NASA-CR-195762

Febuary 1994

UILU-ENG-94-2203
CRHC-94-02

Center for Reliable and High-Performance Computing

DEPENDABILITY ANALYSIS OF PARALLEL SYSTEMS USING A SIMULATION-BASED APPROACH

Darren Charles Sawyer

Coordinated Science Laboratory
College of Engineering
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-94-2203 CRHC-94-02	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research, National Aeronautics Space Administration, and Tandem
6c. ADDRESS (City, State, and ZIP Code) 1308 W. Main St. Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code) 800 N. Quincy St. Arlington, VA 22217 Moffet Field, CA 95043 Cupertino, CA95014
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Joint Services Electronics Program 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER NASA NAG 1-613 N00014-91-J-1116 Tandem
8c. ADDRESS (City, State, and ZIP Code) 7a	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Dependability Analysis of Parallel Systems using a Simulation-Based Approach

12. PERSONAL AUTHOR(S) SAWYER, Darren Charles

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	94-02-02	68

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	fault injection, simulation, models, parallel systems

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The analysis of dependability in large, complex, parallel systems executing real applications or workloads is examined in this thesis. To effectively demonstrate the wide range of dependability problems that can be analyzed through simulation, the analysis of three case studies is presented. For each case, the organization of the simulation model used is outlined, and the results from simulated fault injection experiments are explained, showing the usefulness of this method in dependability modeling of large parallel systems. The simulation models are constructed using DEPEND and C++. Where possible, methods to increase dependability are derived from the experimental results. Another interesting facet of all three cases is the presence of some kind of workload of application executing in the simulation while faults are injected. This provides a completely new dimension to this type of study, not possible to model accurately with analytical approaches.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

DEPENDABILITY ANALYSIS OF PARALLEL SYSTEMS USING A SIMULATION-BASED APPROACH

BY

DARREN CHARLES SAWYER

B.S., Rensselaer Polytechnic Institute, 1992

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1994

Urbana, Illinois

ACKNOWLEDGEMENTS

I would like to express my deepest appreciation to my thesis advisor, Professor Ravi K. Iyer, for all of his assistance and guidance throughout this thesis work. I would also like to thank my peers at the Center for Reliable and High-Performance Computing for their help with an assortment of questions that I posed to them. In particular I would like to recognize Kumar "K-Master" Goswami for his invaluable advice, and Greg Ries who did much of the original development of the DASH model. Additionally, I would like to thank Joe Yarmus, Sue Maxwell, Dave Douglas, Andy Nanopoulos, and the others at Thinking Machines and NCSA who assisted in the RAID and CM-5 work. Finally, I would like to thank my family for all of the support and encouragement they gave me throughout my college experience.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

1.1 Motivation

As parallel systems grow in size and number of components, the possibility of a single failure having a system-wide impact greatly increases. Designers can no longer rely on the inherent component reliability to ensure acceptable system dependability. The systems themselves must be designed to remain operational in the presence of faults, while maintaining an acceptable level of performance. Given the "Grand Challenge" nature of applications running on these machines, we can expect that applications may execute over the course of several days or even weeks. Systems which fail often may never be able to complete these applications without the ability to continue execution when a fault occurs.

Several methods have been used to evaluate the behavior of large parallel systems for reliability and performance. Analysis of simple system reliability and performance can be performed with available analytical and queue-based modeling tools [1, 2, 3]. Such tools enable us to determine metrics including mean time between failures, system availability, and

reliability curves. One problem with these tools is that they suffer from state space explosion when the number of components becomes large, as is typically the case with systems which are massively parallel. Complicated advanced techniques and increased computing power can limit the effect of this problem. More importantly, however, these tools lack the ability to provide information about the behavior of the system while running real applications under various fault scenarios. The inability to model the behavioral aspects of a faulty system leaves many important design questions unanswered. Another approach to analyze a system under faults involves injection of faults into a real machine, and then measuring the performance of the wounded system. While this method is effective for smaller machines, massively parallel systems, because of their size and typically limited access, do not lend themselves well to this technique. Proposed here is the development of a functional simulation model as an alternative to these two approaches. Unlike analytical and probabilistic simulation models, functional simulation does not require that effect faults be specified by a predetermined set of probabilities and distributions. Instead, the actual behavior of the system is modeled, providing us with results which more accurately reflect what will really happen in a massively parallel system which suffers from faults. This thesis demonstrates how functional simulation models can be used to characterize system behavior under faults.

1.2 Related Work

Increasing work has been performed in the area of software and hardware fault injection of prototype systems [4, 5, 6, 7, 8]. However, few studies evaluate a system without the benefit of

some type of prototype, as is necessary in analyzing the system in the early design phase, where improvements can be made much more easily. Czeck [9] and Goswami [10] provide examples on how this type of dependability analysis can be performed using a simulation approach.

The simulation environment used in developing the models of these systems is DEPEND, a joint performability and dependability analysis tool that facilitates the modeling and analysis of fault-tolerant architectures at the system level [11, 12]. DEPEND provides a library of objects which simulate the functional behavior of components commonly used in fault-tolerant systems. These objects also inject faults, initiate repairs, compile fault statistics and generate detailed reports [13]. To model a system, the user writes a control program that declares instances of these objects, initializes them and coordinates their actions in a way that mimics the system being simulated. The bulk of the simulation work is performed by the DEPEND objects. DEPEND also allows the user control over the simulation engine: the user can abort and reschedule events. This feature is extremely useful for fault-based simulations where actions have to be aborted and rescheduled to properly model the effects of faults and errors in the system. The user writes a control program in C++ with the objects provided by DEPEND. Once it is written, the program is compiled and linked with the DEPEND objects and the run-time environment. Another important aspect of DEPEND is the ability to incorporate application execution into the simulation [14]. DEPEND has been used to successfully model a wide array of architectures. Models have been developed for the Tandem Integrity S2, Parasytic GC [15], Stanford DASH, and Connection Machine CM-5, among others.

1.3 Thesis Overview

The analysis of dependability in large, complex, parallel systems executing real applications or workloads is examined in this thesis. To effectively demonstrate the wide range of dependability problems that can be analyzed through simulation, the analysis of three case studies is presented. For each case, the organization of the simulation model used is outlined, and the results from simulated fault injection experiments are explained, showing the usefulness of this method in dependability modeling of large parallel systems. The simulation models are constructed using DEPEND and C++. Where possible, methods to increase dependability are derived from the experimental results. Another interesting facet of all three cases is the presence of some kind of workload of application executing in the simulation while faults are injected. This provides a completely new dimension to this type of study, not possible to model accurately with analytical approaches.

The first case study examines design trade-offs in a RAID-3 disk array system. Disk failures and write errors are injected into the array under realistic workload conditions. Several array configuration parameters are varied to determine their effect on the mean time to data loss in the array. The effectiveness of a scrubbing algorithm is also explored. Hybrid techniques for simulation acceleration are included in the model description.

A directory-based cache coherence protocol in a shared memory multiprocessor system is the focus of the next study. A simulation model of the Stanford DASH system is developed to facilitate this. The effect of faults in directory memory on the results of an executing parallel

application is determined. The results are used to determine a method to increase fault tolerance in the DASH system with respect to a wide class of faults.

The third and final case study centers on network failures in a massively parallel distributed memory supercomputer based on the Connection Machine CM-5 architecture. A simulation model is used to analyze several applications which exhibit different message passing characteristics. Assuming a rerouting strategy in a faulted network, a series of faults is injected in network components, and the associated performance degradation is quantified.

The thesis concludes with suggestions for possible future work in this area.

2. ANALYSIS OF A RAID-3 I/O SUBSYSTEM

In this chapter, we explore how RAID-3 disk arrays [16] react to disk failures and errors under realistic workload conditions. A simulation model is presented which models two fault scenarios common in the type of disks used in these arrays. The model allows a number of parameters to be varied, allowing the analysis of disk systems with various configurations. Of particular interest is the effect of error latency on the disk array mean time to data loss (MTTDL). MTTDL is the most important measure of the dependability of the disk array, since such RAID systems are designed to be secure from data loss caused by disk failures or write errors.

The chapter is divided into sections as follows. A brief description in Section 2.1 details the DEPEND model used in the analysis, explaining the objects used, their interactions, and the hybrid techniques used to accelerate the simulation process. Fault models and the possible failure scenarios which were modeled, as well as a description of the synthetic workload executed on the disk array are also included. Section 2.2 presents results for the sensitivity analysis performed using the model. The effect of a variety of system parameters on the

MTTDL of the array is quantified, including disk hardware fail rate, write error rate, disk array size, and disk capacity. Section 2.3 extends the analysis by determining the impact of scrubbing algorithms on reducing the time errors remain latent in the array. Section 2.4 provides a comparison of the simulation results with those of some simple analytic models. Section 2.5 concludes the chapter and directs future work.

