCP for the short period surrounding a fault injection, creating a fault dictionary. Second, a system-level simulation is done, using the real C++ source code of the software and

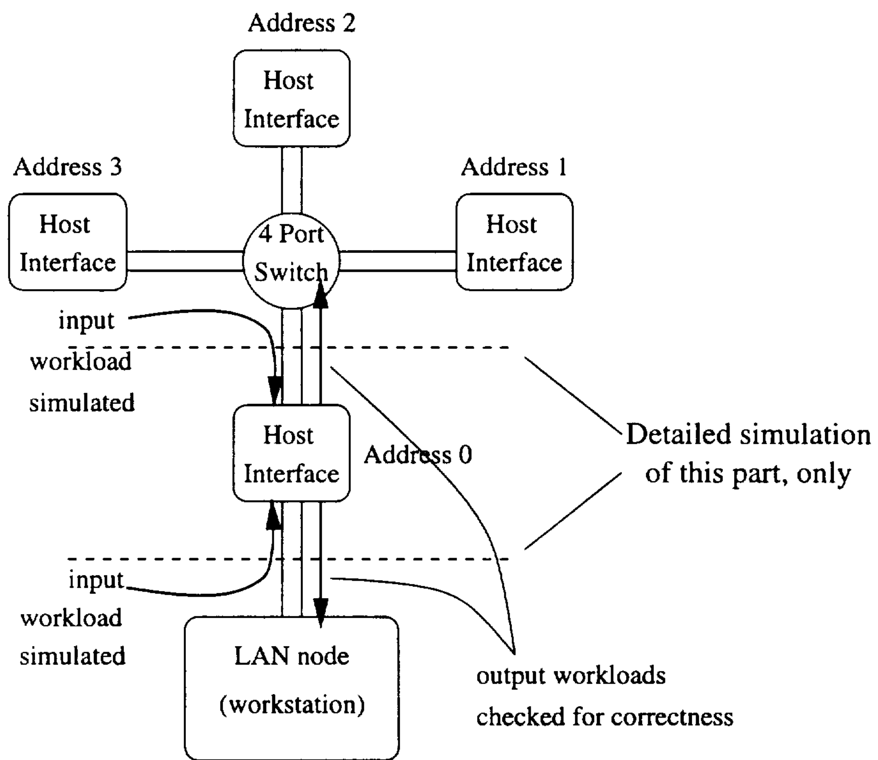

Figure 4.1: Picture of the simulated network.

additional software objects to emulate LANai specific hardware, to propagate the errors from the fault dictionary and determine their ultimate effect under the specified workload.

Only a single node of a Myrinet LAN is simulated in detail. A predefined workload is injected at the interface boundary between the host interface and the network and also at the boundary between the host interface and the host computer. These workloads represent arriving messages that are to be received from the network or sent out to the network. The effect of a fault is determined by examining the messages output by the interface to those same boundaries. If one of these messages is corrupted, the proper response of the network to the corrupted message is recorded (i.e., a message with an illegal route will be dropped). In determining these responses, an LAN with 4 nodes connected to a single 4-port switch is assumed, with the node undergoing fault injection considered as node 0.

Each of the simulated workloads contains a set of messages that arrive at specified times during the simulation for either receiving or sending respectively. A pattern similar to a parallel computation on the LAN is assumed, so the messages follow a repeating pattern of sending to all nodes and then receiving from all nodes a fixed-length message with a simulated period of 0.1 seconds. For this study, the data in each message was set to an arbitrary length of 52 bytes.

4.1.2 Fault model

Finally, in this case study, faults were injected into the LANai processor during only a single module of the MCP program, although their propagation was unconstrained. By injecting to only a single module, we were able to minimize the time spent on implementing the fault dictionary simulation and concentrate instead on modeling issues. In this study we constructed only a single fault dictionary, corresponding to the effect of a fault in the processor during its execution of the "send message" module. See Figure 4.2 for a picture of some of the tasks of the "send message" module. Some additional details of the fault injection are mentioned below.

The module that was selected was the code responsible for sending a message from the LAN node out onto the LAN. This code had to perform three major functions: transferring the message from the LAN node to the interface memory, preparing the message to go out on the network, including generating a correct route, and then transferring the completed message from the interface memory to the LAN. Execution of this module took approximately 540 cycles of useful time from the LANai processor, not counting cycles spent in DMA transfers. For each fault dictionary entry, a single fault was injected during this time, uniformly distributed across the 540 cycle period. At the end of the execution of the module, all persistent variables were written out into memory, and the contents of the memory were compared to those of a good program run in order to find the corrupted variables.

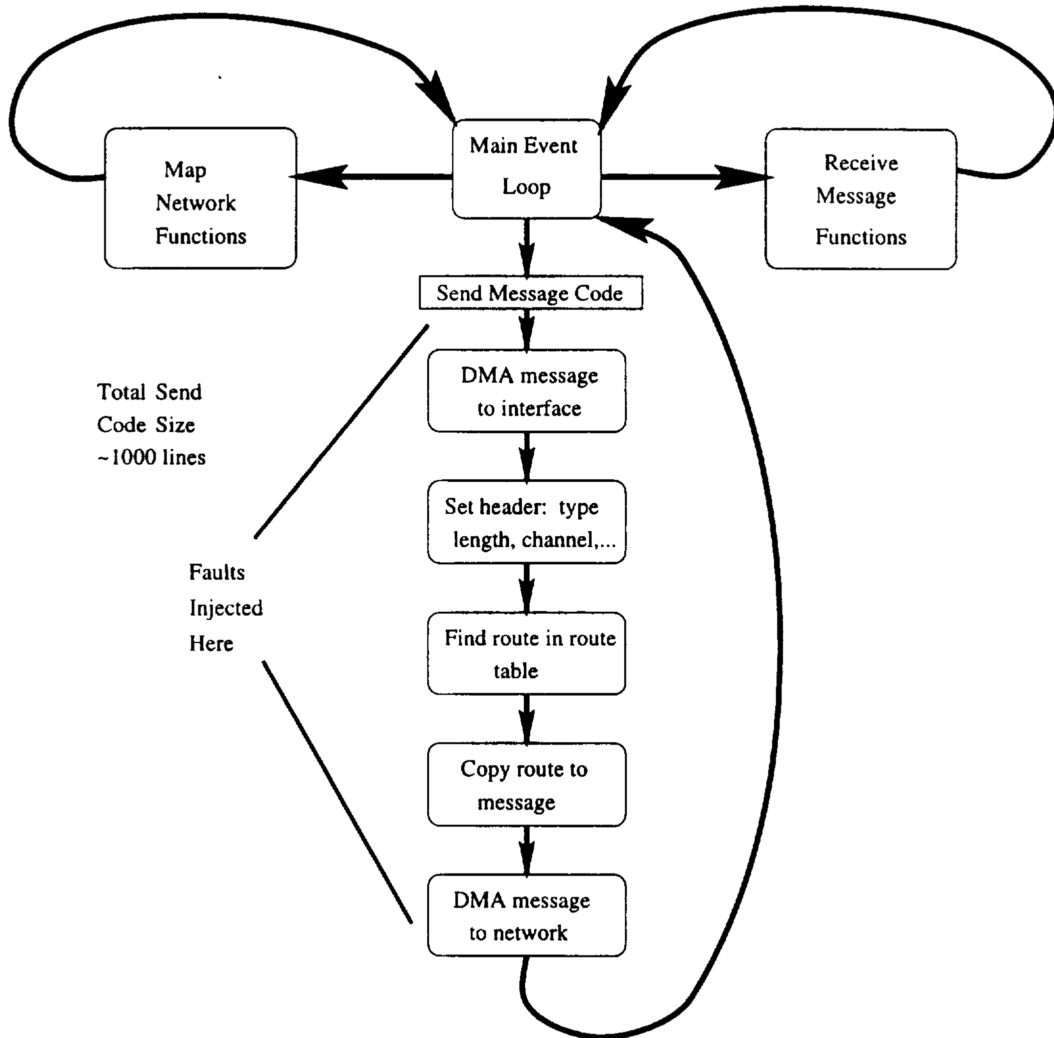

Figure 4.2: A block diagram of the MCP showing the "send message" module.