2.1 Simulation Model

Developing a simulation model for an array of disks requires careful determination of what exactly is needed to obtain the desired results. Incorporating too many details of the system can result in large simulation times, especially for analysis of larger arrays. An initial version of this model, which contained detailed modeling of disk controllers and other facets of the real system resulted in an unacceptable 3:1 simulated time to real time ratio. Since typical RAID-3 disk arrays are designed with MTTDL requirements of over 1000 years, an efficient simulation model is essential. The model presented here achieved performance resulting in simulated to real time ratios of over 10,000,000:1 for some disk configurations. The ratio varies based on the system configuration, workload, and error injection rate.

It is also possible to model the disk system in too little detail, failing to capture important mechanisms which contribute to the MTTDL. Many RAID models, simulation or analytic, use simple fault models, such as general disk failures, which do not take into account more common and equally dangerous disk block write errors. The detection of these errors usually occurs when the corrupted block is read from disk, and thus becomes a function of the workload on

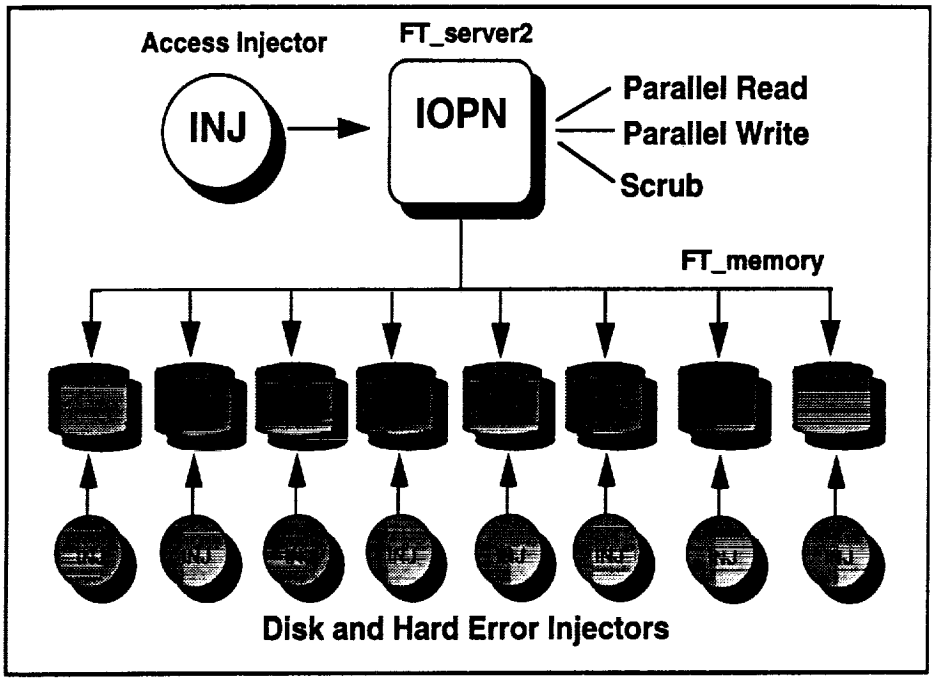

Figure 2.1: DEPEND model of RAID-3 disk array

the array. Modeling the patterns of reads and writes to the array should therefore be included in the model.

In this section, we examine those factors which have an impact on MTTDL, and explain how they are modeled in the simulations. This includes the disks, fault models and fault injection facilities, failure modes, and the workload. A general diagram of the model structure is provided in Figure 2.1. The incorporation of hybrid modeling, which makes use of analytic techniques to accelerate simulation time whenever possible, is also explained.

2.1.1 Modeling the disks

Modeling the disks in the RAID-3 system was achieved by creating a C++ object which represents the disk. The disk object is derived from the FT_server2 base class provided

by DEPEND. The FT_server2 base class provides facilities to monitor the state of the disk hardware, as well as automatic fault injection, repair, fault recording, and statistics reporting. When an access is to be performed to the disk, the hardware state method is called. If the disk is determined to be failed, the calling process is notified and recovery is invoked.

A disk in the disk array is given a command to read or write a block (or group of blocks), and the data are transferred to or from their proper location on the physical media. For the purposes of this model, a disk can be viewed as an array of blocks. A block can hold data, but since we are concerned only with whether or not the block of data is corrupted, only a single bit is needed to indicate the state of an entire block of data. This is sufficient since a block is the smallest unit of data addressable to the disk. Even with one bit per block, a 2.0 GB disk holds almost four million blocks of data, meaning 1.0 MB of real memory would be necessary to store the state of two disks in the array. Modeling arrays with hundreds of disks then becomes a problem for most computing environments.

A more efficient means of storing this state information requires storing a list of corrupted disk blocks for each disk. Each time a block on the disk is to be accessed, the list for that disk is traversed, and the value is compared with each element in the list. This may require more time than the bit field method, but the memory requirements are substantially less. Since the number of corrupted blocks on a disk is usually small (e.g., under five blocks), any block under consideration can be checked against the list in a relatively small amount of time. This method was implemented for the disks in the model.

The injection of faults into the disks are managed by two separate fault injection facilities. To model general hardware failures which make the disk unusable, the FT_server2 facilities are used to automatically inject errors. An exponential distribution is used, and the injection rate is accessible to the user of the model. Other distributions, provided in DEPEND, are also possible. When an injection occurs on the disk object, the state information is updated and the fault is recorded. It then becomes unusable to processes trying to read or write to the disk.

To model block write errors, a separate injector is used. Since the rate of write errors is given in terms of error events per bits accessed, injection becomes a function of the number of reads and writes occurring in the model. Given a normal distribution and standard deviation governing the bits accessed before a write error, the write error injector selects a number denoting the bit accessed when the error is to occur. Thus, when this number of bits is accessed, a write error occurs on the next block written. The number of blocks affected by the error can also be varied by altering a separate normal distribution, but typically only one block is affected by the error.

2.1.2 Modeling the RAID-3 mechanisms

Any number of these disk objects can be assembled into a RAID-3 disk array system model. A series of reads and writes are sent to each disk in the array using a *workload injector* object. This object generates lists of blocks to be read or written, and then requests the blocks sequentially from each disk. The same block number is sent to each disk in the array, as the RAID-3 striping method requires. In zero simulation time, each disk determines whether the

requested block is good or corrupted, or if the disk hardware has failed, and returns this status. When all disk objects return the result of the access attempt, the information is complied and examined to determine if recovery is necessary or data are lost. If no errors are returned, the simulation clock is advanced to represent the time necessary to perform the access in real time.

When one or more errors are present, it is necessary to check if data are lost. This occurs when either:

1. Two disks fail.

2. Two write errors on the same block occur on two separate disks.

3. A disk fails when any write error exists on any other disk.

With the discovery of the first error on a disk, an attempt to recover the unretrievable information using the parity disk is made. Since the storage of data is not simulated, we are concerned only with detecting a situation that would make this recovery fail. For block write errors, the corrupted block must be good on every other disk. Thus, only that block is checked for each disk, and no other corrupted blocks are discovered. Normal operation is then resumed. The time required for this type of recovery is assumed to be equal to the time of one access.

If an entire disk fails, the information on the entire disk must be recovered. This requires that all other disks in the array are defect free. If any error exists on any of these disks, or any are failed, data are lost and the simulation ends. In the case of a successful recovery, simulation time is advanced by a adjustable amount of time based on real system measurements. For the simulations used in this study, this time was 20 min.

Table 2.1: Synthetic workload characteristics

Characteristic	Distribution	Value
Interarrival Rate	exponential	$\lambda = 0.05$/sec
Location	uniform	over entire disk
Size	normal	mean=75 MB std_dev=25 MB
Type	uniform	75% reads / 25% writes

2.1.3 Modeling the workload

From the perspective of a disk, a workload consists of a series of reads and writes of various sizes accessing various areas of the disk. Reads and writes are generated in the model using the aforementioned workload injector, which is a specialized FT_injector object provided by DEPEND. Instead of injecting faults at specified intervals, the workload injector "injects" read and write requests to the simulated disk array, based on an interarrival rate. Random number generators are used to select the type of access (read or write), size of the access, and access location (start block on disk).

Since no studies have been performed to characterize a typical workload for a real RAID-3 system, we have used estimated numbers and distributions provided by engineers at Thinking Machines Inc., a manufacturer of RAID-3 disk arrays. In our simulations, disk accesses occur based on an exponential distribution with a mean of one access every 20.0 sec. Parallel reads comprise 75% of these accesses, with parallel writes making up the remaining 25%. The size of accesses are normally distributed with mean of 75 MB and standard deviation of 25 MB. Finally, access locations are uniformally distributed across the entire disk. Table 2.1 summarizes these parameters.

2.1.4 Hybrid acceleration techniques

Even at the level of detail described above, computation time required to simulate long periods of operation is prohibitive. This is due primarily to the heavy influence of extremely rare events (disk errors) on total simulated time required. The brute force method of simulating every access until system failure occurs must be replaced by an intelligent method that uses information inside the simulation engine to decrease the amount of real time actually simulated. This section describes how analytical methods are used to solve this problem.

Disk errors, either hardware or write related, occur with specified distributions. We can predetermine the time at which a disk hardware error is going to occur by sampling from an exponential distribution. Write errors are workload related, but only in that errors occur based on the number of bits accessed on the disk. Since we are using distributions to determine the rate and size of accesses, we can estimate the time at which a disk write error will occur. Thus, the occurrence of errors can be determined analytically. Error detection, which has a significant impact on MTTDL, however, is a function of the actual workload that is exercised after the error occurs. Detailed simulation is therefore not necessary when no errors are present, since it is useful only in determining error detection time. We use this information to develop a method which accelerates simulation time until the time of an error occurrence. The error is injected and detailed simulation is activated until either the error is discovered and recovered from, or data are lost. In the case of error recovery, detailed simulation ends, and the analytic methods are used to advance the simulation clock to the time of the next error.

2.2 Sensitivity Analysis of Model Parameters

Designers of RAID-3 disk arrays face the problem of balancing a number of design trade-offs to provide maximum data integrity for the user. Determining which parameters the system is most sensitive to is an important part of this process. With this knowledge, designers can concentrate their resources on the aspects of the system which have the highest impact on increasing dependability. Since hardware specifications for disks are often inaccurate estimations, designers can know which numbers should be scrutinized most carefully. In addition, decisions concerning whether a part of the system (e.g., type of disk drive) should be upgraded can be evaluated for proper cost benefit analysis.

In this section, we analyze how several parameters in our RAID-3 model affect the overall system MTTDL. These parameters include the rate of disk hardware failures and write errors, as well as the number of disks in the array and the capacity of the individual disks. All results presented have a 95% confidence interval that is less than 10% of the mean. This typically involved approximately 500 simulation runs.

2.2.1 Effect of disk hardware failures

Disk hardware mean time to failure (MTTF) is typically specified in terms of thousands of hours. Most modern drives provide rates of one failure per 500,000 h, up from 400,000 h one or two years ago. In this experiment, we analyze how this increase in MTTF, as well as further increases to 600,000 h, impacts disk array MTTDL.

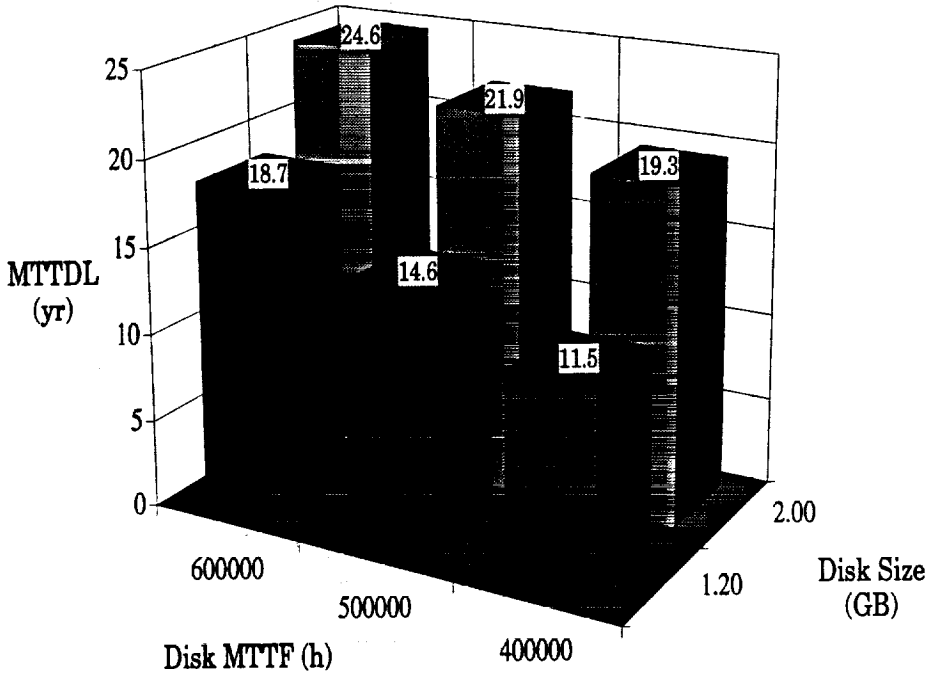

Figure 2.2: Effect of disk hardware failures on system MTTDL

Figure 2.2 presents MTTDL results for simulation of a 216 gigabyte total capacity configuration of disks having a 10^{14} bits accessed per write error event rate. The 216 GB are spread across either 108 2.0 GB disks or 180 1.2 GB disks, displayed as two sets of bars in the graph.

We can conclude that the MTTDL of the disk array is not very sensitive to the MTTF of the disks. While increased MTTF does increase MTTDL, the gains lie in the range of 10-15% which, depending on the costs involved, may or may not be worth an upgrade. However, the percentage increases are steady as MTTF increases, suggesting that MTTF has a direct impact on MTTDL. Examination of the common failure modes shows that the most common errors involved a failed disk when a latent error existed in another disk. Increasing MTTF decreases the probability of this occurrence, resulting in the increased MTTDL seen here.

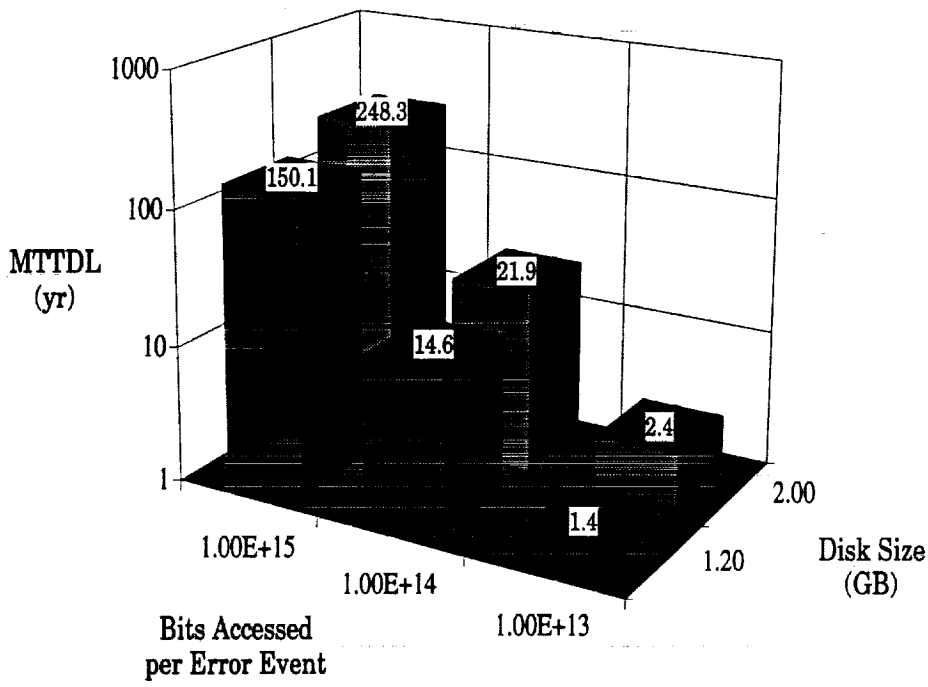

Figure 2.3: Effect of disk write error rate on system MTTDL

2.2.2 Effect of disk write error rate

Disk write error rates are specified in terms of *bits accessed per error event.* An error event involves the corruption of one or more blocks of data on the disk when the data are written. Typical drives provide rates of 10^{14} bits per error event. This is up from 10^{13} in the previous generation, and 10^{15} drives are expected in the near future. This experiment analyzes the effect of these increases on MTTDL of the system.

Figure 2.3 shows MTTDL results for simulation of a 216 GB total capacity configuration for disks with a 500,000 h MTTF. Note the results are plotted on a logarithmic scale. It is evident that a generational change in write error rate has a substantial impact on the data integrity of the disk array. This is not surprising, since every failure event simulated involved a write error (no two disk hardware failure scenarios were experiencèd).

Table 2.2: Percentage of failures caused by double-bit errors

Write Error Rate (bits per error event)	Double-bit Error Failures (% total failures)
10^{13}	11.2
10^{14}	8.4
10^{15}	0.9

Table 2.2 shows the percentage of failures caused by the existence of two write errors on the same block stripe. While the error rate of 10^{13} bits per error is very dangerous, an error rate of 10^{15} greatly diminishes this failure scenario. Since a generational change will increase MTTDL by a factor of 1000%, a designer would be best advised to use these disks as soon as possible.

The results also caution a designer to ensure the integrity of write error rates. Specifications are usually determined based on a small test lot of disks with accelerated failure techniques applied. If this leads to even slight inaccuracies in write error rate numbers, the observed impact on MTTDL of the disk array could be substantial.

2.2.3 Effect of parity group size

As supercomputers tackle problems of larger sizes, the need for disk space increases. One way to handle this problem is to produce disk arrays containing more disks. The experiments presented here show how disk array MTTDL is influenced by the number of disks in a parity group.