The faults that were injected in this study came from a behavioral-level fault model similar to that described in [22]. This behavioral-level model was chosen because a gate or lower-level view of the LANai microprocessor was not available at the time of the study. The fault model involves changing the assembly instruction executing on the microprocessor during the fault in a number of different ways to model faults in different parts of the microprocessor. Some of the possible effects of a fault in this model, selected with equal likelihood in this study, are: wrong source operand selected, source operand is corrupted, wrong destination operand is selected, additional destination operand is selected, and destination operand is corrupted. For additional details on the possible changes to an instruction to model different processor faults, consult [22].

4.1.3 Results

This section will describe the results of the host interface simulations in two parts: the fault dictionary generation and the system-level simulation.

Fault dictionary results

The first results we obtained for the host interface came from the behavioral-level simulations that formed the fault dictionary. These results consisted of a set of zero or more memory words that were corrupted at the end of the send message code as a result of a single fault injection.

An example of a small set of dictionary entries pulled out from the 1000 entries that were computed is given in Table 4.1. Each entry consists of a location where the fault

Table 4.1: A sample set of fault dictionary entries.

```
Loc: setup route lookup
START
   28f08         0     8182      corrupt start of route
END

Loc: find route in route table
START
    dd38         3        2      corrupt cached address
   28f08      8183     8182      corrupt start of route
END

Loc: copy route to message
START
   28f0c         0    10000      corrupt end of route
END

Loc: set DMA parameters
START
   28f30         0 54686973      Corrupt message data
   28f34         0 20697320      (all of these entries)
   28f38         0 736f6d65
   28f3c         0 20646174
   28f40         0 6120666f
   28f44         0 72206120
   28f48         0 6d657373
   28f4c         0 61676520
   28f50         0 77652077
   28f54         0 696c6c20
   28f58         0 73656e64
   28f5c         0 2e202e20
   28f60         0 2e202e00
END
```

occurred and a list of memory words that were found to have corrupted values at the end of the simulation of the given software module. For each of these corrupted words, the address, corrupted value, and good value are listed (in hex). (For example, in the first entry, the variable residing at address 0x28f08 was found to have a corrupted value of 0 rather than the correct value of 0x8182.) This information is used to identify the corrupted software variables so that the same corruption can be simulated at the software level as occurred at this behavioral level. In the table shown, the location of the fault is given in terms of what the program was doing when the fault occurred, and the meaning of the corrupted memory is given to aid the reader in understanding the character of the dictionary–normally the location would be in hex just like the memory addresses. The effects of four faults are depicted in the table. As a single fault may propagate to corrupt more than one variable, the second and fourth faults have multiple entries, notable by their multiple lines between the START and END commands.

Following is an incomplete list of possible effects to the software state that were observed in the generated fault dictionary:

1. Corrupt start of message route
2. Corrupt end of message route/message type
3. Corrupt cached address in routing table
4. Corrupt message data element
5. Corrupt all message data

6. Corrupt interrupt state machine (software object)

7. Corrupt message address

8. Corrupt message channel

9. Corrupt message length

10. Corrupt route in routing table

11. Corrupt pointer to message (move message in memory)

More than one of these effects may occur together from a single fault. For example, if the message address is corrupted by a fault before the route is looked up, the wrong route may also be written to the message, causing it to have both a corrupt address and route.

The effect of a fault as recorded in the dictionary was found to be strongly correlated with what the software was doing at the time of the fault injection for these behavioral-level faults. For instance, if the software was searching the routing table to find the correct destination and route, a fault was very likely to cause an invalid or incorrect route. This correlation occurred because, even though a fault can cause corruption to random memory words by corrupting the address of a load or store, the fault is much more likely to affect the variables that are currently being computed when the fault occurs due to the many references to these variables. Even if a fault corrupts a load or store, resulting in corrupting a random part of memory, the variables that were being

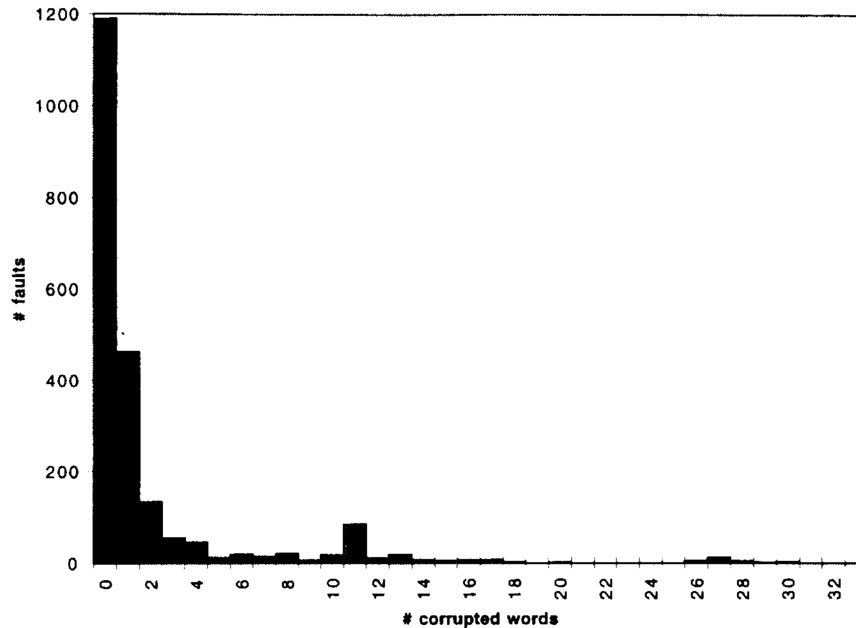

Figure 4.3: Number of faults leading to given number of corrupt words.

computed at the time are also likely to be corrupted because that load or store did not write the correct value to those variables but wrote to the wrong destination instead.

A graph of the number of faults which caused a given number of memory words to be corrupted is shown in Figure 4.3. The X axis of the graph is the number of corrupted words resulting from a single fault, and the y axis is the number of faults that caused the given number of corrupted words. As can be seen on the graph, the great majority of fault injections lead to no change in the program behavior or the corruption of a single memory word. The next peak in the graph is for faults that caused approximately 11 memory words to be corrupted, and the final peak is for approximately 27 corrupt words. Some reasons for these peaks are described below.

Just over half of the injected faults lead to no change in the program behavior. This result is due to a number of different effects. The first is that a fault injected during a NOP cycle was unlikely to cause any corruption in our model. Because the pipeline in this processor is software scheduled with respect to dependencies between instructions, a significant number of NOP's are generated in the compiled code to insert the proper delays between dependent instructions. Another reason a fault may have no effect is because the result it corrupts is not used. Sometimes a fault may lead to corrupting a register that contains a dead variable or corrupting the target address for a branch that is not taken. Finally, a third reason the program behavior may remain the same with a fault is that the effect of the fault is functionally masked. This means that the instruction had the same result with the fault as without. An example would be multiplying the wrong source register by zero. The result is still zero even though the choice of a source register was faulty. The combination of these three effects is what leads to so many faults causing no corruption.