Figure 2.4 reports MTTDL for a variety of parity group sizes. Steady drops in MTTDL accompany increased disk size. Doubling the parity group size results in a 75% decrease in

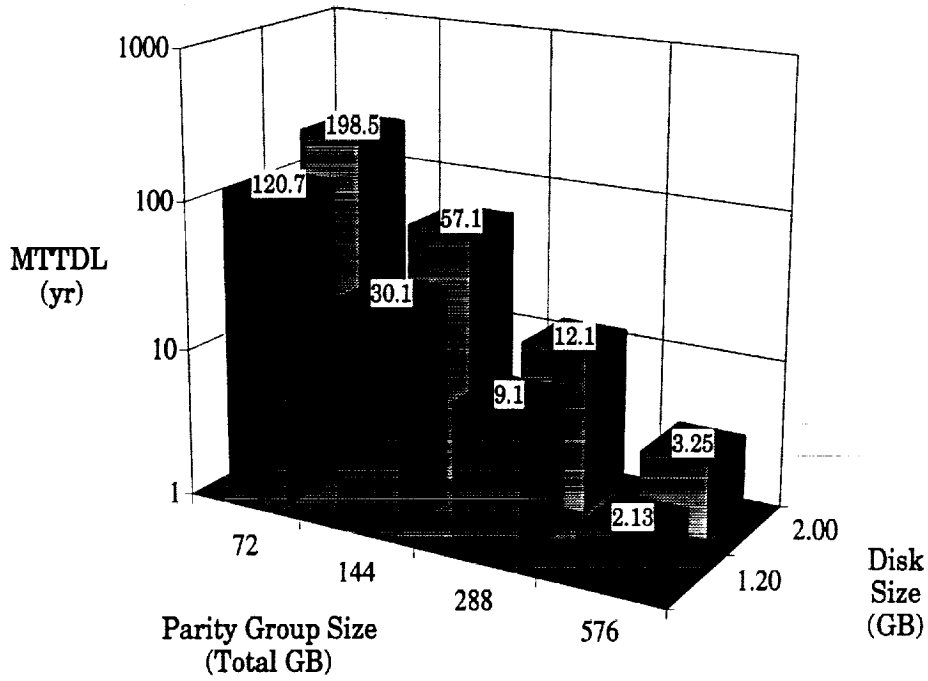

Figure 2.4: Effect of parity group size on system MTTDL

MTTDL. The primary reason for this lies in the fact that the same size disk access will not access as many blocks on a single disk, since there are more disks across a single block stripe. This suggests the need for multiple parity disks for larger arrays, where MTTDL has fallen below acceptable levels. This information is important for the designer to determine an appropriate breakpoint at which the costs of an extra parity disk are justified by tolerable MTTDL figures.

2.2.4 Effect of disk capacity

An alternative to increasing the number of disks is to increase the capacity of the individual disks. The results from the previous section showed the importance of keeping the number of disks small to achieve high MTTDL. In each of the figures corresponding to the previous experiments, the results for disks of 1.2 GB and 2.0 GB are displayed. Surprisingly, the full

Table 2.3: Comparison of error latencies for arrays with varied disk capacities

Array Capacity	Mean Error Latency (min)	
(Total GB)	1.2 GB disks	2.0 GB disks
128	1283	84933
256	2511	167345
512	269248	293442

increase expected based on the results from the smaller number of 2.0 GB disks is not achieved, but a somewhat smaller increase in MTTDL is observed. The answer lies in examination of the error latency times recorded in each case.

Table 2.3 contains mean error latency times for disk arrays with 1.2 and 2.0 GB disks. In all cases, the arrays with 2.0 GB disks show higher latency times. With more space on the disk to access, the chances of hitting an error are lower. The result is a decrease in the potential gains in MTTDL from the smaller array size.

2.3 Scrubbing Experiments

Periodic disk scrubbing is a method to reduce error latency by increasing the frequency with which data are read. A low-priority scrubbing process is activated when the disk has not been accessed for some period of time T_{wait}. When activated, the scrubber reads blocks of data from the disk solely for the purpose of detecting corrupted blocks. This provides earlier error detection than would otherwise occur, based on the normal workload executed. This section will examine whether scrubbing is worth the extra effort required to implement it in a real system.

In our model, a scrubbing process was added to determine the effect of periodic disk scrubbing on the disk array MTTDL. The type of scrubber implemented is a *sequential scrubber*, i.e., one which starts at block zero and reads blocks in order when the scrubber is activated, until the last block is checked. The process then starts over, checking at block zero. More elaborate scrubbers which track the workload to determine an optimal position to scrub are possible, but most real implementations will elect the sequential scrubber, since it requires the least overhead in program development time and on-line computational resources.

Figure 2.5 shows the results of simulations run with various scrubbing processes active compared to the same simulations without scrubbing. Results are presented for three different scrubbing rates, each with a different T_{wait} setting, as well as the no scrubbing case. The impact on arrays of three different sizes is also presented. It is apparent that the rate of scrubbing is an important parameter. High rates of scrubbing (T_{wait} = 60 sec) yield MTTDL increases of as much as 50 times the no scrubbing case. Increasing T_{wait} to 240 sec gives a rather low rate of scrubbing which does not have a significant impact on MTTDL. A moderate rate of scrubbing (T_{wait} = 120 sec) results in much higher MTTDL, possibly without as much overhead incurred by more frequent scrubbing. The designer of such a system can use this information to determine the optimal scrubbing rate based on the results from such simulations, balancing the predicted MTTDL gains with the overhead cost of higher frequency scrubbing.

Table 2.4 provides detailed data from this experiment. Mean error latency for write errors, the time between a write error occurrence and its detection, is presented for each configuration of array sizes and T_{wait}. Also included are the percentage of errors detected while scrubbing

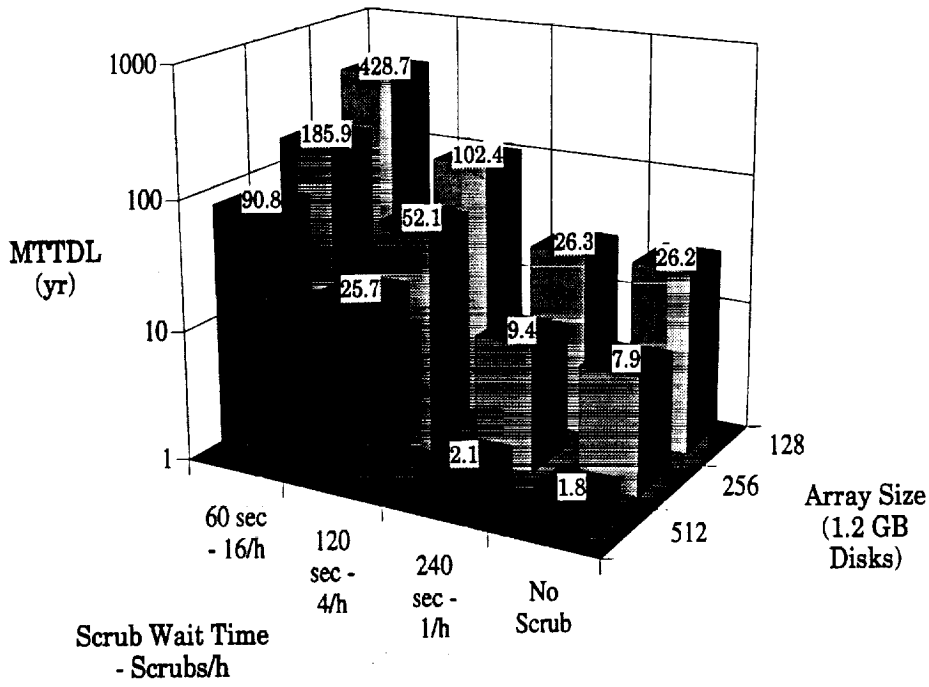

Figure 2.5: Effect of scrubbing interval on system MTTDL

and the percentage of simulations which ended with a data loss caused by the corruption of two blocks on the same block stripe. Error latency time, which has an inverse relationship with MTTDL, is dramatically reduced when scrubbing occurs frequently. The highest gains appear in the larger arrays, which are more vulnerable to failures. Scrubbing on these arrays brings mean error latency times down to lower levels of the smaller arrays. This is due to the fact that most errors are caught by the scrubber, not the workload, when the scrubbing rate is high.

Frequent scrubbing also appears to be most effective in reducing the frequency of the data loss scenario in which a disk drive fails with a write error on any other disk. A larger percentage of trials with the two-write error failure mode occurs at high scrub rates because of this.

Table 2.4: Experimental results from scrubbing experiments

Array Size (1.2 GB disks)	T_{wait} (sec)	MTTDL (h)	Mean Error Latency (min)	Detected by Scrubbing (% total err)	Double-bit Failures (% total sims)
128	60	428.69	68.15	94.83	19.10
	120	102.37	315.75	75.70	5.37
	240	26.31	1245.74	3.20	1.40
	no scrub	26.20	1283.08	0.00	1.00
256	60	185.89	67.67	97.40	13.07
	120	52.13	344.27	86.52	3.64
	240	9.44	2343.91	6.48	0.04
	no scrub	7.88	2511.10	0.00	0.01
512	60	90.84	65.62	98.73	12.93
	120	25.67	356.58	93.04	2.83
	240	2.12	3951.13	13.17	0.02
	no scrub	1.80	4487.48	0.00	0.01

2.4 Comparison with Analytic Models

Comparison of the simulation results with results from simple analytic models demonstrates the impact of accounting for workload based factors through simulation. Table 2.5 contains MTTDL figures for three different modeling approaches. The simplest of these models is a three-state Markov model presented in [17]. For an array of N disks, the model assumes independent disk failures, which are its only failure mode, occurring with a mean time to failure of $MTTF_{disk}$. Repairs are performed at a rate of $MTTR_{disk}$. The equation for MTTDL is given by

$$MTTDL_{i}ndep = \frac{MTTDL_{disk}^{2}}{N(N+1)MTTR_{disk}} \tag{2.1}$$

The results from this model overestimate MTTDL since it does not consider write errors.

Table 2.5: Comparison of simple analytic and simulation model results

Disk Array Configuration			MTTDL (yr)		
MTTF	Error Rate	Disks	Markov	Poisson	Simulated
500000	1.0E+14	60	3225.0	287.2	120.7
500000	1.0E+14	120	800.1	76.2	30.1
500000	1.0E+14	240	199.2	20.2	9.1
500000	1.0E+14	480	49.6	5.31	2.1
500000	1.0E+15	180	510.2	329.6	150.1
500000	1.0E+14	180	354.3	35.2	14.6
500000	1.0E+13	180	226.8	3.8	1.4
600000	1.0E+14	180	354.3	42.3	18.7
500000	1.0E+14	180	354.3	35.2	14.6
400000	1.0E+14	180	354.3	28.1	11.5

To provide a more even comparison, another analytic model was developed. This model contained probabilistic representations of workload and write error detection. These probabilities are computed using a Poisson distribution for each type of event that can occur in the system. These are used to determine the probability of a data loss at any instant, and thus, the mean time to data loss. We can see from the results that these numbers are closer to the simulated results, but there is still a significant difference caused by assuming probabilities that are difficult to accurately determine. As an aside, this model took only slightly less time to develop than the simulation model.

2.5 Future Work and Conclusions

Work must continue in the area of RAID analysis through simulation. Probably the greatest immediate need is the characterization of real workloads on real disk arrays. The workload used in this work is an estimation based on observations by engineers working on the development

of these arrays. Formal analysis of real workloads will provide information for setting up the workload injector in the simulation.

The effect of various scrubbing block selection policies on MTTDL should also be examined. For example, a least frequently read block policy might be effective in reducing error latency in blocks which are not read in the normal workload. Most recently written blocks (or blocks in their vicinity) might be checked to catch write errors quickly. A random scrubbing method might provide an equally effective low overhead solution, and should also be examined. Finally, preventive maintenance cycles, such as a total disk scrub every week, can also be included for realism.

Through this work, we have shown how simulation can be used to develop models which closely represent the realistic mechanisms of a system, such as workload, error detection, and scrubbers. The outcome of performing these simulations is more accurate results, in some cases, drastically different than those obtained analytically. The price for these results is time for model development and simulation execution. However, we have also shown that realistic models can be developed quickly, and techniques such as hybrid simulation for time acceleration can greatly reduce the overhead for execution.

3. ANALYSIS OF A DIRECTORY-BASED CACHE COHERENCE PROTOCOL

Often in the design of a complex system, determining the effect of faults on the operation of a system is a difficult task. Complex interactions of many components may make enumeration of all possible outcomes impossible to achieve. In this case, simulation provides an option to discover these outcomes, and possibly provide feedback on how problems can be recovered from or at least detected.

In this chapter, we will examine the behavior of a complicated directory-based cache coherence protocol under fault conditions. The goal here is to use simulation to determine how a real application is affected by faults. The Stanford DASH multiprocessor system [18, 19] is the architecture modeled and analyzed.

Section 3.1 provides a brief description of the key elements of the DASH architecture modeled. In Section 3.2, the structure of the DEPEND model used in the analysis is outlined. Section 3.3 presents the results from fault injection experiments involving faults in the directory controller. Section 3.4 uses these results to develop methods which improve the fault tolerance of the system. Section 3.5 concludes the chapter and suggests future extensions.

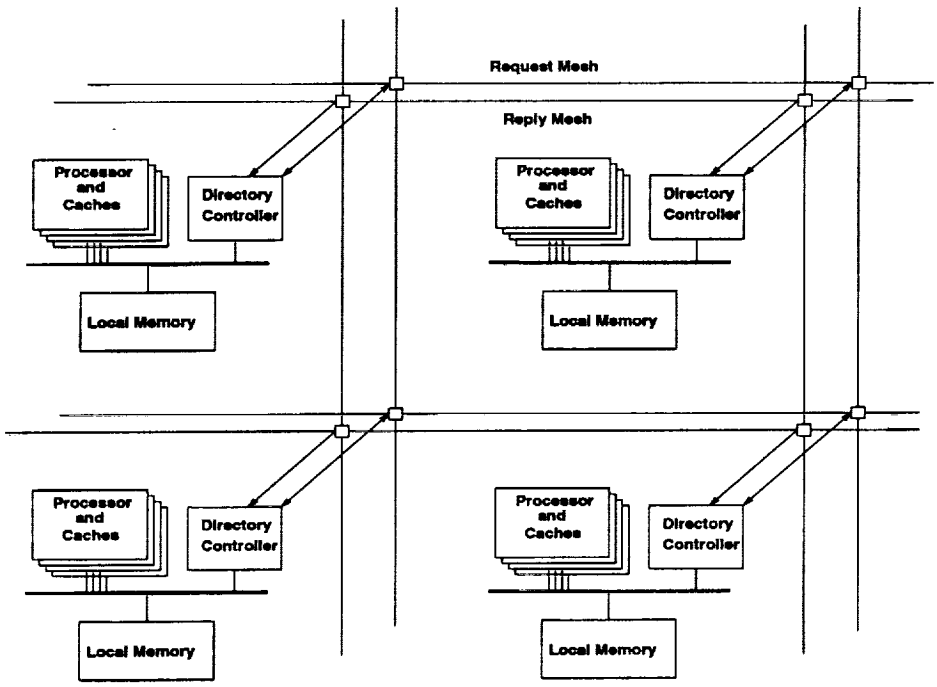

Figure 3.1: Architectural organization of the Stanford DASH

3.1 Stanford DASH

The DASH is a scalable shared-memory multiprocessor developed at Stanford's Computer Systems Laboratory. The architecture consists a number of processing clusters, each with a portion of the shared memory, connected by a scalable interconnection network, as shown in Figure 3.1. In the prototype, clusters consist of a commercial bus-based multiprocessor system. The system consists of four high-performance processors with two levels of direct mapped caches. Cache coherence is maintained within the cluster using snooping bus protocols.

Sharing of the memory locations located in a cluster with remote clusters is managed by a *directory controller* (DC). The DC responds to requests from external clusters for memory local to its cluster. It also manages requests from local processors for memory locations external to its

cluster. Coherence throughout the system is maintained by a message-based protocol existing between the cluster directory controllers. Messages containing coherence information, memory requests, and memory replies are passed using separate wormhole-routed mesh networks.

Of interest to this work is the structure of the directory memory. This memory is organized as an array of directory entries, one for each block of local cluster memory. Each entry consists of a bit vector, including one *state bit* used to indicate whether the block is shared with any of the remote clusters, or if the block is dirty in a remote cluster. A *cluster bit* for each cluster in the system denotes whether the cluster has a copy of the block in the given state. The DC then uses this information to determine where a valid copy of the data exists, and sends appropriate messages to fulfill the request. We will show how altering the information in this directory memory affects the results of a real algorithm executing on the system.

3.2 Simulation Model

Modeling the DASH to determine the effect of directory memory faults requires implementation of many of the details found in the real machine. This includes creating objects to model each of the basic components in the system, their functional behavior, and the protocols used among them. This section describes the level of detail modeled for each of these objects, as well as how a real application program is executed within the model. Figure 3.2 shows the object layout for the cluster model, including the DEPEND objects from which they were derived.