The next largest group of faults lead to the corruption of a single word. The size of this group can be attributed to the fact that a corrupt instruction could directly cause only a single memory word to be corrupted. For further corruption to occur, the result of the corrupt instruction had to be used multiple times. Because the send message code to which we injected was very sequential with almost no loops, most variables were computed once and then written out to memory without being reused. Thus, the largest

group of faults that lead to a program change were those that corrupted a single memory variable.

The third and fourth largest peaks accounted for about 11 and 27 words respectively. These two peaks were caused by faults that corrupted the pointers in the send message code used in an array computation that wrote to 10 words. The first set is characterized by two different kinds of faults. The first kind changed a pointer to the input data, causing a significant number of variables to be miscomputed. The second kind changed the output pointer, but only slightly, so that the correct results were output to memory but offset one or two locations, again leading to many corrupt words. The peak at 27 was similar to that second kind of fault in that the output pointer was changed. This time, though, it was changed to point to a significantly different part of memory. So, not only were the original 10 array words corrupted, but at least an additional 10 words in another part of memory were corrupted by writing the output results there instead.

The computation of the fault dictionary here was interesting because it showed that each software module will have its own characteristic faulty behavior, and that faulty behavior can be characterized by the way in which the variables in that module were used. The generation of the fault dictionary was only an intermediate step in this analysis, however. The use of the fault dictionary in a system-level simulation of the host interface to determine the ultimate effect of the fault is described in the next subsection.

System-level simulation results

Once the fault dictionary was computed, it was used as a fault model at the system level. Here, the host interface was simulated handling a predefined message workload–both sends and receives. When it was determined that a transient fault occurred within the LANai processor, the fault dictionary was used to determine what errors would occur in the executing software as a result of the fault. These errors were then injected into the software state, and the execution of the software continued, typically with a change in behavior because of the fault injection.

At this level, we were interested in the ultimate effect the fault would have on the software behavior in terms of things visible to the user. For that reason, the results here are in terms of the effect on the messages that were to be sent or received by the interface. If all of these messages were sent and received correctly, then the fault was said to have no effect at this level.

We injected 1000 sets of errors, each set corresponding to the impact of one fault on the send message module of the MCP, from our fault dictionary into the running software. Each time we performed an injection, it was assumed that a long enough time had passed since the last fault that the system was fault free. This assumption seems reasonable because the host interface regularly updates its persistent variables, such as network routes, and an interface that doesn't respond for a significant period of time will be reset. Note, however, that there is nothing preventing us from considering the impact of near-coincident faults on a software module in forming our fault dictionaries;

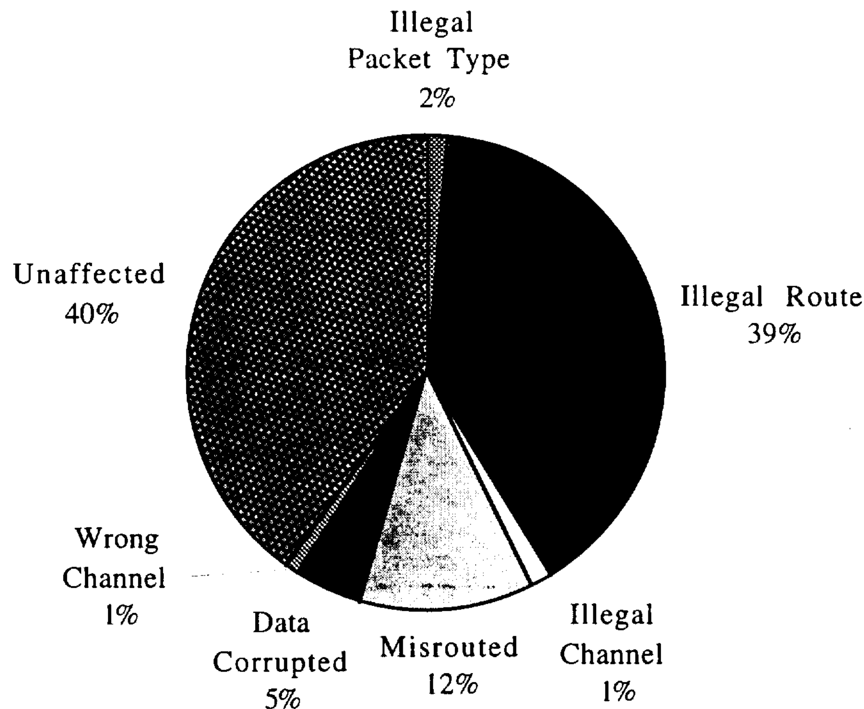

Figure 4.4: Effect on messages sent during fault lifetime.

they simply weren't considered in this example study. In addition, the period between network route updates was modeled so that faults that caused changes to network routes would be able to effect messages only until the next route update.

Figure 4.4 shows a breakdown of the errors seen in the messages workload. Of the faults injected, 40% cause no errors in the transmission of the network workload. Of those messages that were affected by faults, it can be seen that the majority of them ended up with illegal routes and were dropped in the LAN. The next largest group of errant messages were misrouted with a legal route, and the third largest group consisted of messages with corrupted data. For those 60% of faults that did lead to an error, 95.5% of them affected only a single message, and the other 0.5% affected multiple messages (all the messages to the same destination until a mapping update was done).

4.2 Modeling an Entire Myrinet LAN With Validation

This second case study built upon the work in the first with the goals of modeling an entire Myrinet and verifying the results against a real system. The system model for the simulation was therefore extended to modeling in detail all of the interfaces in the four-node LAN from the first study, and the object encapsulation and custom pointer class special techniques were added. A behavioral-level fault model was still used, however, to facilitate the fault injections into the real system that would be necessary to do the validation. This comparison study first appeared in [23].

Before describing the simulation models that were used and the results, however, it is important to understand the way in which the simulation was to be validated against the real system and why this way was chosen. The validation was done by choosing a fault model that could be injected into both systems (real and simulated) and then injecting identical faults from this model into both systems and comparing the fault impacts. The same fault model was used in each system to eliminate potential differences in the results due to differences in the fault models. The impacts from identical faults were compared in order to avoid the large number of injections that would be necessary to compare distributions from two different fault lists.

There were two further reasons for comparing the fault impacts on a fault-by-fault basis. First, the comparison was more precise than just comparing the distributions. For example, the simulation could show the same number of system reset results as the real system but not be 100% accurate because some of the reset results were attributed to

different fault injections than in the real system. Thus, some of the faults that caused resets in the real system didn't cause resets in the simulation. The simulation was then said to predict the wrong behavior for those faults. Second, the fault-by-fault comparison allowed the percentage of faults for which the real and simulated systems matched to be examined by result category. A low percentage of matches between the two methods for any one category showed a weakness of the simulation at modeling faults with impacts in that category.

In this case study, the two claims that were made about the fault dictionary method in the second chapter will also be revisited. In particular, the first claim said that the fault dictionary helped reduce the number of faults that must be considered at higher levels because many of those faults would result in no error or in errors identical to an entry already in the fault dictionary. The second claim was that the behavior of the majority of faults would be independent of the software state when the fault was injected.

4.2.1 System model

As in the first study, the system model consisted of a four-node Myrinet LAN with faults being injected into the host interface of only one of the nodes. For this study, however, each of the four host interfaces was modeled in detail, including executing its own copy of the MCP, and the interfaces communicated through a simulated Myrinet switch. The full MCP software was modeled in this study, including the cooperative mapping functions. The host workstations connected to each of the four interfaces were

not modeled in detail. Instead, only the application (a synthetic workload described below) was modeled.

A real Myrinet testbed was set up, as well, to allow fault injections to be duplicated on a real Myrinet. Fault injection was done to one of the interfaces on this real network using Software-Implemented Fault Injection (SWIFI). The testbed and simulation model were made to be as similar as possible to facilitate the comparison of fault injection results from the two injection methods. Thus, the testbed also consisted of a four-node Myrinet connected through a switch. The same addresses and switch ports were assigned in the simulation and testbed. A second network, an Ethernet that was not undergoing fault injection, was used in the testbed to provide control and monitoring functions. A picture of the system configuration is shown in Figure 4.5.

4.2.2 Fault model

A simplified fault model was chosen that would allow fault injections to be easily duplicated for the two fault injection methods. Faults were injected as transient single bit flips of one of the instructions to be executed in the "host send" message module. A diagram of the "host send" message module of the MCP is given in Figure 4.6. (This module is similar to the send module used in the first study, but due to revisions of the MCP between studies, the same send module no longer existed.)

In the simulation experiments, faults were injected by corrupting one of the instructions in the given module at the start of each of the behavioral cycle-accurate simulations.

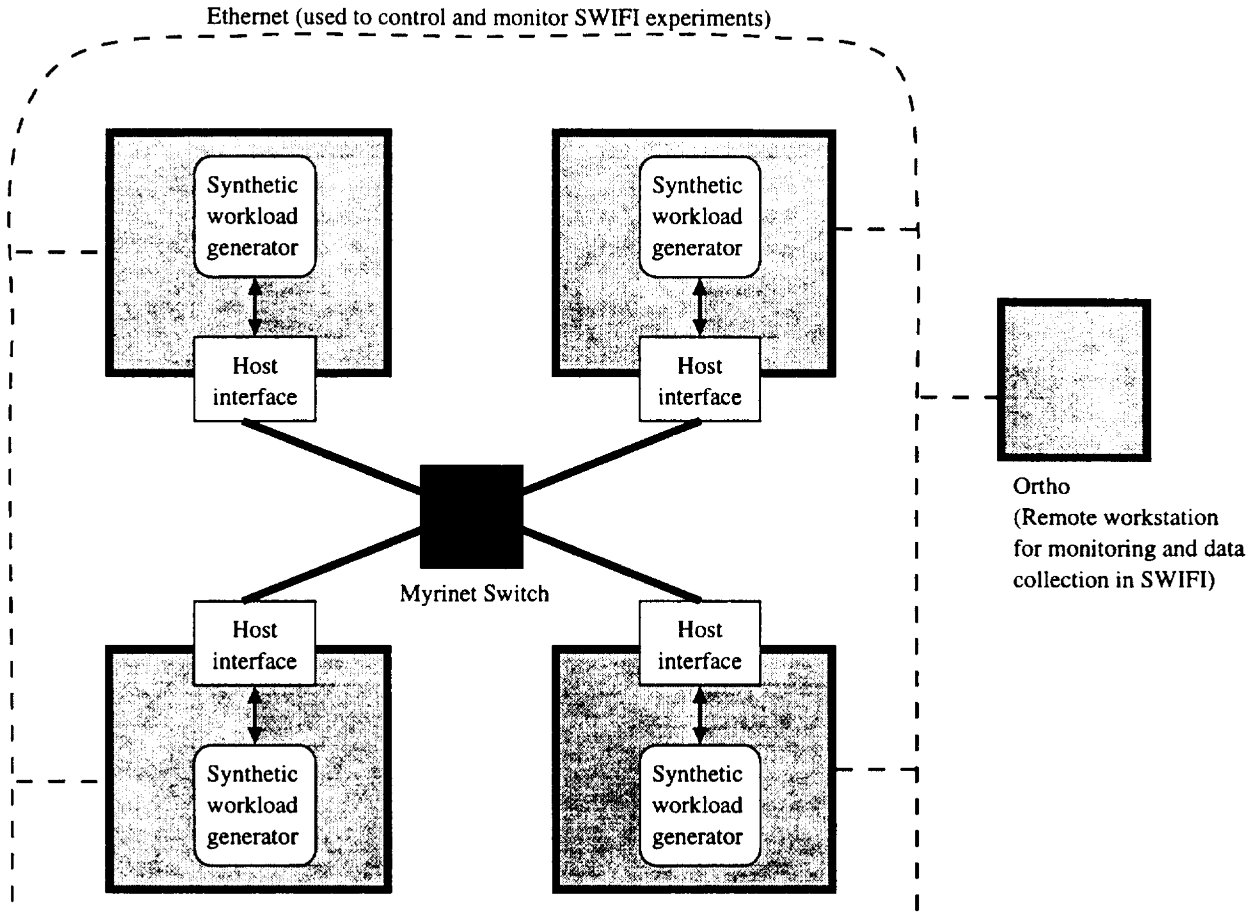

Figure 4.5: Target system for fault injection (simulation and SWIFI).

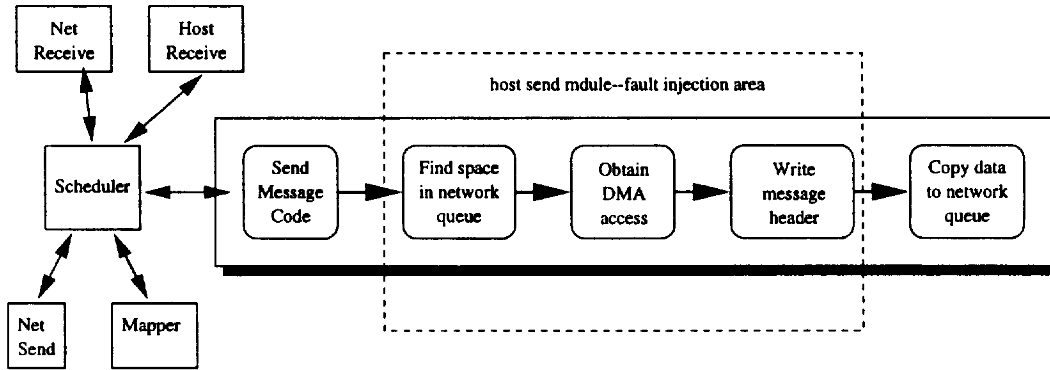

Figure 4.6: Diagram of fault injection region.

The execution of the module with the fault was then simulated, and at the end of the module, a fault dictionary entry was made recording the corrupted control flow and variables due to the fault. The dictionary was then later used in system-level simulations to determine the ultimate impact of each fault.

In the SWIFI experiments, faults were injected in a similar way. The execution of the MCP on the interface undergoing fault injection was paused at the start of the "host send" module. Then, the selected instruction was corrupted by the interface's host computer. Note that the same instruction and same bit would be corrupted for the SWIFI as in the simulation for a duplicate fault. The interface would then continue executing the MCP module with the fault. At the end of the module, execution would pause again while the original instruction was restored.

Each of the fault injections was repeated 10 times in the SWIFI experiments to ascertain the repeatability of the result. Faults that showed only one error behavior and did so for at least 6 of the 10 injections were used in the validation comparison. Other faults that showed multiple error behaviors for different fault injections or that only rarely caused errors were thrown out.

In this way, faults whose behavior was highly dependent on the software state when they were injected were identified and removed from the comparison. These faults would complicate the comparison between the simulation and SWIFI experiments because it would be difficult to make the software states exactly the same between the two. For the

remaining faults, however, that showed very little dependence on the software state, the two systems should be easily comparable.

4.2.3 Results

In the comparison study between the simulation and SWIFI on the real system each method injected the same set of 500 faults. Some of these injections (4 injections) had to be discarded because they caused the simulator to crash. Of the remaining faults, 423 were found to cause very repeatable results and so to be fairly independent of the software state. The breakdown of the comparison for these 423 faults is presented in Table 4.2.

Table 4.2: Number of errors by category for simulation and SWIFI.