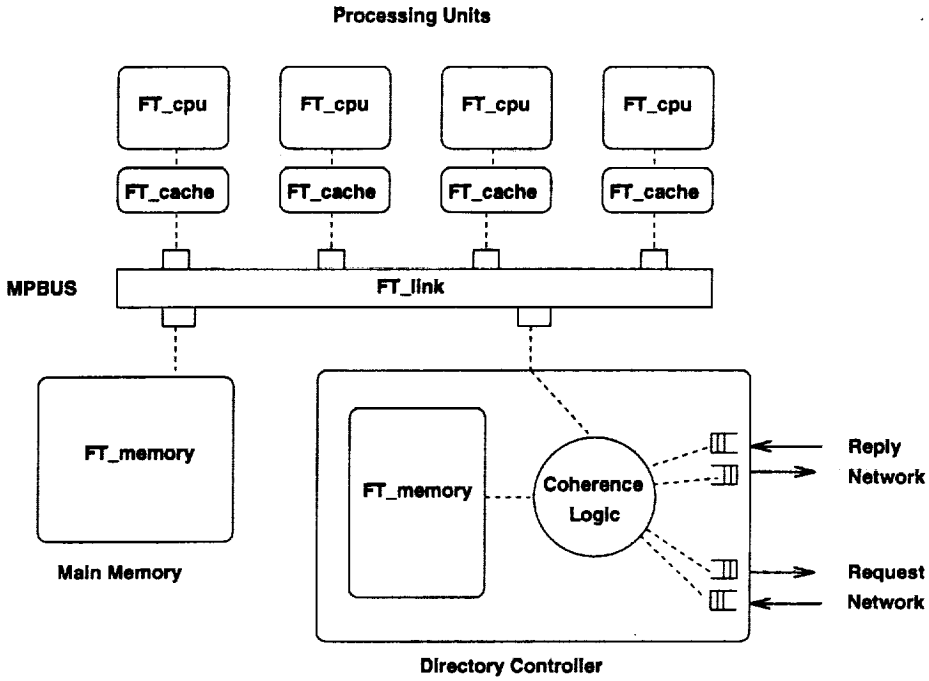

Figure 3.2: Object configuration of a DASH cluster model, with DEPEND base classes used

3.2.1 Modeling the memory

The memory actually stores each simulated word of data, either symbolically, or with an actual value. In addition, information about faults and corruption is stored with each word. The symbolic version is useful for simulating memory access patterns without implementing a real program, while the data storing version allows us to actually simulate a run of a simple parallel program, such as Gaussian elimination or QR factorization. Furthermore, information about a word of memory is saved only if that word is actually stored. This reduces memory required by the simulation.

3.2.2 Modeling processors and caches

The processor object is simply a process within the pseudoparallel simulation environment. The code within this process represents the code running on the real processor. The processor can therefore execute real parallel C code, and this feature was used to execute a sample application which executes during fault injections. Memory accesses within the program must be explicitly stated, and therefore, a type of memory management algorithm has been created to mimic that part of the operating system.

The cache object is derived from the memory object. In the simulations, only one cache level is modeled, and this is sufficient since the first-level cache is a subset of the second-level cache. Caches in the simulation are direct mapped as in the prototype machine. Each memory word of the cache contains a cache line and a tag, with address and state information stored in the tag. When a processor object requests a memory location, the cache will check the appropriate memory location, and if it is in the cache, return it after the appropriate amount of simulation type has elapsed, modeling cache latency. If it is not in memory, a request is made to the bus object which relays the request to other caches in the cluster and the directory controller. Inherited from the memory object is the same information on faults and corruption. The caches also implement a snoopy bus protocol to maintain consistency. Implementing this protocol required additional code which interacted with the other objects in the cluster via the bus object. Thus, the caches have a realistic effect, both in reducing latency and creating or propagating corruptions due to faults.

3.2.3 Modeling the directory controller

Each directory controller object also contains a memory object, this time to store the actual directory. State information and bit vectors associated with a particular memory location can be saved and, under a fault, altered. Here note the importance of the fact that only memory words used by the application are actually stored. This feature made feasible the allocation of space for 2M words for each main memory block, 256K words for each cache, and 512K words for each directory controller.

Implemented within this object are the internal algorithms used in the directory controller hardware. A single directory controller object spawns six active processes within the pseudo-parallel run-time environment. These processes are used to handle requests from the local bus and the network objects. Internal queues track outstanding requests and satisfy them when a response arrives from a connected source. The same coherence protocol used by the directory controllers in the real machine is established between directory controller objects in the simulation, and protocol messages are passed via the simulated network described below.

3.2.4 Modeling the network

The network is modeled as a completely separate object from each processing cluster. Thus, it is possible to use a completely different network, for instance, a hypercube instead of a mesh, in different simulations to examine the effect of the network type on the architecture. Links between clusters are modeled using the FT_link2 object provided by DEPEND, and custom router objects route messages through the network. Messages are passed as single packets

between clusters, and in the experiment performed, it was assumed that no corruptions would occur in these messages.

3.3 Fault Injection Simulation Experiments

Fault injection experiments were performed using the fault injection facilities provided in the simulation model. Since we are concerned with how faults affect the directory-based cache coherence protocol, injections were limited to faults occurring within a cluster directory while a typical application was executing. This section describes the fault model used, the application chosen to run on the simulated machine, and the results obtained.

3.3.1 Fault model

During execution of the sample application, a fault is injected randomly into a random cluster. The effect of the fault is to corrupt a particular word in the directory memory by altering one bit. The bit can be either that storing the state information for the associated main memory word, or one holding information about the caching of the word by a processor in a particular cluster. In either event, loss or alteration of the true information held about that main memory word will occur. The impact on the execution of the sample program is dependent on both the time and location of fault injection. To ensure that the fault did not remain dormant in an unused location, the fault was purposely injected in the next word to be accessed from the directory memory after the time of injection. An extension of this study could examine latency of errors injected in any word, but this issue is not addressed here.

3.3.2 Sample application - matrix multiply

For these experiments, one application was chosen to examine the impact of the directory faults. A parallel matrix multiplication algorithm which multiplies matrices in the form A·B = C is executed on the simulated processors. The matrices used were $256x256$ elements, although larger matrices could easily be used with a penalty of increased simulation time. The specific algorithm used was chosen to generate coherency messages among the clusters, and therefore is not the most efficient method available. The A matrix was distributed among the 64 processors, each writing 4 columns of the matrix. The B matrix, which is used by all processors, is distributed in a similar manner. Each processor must compute one row of the C matrix using its column of the A matrix and every entry of the B matrix. The pattern of memory access causes great demand for writing elements of the C matrix, fully exercising the intercluster cache coherence protocol.

3.3.3 Results

The fault injection experiments were performed for 500 fault scenarios. Three distinct symptoms appeared during the execution of the program. In one case, the fault had no observable impact on the results of the matrix computation. It is possible that the fault was injected on the final access of the main memory word whose entry was corrupted. Another scenario resulting in this case could have been a word in the shared state was corrupted by the addition of another cluster bit marked as having a copy of the main memory word. In this case,

an extra invalidation would be sent to the added cluster, and it would invalidate that word, even though it did not have it. Thus, no impact on the matrix computation would be observed.

Other faults caused a number of entries in the product matrix to be corrupted. These types of faults usually occurred because of a missed invalidation due to a corrupted cluster bit. The processor which had the word cached would therefore use the cached version, even though the word had been altered. These types of faults are potentially the most dangerous, since they produce altered directory entries which appear to be normal to the directory controller. Without some type of checksum information, these faults are not detectable.

Finally, certain bit combinations in the directory word cannot be interpreted by the directory controller. An example of this case is when the directory entry has the state bit as dirty, yet no cluster bit is set to indicate the owner. This could have been caused by either the state bit flipping from zero to one (shared to dirty), or by the cluster bit indicating the owner flipping from one to zero. In either case, the directory controller would not be able to make a decision about what should be done. It is difficult to predict how the real machine would react, but it is assumed that some type of checking to avoid continuance given this situation has been implemented in hardware. The next section details some mechanisms which could be implemented to recover from these types of errors, with as little degradation in performance as possible.

Table 3.1 summarizes the percentage of faults which fell into each of these three categories. The most common type of outcome was that in which no corruption occurred in the results of the computation. This is expected to be lower in most applications, especially when the data are shared by many processors and are frequently written. This is not the case for the

Table 3.1: Distribution of possible fault outcomes for Matrix Multiply

Fault outcome	*Occurrence*
No impact on computations	65%
Corrupted results	10%
DC detected fault	25%

algorithm chosen for this example. The second most frequent outcome was failure to complete due to directory controller detection of the error. The corrupted result matrix scenario was least frequent. This is encouraging since these faults can not be easily detected.

3.4 Adding Fault Tolerance to the DASH

The preceding analysis suggests that a certain degree of fault tolerance with respect to certain faults may be incorporated into the architecture. In this section, a method is presented which will make the DASH tolerant to a subset of the examined directory entry corruptions, with only a small degradation in performance over a very short period of time. This is accomplished by algorithmic means in the intercluster coherence protocol, rather than the usual methods of adding CRC checks or other hardware features. This method was chosen for two reasons; the algorithm change would be inexpensive in terms of performance and it would cover a majority of the faults.

The change to the algorithm can be described in two parts: changes to the home directory's actions and changes to remote actions. Focusing on the remote actions, whenever a directory receives a remote read request for a location it does not own, it puts that request on the bus. In the fault-free case, one of the processors will answer the request. However, in the faulty case,

it may be that none of the processors actually have the data. In this situation, the directory will then NAK the remote request. In addition, if the request was a read-exclusive, all the processors will also invalidate any copies they have of the line. Thus, there are two possible results to this action: one of the processors had the data and the data are being returned, or none of them had the data and a NAK is returned. In either case, if the request was exclusive, all processors invalidated their copy so the given cluster will not have any copies of that location.

The remote actions used a feature already in the algorithm to handle network latency, the NAK command, but the home actions are added. As long as a directory entry is valid, no unusual action is taken. Valid directory entries are those that are shared or those that are dirty with *one* owning cluster. A request is sent out only to the owning cluster, and when the data are returned, normal operation ensues. If the entry is invalid, however, or if a NAK is received, the home directory knows there has been an error. In that case it broadcasts the request to all clusters (except the requesting one). It then waits until it receives all replies. The replies fall into two different classes: all NAK replies, or one with data and the rest being NAK. In the first case, the directory knows there were no copies of the data remote, and it accesses memory, fetches the data, replies, and resets the directory state accordingly. If it receives a reply, it stores that reply in main memory, and sets the directory entry accordingly.

In examining the cost of this scheme, it is evident that as long as the machine is operating correctly, there is no additional cost to performance. There would also be only a slight increase in the directory control logic. The only cost that is incurred is that of the broadcast when an error is detected. Although a broadcast is significantly expensive, it occurs very infrequently.

The directory always corrects the error once it is detected, and therefore a broadcast happens only once per fault.

This same scheme could also handle permanent faults. The directory would just test the faulty location, once found. If it performed properly, a temporary fault would be assumed, and the algorithm would perform as before. However, if the location were shown to have a permanent fault, it could be mapped to a new memory location.

What kind of errors would this modification cover? Any error in a dirty entry, resulting in either changing the owner of the dirty location or making a dirty location shared. It would also cover some errors in a shared entry. There are two basic errors it could not cover. Any error that removes a cluster from the list of sharing clusters could cause incorrect data to be transferred because any shared entry is valid (nothing can be determined about a shared entry). Also, an error that changes a dirty entry to shared could cause data to be lost. Both cases occur because the home directory cannot tell if a request to a shared location should be broadcast. Any number of clusters are allowed to share a location, and thus there is no way to detect an error. In the first case, if a cluster is removed from the list, that cluster is never invalidated. If the data are changed and that cluster uses its own (incorrect) version again, incorrect data are read. In the second case, errors result when another cluster tries to acquire read-exclusive access. The home directory (thinking the data shared) invalidates everyone and passes the memory copy. However, the dirty copy is the correct one, so incorrect data are read. Finally, note that these are only possible errors. An error is never guaranteed, because incorrect copies may be invalidated or otherwise destroyed without being used.

3.4.1 Results of simulated fault tolerance

The new coherence algorithm was simulated on the same 16 cluster DASH model already discussed. This time, the same 500 simulations of the matrix multiplication were executed, but with the fault tolerant algorithms embedded into the directory controller object. The faults are still injected at the same time and location as in the nonfault tolerant experiments.

As expected, any simulation that ran correctly with no fault tolerance also ran correctly with the changed algorithm. In addition, there were no performance penalties to these routines. Also, all of the simulations that had failed to complete before (due to directory problems) also ran correctly this time. However, these suffered an average performance penalty of 18 cycles due to the broadcasting. Many broadcasts do not add to the latency, because the correct cluster was the one that would have taken the longest anyway. These two cases accounted for 90% of the simulation runs.

The cases that ran to completion but calculated incorrect results, however, continued to yield these incorrect results. The addition of fault tolerance made no difference to these routines. This case included 10% of the simulations.

3.4.2 Explanation of results

The two cases that ran correctly executed as expected. The cost of 18 cycles is probably not realistic, though, because on a heavily used machine, the real cost would be the network contention. For those cases in which an error still occurred, it was found that all of these cases involved errors to shared directory entries. Thus, they cannot be handled by the fault

tolerant protocol additions. Only some kind of CRC or other hardware would be effective. The resulting fault coverage of the algorithm was therefore 90%, an impressing reduction in the number of failures experienced by the algorithm.

3.5 Conclusions and Future Work

In these experiments, only one important fault susceptible area in the system was carefully analyzed. Such a complex system presents many other areas which may have adverse affects on the system when faults occur. One area is the local processor caches. Faults in the cache tags can potentially propagate invalid data throughout the system, much in the way faults in the directory cache can. Analysis of this type of fault is of particular interest. Fault effects can also be examined in the communication mechanisms between and within the clusters, such as corruption of messages or bus lines. Finally, additional corruptions within the directory controller, such as in the remote access cache, can be performed if implementation details are known. In all of these cases, it can be expected that a wealth of information regarding the effects of the faults can be obtained. This can then be used by designers to identify potential sources of fault tolerance, as has been shown with the example presented here.

In conclusion, a successful model was developed to determine the effect of single bit faults on a directory-based cache coherence protocol through simulating a 16-cluster DASH multiprocessor in the DEPEND simulation environment. Fault injection experiments in the directory controller memory resulted in three types of outcomes with respect to the computations being performed. The distribution of these outcomes was presented for a matrix multiply

application. A method for improving the fault tolerance of the DASH with respect to these faults was also presented, and shown to be successful in eliminating the effects of a high percentage of the fault scenarios.

4. IMPACT OF DEPENDABILITY ON PERFORMANCE IN MPP SYSTEMS

In this chapter, the relationship between performance and dependability in a massively parallel supercomputer is studied. A simulation model representing a Connection Machine CM-5-like system is developed. The model has the ability to execute real parallel C programs. Sample applications are used to show how a series of network link failures affects performance. This information could be used for determining the number of failures a system can tolerate before performance has degraded to a point where it is worthwhile to shut down the system and fix faulty components. Also presented is an example of how a simulation approach may reveal unexpected fault scenarios which result in improper operation. For example, the experiments revealed a deadlock condition due to an obscure fault pattern.

The chapter begins with a brief description of the Connection Machine CM-5, which serves as the base architecture for the simulation model used in the experiments. Section 4.2 outlines the structure of the model, including its ability to execute real parallel code and the fault injection process used in model components. Section 4.3 presents the performance results obtained for the network fault injection experiments, as well as a description of the deadlock

condition mentioned above. Section 4.4 summarizes the example and suggests possibilities for further study.

4.1 Connection Machine CM-5

The Connection Machine CM-5, developed by Thinking Machines Corporation of Cambridge, Massachusetts, is a Multiple Instruction Multiple Data (MIMD) supercomputer which utilizes a unique network architecture for interprocessor communication [20] . Each processing node in the CM-5 contains a 32 MHz SPARC processor, 32 MB of memory, and a 128 Mflop vector processing unit capable of processing 64-bit floating point and integer numbers [21] . These processing nodes are typically divided into partitions of various sizes, each with its own control processor to manage the partition's activity and I/O. Processing nodes are connected using two networks; the control and data networks. A third network, the Diagnostic Network, is used to detect, diagnose, and recover from hardware failures.

The data network is responsible for providing point-to-point communications between processing nodes. This is achieved using the "fat-tree" architecture introduced in [22] . The tree is organized with the processing nodes at the leaves of the tree, and special data routing chips at the internal nodes. The benefit of a fat-tree is that bandwidth is larger between nodes higher in the tree than that between lower nodes. This increases the bisection width and results in a significantly lower number of collisions between messages. A 16 processor fat-tree is shown in Figure 4.1.

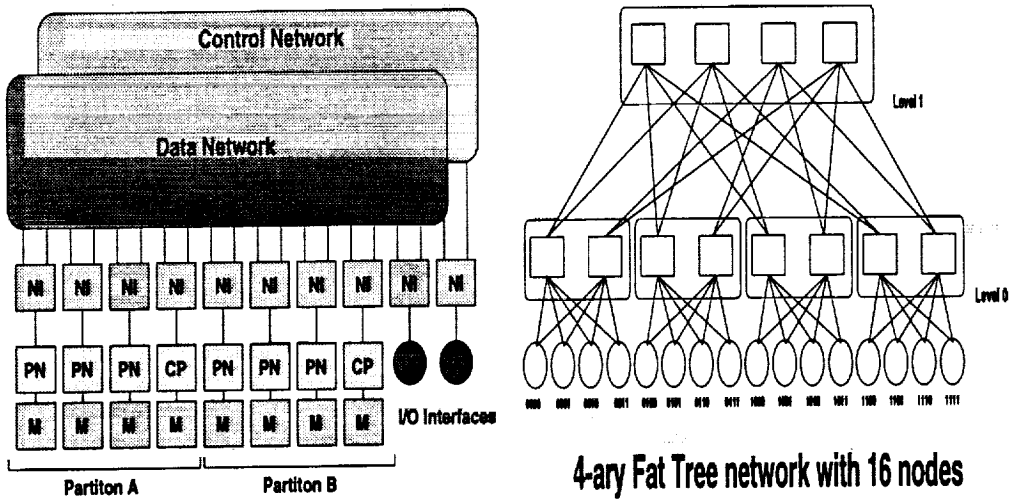

Figure 4.1: Connection Machine CM-5 system diagram and Fat-Tree network

The control network's primary function is to provide global broadcasts, scans, and reductions, as well as synchronization among processors. This network is a simple binary tree with the processors in the partition at the leaves. A processor broadcasts a message by sending it up the tree to the root. The message is then sent down each branch, eventually arriving at each processor. Reductions and scans are accomplished by having each processor send its value up the tree. At each tree node, some function (e.g., addition, comparison) is performed on the two values received from each child node. The sum of all these functions arrives at the root where it is then distributed to all nodes.