Fault injection result	Simulation	SWIFI	Match
MCP hang	61	82	62.2% (9.4%)
MCP restart	6	16	37.5% (20.9%)
Message dropped	58	55	94.5% (6.0%)
Data corrupt	19	19	84.2% (16.4%)
No error	279	251	97.6% (1.9%)
Total	423	423	87.5% (3.2%)

The leftmost column of Table 4.2 shows the fault injection results. Faults in the MCP hang category caused one interface's MCP to stop performing some or all of its functions. Faults in the MCP restart category cause one MCP to reset, momentarily disrupting communication and causing it to drop all the messages currently in its buffer. Those in the message dropped category caused one message being transmitted on the the Myrinet to be dropped. The data corrupt category was used when the only impact

of a fault was to corrupt some of the data in one of the messages being transmitted, and all of the remaining faults fell into the no error category, signifying all messages in the workload were sent and received correctly, even with the fault.

The second column shows how frequently the selected result category was observed in the simulations. The SWIFI column shows the frequency with which the selected result category appeared in the real system. Finally, the match column shows how often the simulation and the real fault injection results were identical, and the number in parentheses gives the 95% confidence interval. In computing the value for the match column, 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 52 injections in the *message dropped* category. The maximum possible number of matches for this case would be 55 (100%). Thus, for the *message dropped* category the simulator accuracy was 94.5% (52/55), plus or minus 6.0% for the 95% confidence interval.

Table 4.2 shows that the simulation does extremely well at detecting when no error will occur (matching SWIFI for over 97% 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 where SWIFI determined that result). The simulation, however, has relatively low accuracy in predicting severe fault results, such as host interface hang

where 62% 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. Factors affecting this accuracy rate are addressed in detail in the following subsection.

The two claims made in the second chapter will now briefly be revisited. The first claim was that the fault dictionary would help reduce the number of simulations that were necessary at higher levels by removing many faults that didn't propagate or had identical error patterns to another fault already considered. For the 500 faults presented above, injected at the chip level as single bit flips of instructions, 245 did not propagate to the system level because they caused no change in the software state of the host send module. The remaining 255 faults can be divided into three categories. One-hundred ten faults had identical error patterns to another fault in the dictionary and could be discarded. Sixty five faults had error patterns that were similar to another fault in the dictionary, but not identical. That is, these similar faults corrupted exactly the same software variables but the corrupted value for one or more of the variables was different. With no further analysis, these 65 faults *did* have to be simulated at higher levels to be sure of their impact. However, faults with similar error patterns were very likely to cause identical results, and further analysis may have been able to identify faults with significant differences from those without. Finally, 80 of the 500 faults caused unique error patterns. In this study, therefore, one would have had to simulate at the system level the 80 unique faults plus the 65 similar faults in the worst case, for a total of 145

faults (out of the original 500). With further analysis, it may have been possible to reduce this number by eliminating some of the 65 similar faults.

The second claim was that for embedded microprocessor systems with control software like the one examined in this study, the majority of the fault impacts would be dependent only on the fault injected and the design of the hardware and software modules, not on the particular state of the software when the fault was injected. The above case study supports the claim because of the 500 faults injected, 423 faults had a very repeatable behavior that was independent or nearly independent of the software state. For these 423 faults, it was only necessary to consider a small set of representative states of the software to obtain the correct behavior.

4.2.4 Discussion

The initial weeks of our comparison effort turned out to be a learning process. A number of problems associated with our simulation efforts made it difficult to match the behavior of the simulation to that of the real device. Those problems are discussed in this subsection, including limitations in the simulator model, specification problems, and effects of the simulation environment.

Cycle-Accurate Simulator Limitations

The cycle-accurate simulator had three basic limitations for our purposes. First, the standard version supplied by Myricom simulates only the CPU core of the LANai chip. None of the memory-mapped I/O on the real chip is implemented. As a result, we were

restricted from simulating certain regions of the MCP code at the cycle level and so could not inject faults into these regions. The simulator could be extended to model the entire LANai chip; however, we chose instead to find an important region of code (the "focused send" region) that could be simulated in the cycle simulator without modifications. In this way we were able to make a convincing argument for the validation of the simulation without bringing in the additional issues of developing and testing new features in the cycle simulator.

The second limitation in the cycle simulator was that it was not guaranteed to match the host interface behavior for certain error conditions, such as the response to an illegal instruction or invalid memory access. One particular difference we noted was that the simulator did not implement memory protection of the low memory segment, where the MCP is stored, as was done in the real host interface. Some faults were observed to attempt writes to this region. While these writes would be discarded in the real interface, they were allowed to proceed in the cycle simulation. In the examples we observed, the lack of memory protection did not appreciably alter the results of the simulation, but the potential certainly exists.

The third limitation applies to cycle-accurate simulation in general. Such simulators are typically thousands of times slower than the real device they simulate. For most faults, this simulation time was very short because we could quickly translate the fault effect to the software level. There were rare faults, however, that would change the program counter to a random value. In these cases, execution would be outside of typical

program paths, and thus the effect of the fault could not be translated to the software level right away. If execution never returned to normal, a hang could be assigned to these cases. However, this *random* code could lead to a reset of the LANai chip or a return to normal execution. The only way to be sure of the result was to continue simulation. Due to the cost of simulating at this level, however, an arbitrary limit of 80,000 cycles was set. If the program had not reset or resumed normal execution within this period, the result was marked as a hang.

Specification Problems

Another problem we encountered in the development of our simulations was that the response of the LANai device to various error conditions was very vague or missing in the official device specification. One example of unspecified behavior was 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. This information is clearly stated in the specification. However, the behavior when accesses are not aligned is not specified. While it is understandable that such information is unnecessary for programmers, it is necessary for the proper simulation of the device, particularly when such error conditions are likely to occur due to fault injection.

Some of the unspecified behavior we came across was cleared up through discussion with Myricom. In other cases, however, we did not even realize we had overlooked some

error condition until we observed it to cause a difference between the SWIFI and simulation experiments. A particular example involved a fault that changed the DMA_DIR register. This register 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, and so writing a five or a one to this register would give the same result. Experiments on the real device showed, however, that values other than zero or one written to this register could cause a hang. Experiences like this one led to a period where we, in essence, trained our simulator by correcting its behavior.

Effects of Simulation Environment

One problem in simulation is deciding where to draw the boundaries of the simulated system. Interaction between objects inside the boundaries and those outside that have been abstracted away may cause the simulation to act differently than the real device. One case where this problem appeared in our study involved a fault in the real host interface that caused the reset of its host workstation. Because the host was outside our simulation boundaries, we did not model any of the interaction between host and interface. Therefore, we were not able to predict this occurrence.

Another effect of the environment was due to our execution of the MCP code natively on a workstation (for the software-level model). While this approach allowed the *simulation* of the MCP to be very fast, it posed a problem. The LANai chip itself has only a limited memory protection that simply discards illegal accesses, but on a workstation,

illegal memory accesses can cause the termination of an application. While every attempt was made to avoid crashing of the simulations due to memory access violations, 48 such crashes still occurred (see Section 6.1).

Finally, because the simulation engine and the MCP shared one user process in our software-level simulation and had to communicate with each other, it was possible for an errant MCP to corrupt variables belonging to the simulation engine. In the runs we recorded, this behavior was not observed, but it was considered in our design. Avoiding this problem, at least in part, would have meant distributing our simulation among multiple processes so as to provide the operating system's memory protection to each simulation process. Such a design was beyond the scope of this study.

4.3 Inclusion of Recovery to Improve Dependability

In the third case study, the results of the validation study were analyzed in an attempt to improve the behavior of the MCP software under faults. One key result from the validation study was the high number of fault injections that lead to an MCP hang. As this result category was also the most severe, it was targeted for recovery.

4.3.1 Recovery added

Three of the most common mechanisms that caused an MCP hang were identified from the simulation results. Here, the behavioral-level fault dictionary was very useful since it gave a breakdown of how different faults affected the MCP software variables and

control flow. From this dictionary, typical patterns for erroneous variables and control flow could be identified so that recovery could be added.

The first of the three mechanisms was hole in the received message acceptance tests that passed erroneous messages with data lengths of zero bytes. These messages were invalid, and handling them caused the receiving MCP to hang. The recovery for this mechanism just plugged the hole in the acceptance tests by adding a check for zero-length data messages.

The second mechanism that was targeted for recovery involved the state of the send message buffer called NetSendQueue. A large number of faults could cause the MCP to erroneously determine that this queue was full when there was still space remaining. Once the MCP decided the queue was full, it would accept no more messages for sending until the queue made a transition from full to not full. Because the queue was really not full already, though, this transition would never occur, and the MCP would send no more messages. The recovery for this mechanism added a timeout to the wait for the queue to transition to not full. If the timeout expired without the not full notification, the MCP would recheck the state of the queue. Typically, this second check would allow it to make the correct determination and resume normal behavior.

The third mechanism was very similar to the second, but it involved the host to LANai DMA rather than a queue. Some faults would cause the MCP to erroneously determine this DMA interface was in use when it was not, and the result was that the MCP would wait forever for the interface to become free, similar to the situation with

the send queue above. Again, the solution here was to add a timeout to this wait period. If the timeout expired without a notification the DMA was free, the MCP would check the state of the DMA again and typically continue with normal operation.

4.3.2 Results

After the MCP code was changed to add the three types of recovery mentioned above, a similar set of fault injections was run as in the verification study. That is, the target system model (aside from the software changes) was the same four-node LAN, the same fault list and workload were used, and experiments were run on both the simulated system and the real testbed.

The results of the experiments are shown in Table 4.3. The number of faults causing a given impact are listed for each of the five error categories for the simulation and real system as before. This time, however, there are two numbers separated by a slash. The first number is the impact of the fault *with* the new recovery routines added, and the second number gives the old behavior.

Table 4.3: Comparison of errors before and after recovery was added.

Fault injection result	Simulation	SWIFI
MCP hang	61/36	82/32
MCP restart	6/6	16/16
Message dropped	58/63	55/67
Data corrupt	19/19	19/19
No error	279/299	251/289
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 drop in the number of faults that caused hangs from 58 to 29, or a 50% decrease. Note that the recovery that was added was all outside the fault injection region, so the faults that were injected could be made exactly the same as in the verification study. Of those same faults, 29 no longer caused a hang after the addition of the recovery code.

Of the remaining faults that did cause a hang, 15 were due to random changes to the program counter that put it outside normal program control flow paths. These faults could not be targeted with simple software recovery mechanisms. The remaining faults included some that used a variation on the three targeted mechanisms that could slip through recovery, and the rest used a mechanism that was not targeted by the recovery.

The next step was to implement the same recovery features on the real system and verify the effectiveness. Of particular interest was whether the recovery would be as effective on the real system given the difficulty the simulation had in predicting hangs on the real system. While the three mechanisms that were targeted for recovery were valid mechanisms for which the simulation was known to simulate the correct behavior, it was unclear whether they would help for the hangs in the real system which the simulation didn't predict.

The results showed the recovery to be every bit as effective in the real system as the simulation had predicted. The number of faults that caused a hang result in the real system dropped from 82 to 32 for the same faults, a 61% decrease.

This study demonstrated the usefulness of the simulation method in determining failure modes for the system software and identifying recovery mechanisms, even when the simulation model was not detailed enough to model all of the failure modes present in the real system. Even though the simulation was not able to identify all of the faults that would hang the Myrinet system, the simulation analysis was still very useful in determining what some of the common failure modes were and guiding the development of recovery for those modes.

4.4 Incorporation of a Device-Level Fault Model

The final case study presented here was designed to demonstrate the use of the entire fault dictionary hierarchy, from a device-level fault model up to a system-level result. To accomplish this goal, the same system model and workload was used as in the previous two studies. That is, multiple host interfaces were simulated in a four-node LAN, and the interfaces were logically arranged in a circle, with each interface receiving from its counterclockwise neighbor and sending to its clockwise neighbor. The fault model and fault dictionary structure were changed, however, from a two-level to a five-level hierarchy. This case study was first published in [24].

Because of the unavailability of a gate-level or transistor-level model for the LANai chip, however, a special strategy had to be used to allow a demonstration of the full fault dictionary hierarchy. The solution was to decide on a subcircuit of the LANai chip that was frequently used and was visible to the behavioral-level simulator and to generate transistor-level models for that subcircuit. The chosen subcircuit was the ALU 32-bit adder. Because the code simulating the adder in the behavioral-level simulation was easily identified and was a good match to the behavior of the transistor-level description of the adder, it was easy to propagate the errors from the adder faults directly in the behavioral-level simulation rather than going through an intermediate gate-level simulation step. Thus, a complete gate-level model of the LANai chip was not necessary. A full description of the fault dictionary hierarchy is given below and may better illustrate this strategy.

4.4.1 Fault dictionary hierarchy

The fault dictionaries in this study were designed to begin with a device-level fault model for a radiation particle impact on the adder circuit and end with the system level effect of that event. The steps of this process are described below, from the beginning at the lowest level until the end when the system level is reached.

Device level

Fault injection began at the device level in a two-level simulation step to determine an appropriate current burst due to a heavy ion particle impacting the adder circuitry. The fault event was the impact of a radiation particle with an energy of 8 MeV at a

reverse-biased transistor junction at an angle of 45 degrees to the surface. The device parameters for this stage were taken from a 0.25-μm CMOS technology.

The DESSIS simulator was the first level of this two-level approach, and it was used to model the MOS transistor that was hit by the particle. The DESSIS simulator solved the coupled Poisson, Electron, and Hole equations of the transistor in the transient mode to calculate the resulting current surge.

The second level of this simulation step was the transistor-level simulator called HSPICE. HSPICE was used in an iterative manner with the DESSIS simulator to provide the bias conditions in which the effected transistor was operating.

Together, the two simulators obtained the model for the particle impact in the following way. First, HSPICE would would simulate a short period of time which would define the original bias conditions around the MOS transistor. Next, DESSIS would begin simulation of the particle impact for a short period, obtaining the beginning of the resulting current burst. This current burst was then entered in the HSPICE model, and simulation continued a timestep further, resulting in a new bias. The new bias was then put back in DESSIS to generate the next timestep for the current burst. The back and forth iteration proceeded in this way until the whole particle impact completed, resulting in a time-varying model for the current burst.

Logic level

The next fault dictionary level was the logic level. Here, the target system was now the whole adder, not just one transistor. The 32-bit adder was organized as eight 4-bit Manchester carry adder stages with ripple carry between the stages. Faults, in terms of the current burst derived in the device-level step, were injected, one per simulation run, at reverse-biased transistor junctions in the adder circuit as it performed a computation. The errors latched at the outputs of the adder circuit were then recorded as the logic-level fault model for the adder.

For each input combination, faults were injected exhaustively at all reverse-biased transistor junctions, and the error patterns observed along with their frequency were recorded. Symmetry in the adder, such as the fact that the 32-bit adder was made of 8 identical 4-bit adders was used to reduce the number of simulations necessary in this step. At the end of this step, the output was a fault dictionary that listed for each input combination the likelihood of observing each possible error pattern given the occurrence of a single radiation particle impact on the circuit.

Chip level

The logic-level fault dictionary was then used as the fault model in a behavioral-level simulation of the LANai chip. This step used the same behavioral-level simulator as was used in the other three case studies, and faults were injected into the same "send message" module of the MCP. Instead of a single-bit flip of an instruction, however, faults

were now injected as corruption of one of the add operations during the execution of the module.

More than half of the instructions in the 67 instruction region where faults were injected used the adder. Add operations were required by load and store instructions to compute effective memory address as well as by add and subtract instructions to compute arithmetic results. Also note that comparisons are generally performed by subtracting one of the numbers to be compared from the other, and thus comparisons are coded as subtracts and also use the adder. Finally, NOP instructions are coded as adding zero to zero and discarding the result, and so they also use the adder. Discarding the NOPs, there were 22 instructions that actually relied on the results of the add operation.

Faults were injected exhaustively during the execution of every add operation, including all possible error patterns in the logic-level fault dictionary, one per simulation run. At the end of the execution of the module, the software state of the simulated MCP was examined for erroneous control flow and variables. These errors were recorded in the chip-level fault dictionary as the chip-level model for the given fault.

4.4.2 Results

Finally, the ultimate impact of each fault was determined by a system-level simulation that incorporated the chip-level fault dictionary. This system-level step was identical to that of the previous two case studies, and the same result categories were used. The overall results of the fault injections are shown in Table 4.4.

Table 4.4: Breakdown of number of errors by category.

Fault injection result	Simulation
MCP hang	112
MCP restart	32
Message dropped	134
Data corrupt	54
No error	918
Total	1250

One number that stands out in the table is the large number of fault injections that caused no impact. It is important to note that there are many reasons a corrupt add operation might not impact the host interface's operation. The most common reason is that the message buffers are being continually reused, and the new message that is being written into the buffer may share many header items in common with the past message in that buffer. If so, then a failure to write some information into the buffer (due to an address miscalculation) may not have any impact because the correct information is already there. As long as the failed write does not overwrite some other important data, it has no effect. Another common reason a fault may have no impact occurs when the adder is used to compare two unequal numbers. In this case, the adder is used to subtract one number from the other. A zero result indicates equality. However, a single fault in the adder is unlikely to change a nonzero result into a zero result. The comparison decision, then, doesn't change, even though the adder result is wrong, because the result is still nonzero, indicating unequal numbers.

Breaking up the injection results by the type of instruction affected reveals some more interesting points. Table 4.5 shows the result categories according to whether the affected instruction was a load, store, add, or subtract.

Table 4.5: Breakdown of number of errors by category and instruction type.

Fault injection result	Load	Store	Add	Subtract
MCP hang	29	–	19	64
MCP restart	26	1	5	–
Message dropped	89	45	–	–
Data corrupt	27	–	27	–
No error	223	468	62	165
Total	394	514	113	229

As can be seen from the table, the add operations in loads are more important than those in stores for the code in the fault injection region. (Stores have only a 9% chance of causing a user-level error whereas loads have a 45% chance.) One reason for this behavior has already been given above: a given store may be unnecessary because the correct information is already in memory. Where writing to a wrong address is not so disruptive for a store, loading from the wrong address is much more likely to cause an error. The load result is guaranteed to be used in a future computation, while the bad address to which the store wrote is only sometimes used. Since a load from a wrong address almost always loads a wrong value, loads are much more likely than stores to generate a wrong value that will be reused and eventually cause an error at the user level.

Another interesting point from the table is that loads lead to all of the different types of result categories. Loads are used to begin computations that affect message headers, and corrupting these loads can cause message drops. Loads also set pointer values for the

message data, and thus corrupt loads can cause message corruption. Finally, loads are used in computations to make decisions about the state of shared resources. Corrupting a load that is determining the state of a shared resource can lead to a deadlock for that resource or a crash because of an invalid use of the resource.

Almost half of the subtract instructions can be seen to cause hangs in the table, and subtracts cause no other kind of result. Only hangs occur because the only use of subtracts in the fault injection region is to make comparison decisions about the state of shared resources. Making the wrong decision in these cases typically causes the MCP to hang. Four decision points are affected in the code, two of them are typically unequal and two equal. As mentioned above, comparisons that would generate an unequal result in the fault-free case are unlikely to be affected by a single fault in the adder. Equal comparison results are very likely to be affected, though. As a result, almost half of the subtracts cause an impact at the user level because half of the comparisons in the fault injection regions would normally be equal in the fault-free case.

5. CONCLUSIONS

This chapter summarizes the work that was done in this thesis and then describes some future work which could be done to improve the method that has been presented.

5.1 Summary

This thesis has presented a new method for simulated fault injection analysis of computer systems. The key points where the method differs from previous work are allowing detailed fault models, simulating the impact of the faults on system software behavior as well as hardware behavior, and obtaining the impact of the faults at the user level.

The key technique applied to meet these goals is the use of fault dictionaries to raise the abstraction level of faults. A fault dictionary for a given component characterizes that component's behavior for faults occurring within it. The fault dictionary stores the change in behavior of the component for many different faults. When a fault is to be injected in a given component, the dictionary is accessed, and the components behavior is modified according to one of the appropriate entries in the dictionary.

A fault dictionary used at one level of abstraction is developed through many simulations of the given component at the next lower level of abstraction. Fault dictionaries are built from the bottom up. That is, simulations begin at the level of abstraction of the primary fault model, for instance at the transistor level for a transient current burst model. From there, multiple simulations are run to build up a fault dictionary which can be used for fault injections at the next higher level, for instance, the logic level. Simulation then continues at that higher level, building a new fault dictionary. The upward propagation continues until fault effects reach the system level and their ultimate impact on the system, as it is visible to the user, is determined.

Four case studies were presented, demonstrating the use of the method to analyze a commercial network system called Myrinet. Each case study focused on presenting a different piece of the method in detail. In the first study, the focus was on the use of the method to model the behavior of the system software. In the second study, the focus was validating the simulation method versus similar fault injections into a real Myrinet. The third study showed how the analysis from the method could be used to improve the behavior of the Myrinet under faults, and the fourth study demonstrated the use of the complete fault dictionary hierarchy, from the device level to the system level.

5.2 Future Work

Two ways in which this work could be extended will be discussed in this section. There are, of course, improvements that could be made to the case studies presented earlier,

but the two ideas presented here are both improvements to the basic fault dictionary simulation method.

First, the software-level fault dictionary representation could be made more generalized. The current fault dictionary records the impact of faults in one module for just one state of the software. To include additional software states in the study, new fault dictionaries must be computed, one per each new state. From observations made during the case studies in this thesis, however, the impact of many faults remains the same for multiple software states. This data suggests that the computations for many faults need not be repeated for many software states.

Instead, the fault dictionary entry could be modified to describe those parts of the software state that the dictionary entry depends on. If those parts of a new software state match the state for which the dictionary entry was made, the same entry is used rather than computing a new entry for the new software state. In this way, a fault in a module that will cause a system reset independent of the software state when it enters that module will be computed only once in the fault dictionary computations.

Second, the demonstrations presented in this thesis and the special techniques that have been described are all tailored towards simulations with hardware transient faults. The generalized fault dictionary method could also be applied to other types of faults, however. One valuable addition to this work would be its extension to other types of faults, particularly permanent hardware faults.

The difficulty in using this method, as is, to model permanent faults is that permanent faults do not alter the system state at just one point in time. A permanent fault cannot be modeled, then, by using a single fault dictionary entry to modify the system state as required by the fault. The easy solution, repeated application of fault dictionaries at each successive module of software execution, isn't very tractable, either. At each successive module, the software state may be further corrupted by previous execution with the permanent fault. If this corruption were not considered in forming the fault dictionary entries that were used, the entries may not represent the correct module behavior at the current point in the software's execution. If the previous corruption of the software is taken into account, however, the fault dictionary computations may become unmanageable due to the large number of software states that must be considered, both fault-free and with corruption due to permanent faults.

A good approach to implementing permanent faults may be to combine the idea of fault dictionaries with a form of hierarchical simulation. The hierarchical simulation would differ from the normal fault dictionary method in that the faulty behavior of a module would not be precomputed (in a dictionary) but instead computed on the fly by dynamically switching simulation execution to the lower level at which the dictionary would have been made. Fault dictionaries would still be used, but only at the lower levels of system abstraction where the software state wouldn't impact the dictionary entry.

For example, consider a permanent fault in an adder. Fault dictionaries could be computed for this fault up to the behavioral level, where the adder takes two inputs and

outputs the sum. The software state doesn't impact the operation of the adder here, only the inputs. Thus, a fault dictionary can be made for permanent faults in the adder. If a software module uses the adder, it will find it has no fault dictionary at the system level. Execution will then dynamically switch down to the behavioral level where the already computed fault dictionary for the adder can be used. Additional software modules using the adder will require continued simulation at the behavioral level, but the hope is that the permanent fault will be quickly detected, ending the need for further simulation.

REFERENCES

[1] R. K. Iyer and D. Tang, *Fault-Tolerant Computer System Design*, Chapter 5, "Experimental Analysis of Computer System Dependability." Prentice Hall, 1996.

[2] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J. Fabre, J. Laprie, E. Martins, and D. Powell, "Fault injection for dependability validation: A methodology and some applications," *IEEE Trans. Software Engineering*, vol. 16, pp. 166–182, Feb. 1990.

[3] 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*, July 1992, pp. 336–344.

[4] K. Prodromides and W. H. Sanders, "Performability evaluation of csma/cd and csma/dcr protocols under transient fault conditions," *IEEE Transactions on Reliability*, vol. 42, pp. 116–127, March 1993.

[5] W. H. Farr, "A survey of software reliability modeling and estimation," Tech. Rep., Naval Surface Weapons Center, Sept. 1983.

[6] J. A. Clark and D. K. Pradhan, "REACT: A synthesis and evaluation tool for fault-tolerant multiprocessor architectures," in *Proceedings of the Annual Reliability and Maintainability Symposium*, 1993, pp. 428–435.

[7] A. K. Ghosh and B. W. Johnson, "System-level modeling in the ADEPT environment of a distributed computer system for real-time applications," in *Proceedings of the IEEE International Computer Performance and Dependability Symposium*, April 1995, pp. 194–203.

[8] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson, "Fault injection into VHDL models: The MEFISTO tool," in *Proceedings of the 24th International Symposium on Fault-Tolerant Computing*, June 1994, pp. 66–75.

[9] E. G. Ulrich and T. Baker, "The concurrent simulation of nearly identical digital networks," in *Proceedings of the 10th Design Automation Workshop*, June 1973, pp. 145–150.

[10] S. Seshu, "On an improved diagnosis program," *IEEE Trans. on Electronic Computers*, vol. EC-14, pp. 76–79, Feb. 1965.

[11] D. B. Armstrong, "A deductive method for simulating faults in logic circuits," *IEEE Trans. on Computers*, vol. C-21, pp. 424–428, May 1972.

[12] T. M. Niermann, W. T. Cheng, and J. H. Patel, "Proofs: A fast, memory-efficient sequential circuit fault simulator," *IEEE Trans. on Computer-Aided Design*, vol. 11, pp. 198–207, Feb. 1992.

[13] G. L. Ries, G. S. Choi, and R. K. Iyer, "Device-level transient fault modeling," in *Proceedings of the 24th International Symposium on Fault-Tolerant Computing*, June 1994, pp. 66–75.

[14] 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, pp. 60–74, Jan. 1997.

[15] K. K. Goswami and R. K. Iyer, "Simulation of software behavior under hardware faults," in *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing*, June 1993, pp. 218–227.

[16] G. Choi and R. K. Iyer, "FOCUS: An experimental environment for fault sensitivity analysis," *IEEE Trans. on Computers*, vol. 41, no. 12, pp. 1515–1526, 1992.

[17] F. Yang, "Simulation of faults causing analog behavior in digital circuits," Ph.D. dissertation, University of Illinois, Urbana, IL, 1992.

[18] G. L. Ries, "Transient fault modeling," Master's thesis, University of Illinois, Urbana, IL, 1995.

[19] J. D. Barnette, "Acceleration techniques for dependability simulation," Master's thesis, University of Illinois, Urbana, IL, 1994.

[20] 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. 15-1, pp. 29–36, Feb. 1995.

[21] 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. Scheduled to appear.

[22] M. Rimen, J. Ohlsson, and J. Torin, "On microprocessor error behavior modeling," in *Proceedings of the 24th International Symposium on Fault-Tolerant Computing*, June 1994, pp. 76–85.

[23] D. Stott, G. Ries, M. C. Hsueh, and R. K. Iyer, "Fault injection for high-speed network dependability analysis," *IEEE Transactions on Computers Special Issue on Dependable Computing*, 1997. Scheduled to appear.

[24] Z. Kalbarczyk, R. K. Iyer, G. Ries, J. U. Patel, and M. S. Lee, "Hierarchical approach to accurate fault modeling for system test and evaluation," in *Proceedings of the 3rd IEEE International On-line Testing Workshop*, 1997. Scheduled to appear.

VITA

Gregory Lawrence Ries was born in [redacted], on [redacted]. He graduated from Case Western Reserve University, in Cleveland, Ohio, with a B.S.E.E. in 1992, and in January, 1995, obtained a M.S.E.E. from the University of Illinios at Urbana-Champaign. His awards include National Science Foundation Scholar, Robert C. Byrd Scholar, Leonard Case Scholar, and CWRU Alumni Scholar.

Ries was a teaching assistant at the University of Illinois from August, 1992, until August of 1993, following which he was a research assistant in the Center for Reliable and High-Performance Computing until May, 1997. He also worked at Rockwell Semiconductor Systems as an engineer in the advanced DSP architecture group during the summer of 1996. Ries is a member of the Tau Beta Pi national honor society. He began working toward his Ph.D. in the spring of 1995 under Dr. Ravishankar K. Iyer at the University of Illinois at Urbana-Champaign in the area of dependability modeling.

HIERARCHICAL SIMULATION TO ASSESS HARDWARE AND SOFTWARE DEPENDABILITY

Gregory Lawrence Ries, Ph.D.
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, 1997
Ravishankar K. Iyer, Advisor

This thesis presents a method for conducting hierarchical simulations to assess system hardware and software dependability. The method is intended to model embedded microprocessor systems. A key contribution of the thesis is the idea of using fault dictionaries to propagate fault effects upward from the level of abstraction where a fault model is assumed to the system level where the ultimate impact of the fault is observed, and a second important contribution is the analysis of the software behavior under faults as well as the hardware behavior.

The simulation method is demonstrated and validated in four case studies that analyze a commercial, high-speed networking system called Myrinet. One key result from the case studies shows that the simulation method predicts the same fault impact 87.5% of the time, as is obtained by similar fault injections into a real Myrinet system. Reasons for the remaining discrepancy are examined in the thesis. A second key result shows the reduction in the number of simulations needed due to the fault dictionary method. In one case study, 500 faults were injected at the chip level, but only 255 propagated to the system level. Of these 255 faults, 110 shared identical fault dictionary entries at the system level and so did not need to be resimulated. The necessary number of system-level simulations was therefore reduced from 500 to 145. Finally, a third result in the case studies shows how the simulation method can be used to improve the dependability of the

target system. The simulation analysis was used to add recovery to the target software for the most common fault propagation mechanisms that would cause the software to hang. After the modification, the number of hangs was reduced by 60% for fault injections into the real system.