4.2 CM-5 Simulation Model

A model which accurately models the behavior of a real CM-5 system has been developed to analyze the impact of faults and reconfiguration strategies on system performance for several simple parallel algorithms. The model was developed in C++ using objects from the DEPEND

object library. In many cases, objects provided by DEPEND are customized to more accurately model the real CM-5 components. Each object in the model represents a subsystem component. For instance, there are individual objects for processing nodes, network interfaces, router chips in the networks, and the communication links connecting them. These objects are then connected together to form a complete CM-5 system.

Each component in the model may be injected with a fault which, depending on the type of component, causes the component to discontinue proper service either temporarily or permanently. For example, a processor fails to respond, or a link in the data network corrupts or loses messages. The fault models used for each component in our simulations are described in a following subsection. The model is also totally scalable. The user may select any size CM-5 system by specifying the number of processing nodes and its organization into partitions. The actual model is then constructed to these specifications. This allows a user to analyze the impact of certain faults on machines of various sizes. The model also allows the user to experiment with design changes in an effort to increase dependability. This includes methods to provide component redundancy and evaluate the effect of different sparing policies. Finally, real parallel C or C++ code which use the CMMD message-passing libraries [23] can be executed on the simulated CM-5. Thus we can observe how real applications will behave on the system when faults are introduced.

The following subsections describe in detail the organization of the model and the assumptions used in its development. In addition, a discussion of the fault models used and the execution of real code is included.

4.2.1 Model organization and fault models

In the development of our CM-5 model, we attempted to provide a direct mapping of each CM-5 component to an object in our model. These component objects were then encapsulated into objects which represent subsystems of the CM-5 architecture. These include the processing nodes, network interfaces, Data Network, and Control Network. Figure 4.2 shows the basic structure of the model and the objects it contains. Note that the Diagnostic Network was not included since it does not contribute to the operation of the machine from the user's perspective, which is of most interest in this study. We now describe how each of these four subsystems is modeled using modified DEPEND objects.

A Processing Node (PN) of the CM-5 is simulated with an FT_server object provided by DEPEND. Each PN object has the ability to actually execute real parallel code provided by the user. The user may also opt to simply model the computation time without actual execution. All PNs in a partition of the CM-5 are injected with faults by a single FT_injector object. This allows correlated faults between processors within a partition to be modeled and provides centralized control over all processing nodes in the partition. When a fault occurs, the processing node ejects any jobs in progress and ceases to respond to any requests from outside components.

Message passing is achieved by assigning a Network Interface (NI) object to each PN. An NI object is modeled using FT_server as a basis. A number of FIFO queues reside within the NI object, to queue incoming and outgoing messages. A PN object which wishes to send a message can communicate it to the NI, which may queue them and then send them to the appropriate network. Messages received from either network are queued until the PN decides

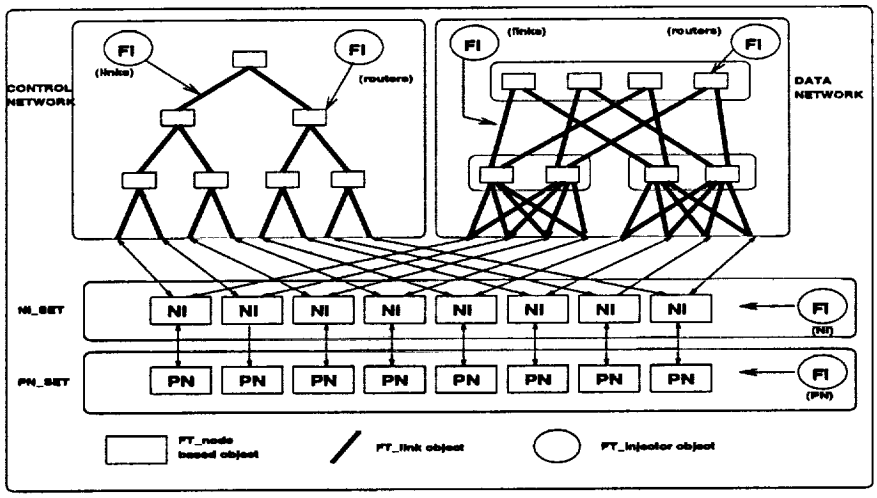

Figure 4.2: System-level diagram of CM-5 model created with DEPEND

to receive it. Like the real machine, each NI has two communication paths to the Data Network, and one to the Control Network. The NI tries to maintain a balance in the Data Network by randomly choosing which of the two paths a message is sent through. If a link or router failure in the Data Network causes one path to be blocked, the NI will send the message through the other path. As with the processing nodes, the network interfaces in a partition are injected with faults using a single FT_injector object. The fault model used in an NI is such that a fault will cause the object to cease responding to requests for the sending or receiving of messages from the PN. The NI also fails to accept messages coming from the network.

The Data Network of the CM-5 is modeled using a number of router chip objects which communicate messages via communication link objects modeled with the FT_link object from the DEPEND library. A Fat-Tree network of router chips (nodes in the tree) is constructed dynamically based on the number of processing nodes specified by the user, with network interfaces connected at each leaf. Each router chip is modeled using the general-purpose

functions of the FT_server along with logic to perform routing of messages travelling through the network. A decision of which path to send a message is made dynamically at each chip based on the message address. If the path of choice is not available due to a failure, a rerouting algorithm is invoked in an attempt to reach the destination via another route. Each router chip object is connected to four router chips lower in the tree, and two or four chips higher in the tree. The passing of messages between these chips is performed by sending the message through FT_link objects, which represent the wires in the actual network. Messages which collide in a router chip are buffered until the path is free for transmission. When a fault is injected in a router chip, the router no longer communicates with its links, and all messages buffered in the router are lost. Faults occurring in the link objects may have the effect of corrupting or losing message data for transient faults, or failing to accept messages if a permanent fault has occurred.

Finally, the Control Network of the CM-5 is modeled in a manner similar to that of the Data Network. A collection of router chip and link objects is connected in this network as a binary tree, with a network interface object at each leaf. In the Control Network, however, the logic in each router chip is altered to perform the special message functions to provide broadcasting, reduction, and combining operations, just as in the real machine. Single source messages (broadcasts) are sent up the tree from the NI which received the message from a processing node. At the root of the tree, the message is spread down all branches to each NI in the partition. Multiple source messages (reduction/combining) are queued at each router chip object as they come up the tree. A second message is waited for from the other child router,

and when it arrives, the proper reduction function is performed (e.g., addition, multiplication, logical operations). The fault models for the router chips and links in this network are similar to those of the Data Network.

4.2.2 Executing a real workload

An important feature of the CM-5 model is its ability to execute real parallel programs. This allows us to determine the effect of faults on specific applications, in terms of loss of performance or correctness of the results produced. For example, the failure of a router in the data network may invoke a specific routing algorithm which routes messages around the faulty area, causing the messages to take longer to reach their destination. The model provides a means to measure the overhead incurred by this strategy for various applications. Another case may be that a communication link in the network has become faulty and is occasionally corrupting message data. Here the model is useful in first checking the effectiveness of error detection (checksum) mechanisms in the communication hardware. We can then quantify the amount of damage undetected message corruption causes by examining results of the application.

The model executes application code by having DEPEND compile it as a method of each processing node object. The code can then be executed as a separate process by simply calling this method. The only additions to existing code are a simple "create" command, used by the simulation engine, and "use" commands needed by simulation engine to advance the simulation time clock after a block of code has been executed. By advancing the simulation clock, we

```
void CM_proc::execute()
{
  create ("test");                    // start process

  int token, n_procs, my_id;
  my_id = CMMD_self_address();       // get this node's address
  n_procs = CMMD_partition_size(); // find out how many nodes
  use(5.0);                          // use CPU for one second
  if (my_id==0) {                    // Proc 0 passes 1st token
    token = 0;
    use(2.0);
    CMMD_send(my_id+1,0,&token,sizeof(int));
                                     // send token to neighbor
    CMMD_receive(n_procs-1,0,&token,sizeof(int));
                                     // wait for token to return
  }
  else {
    CMMD_receive(my_id-1,0,&token,sizeof(int));
                                     // wait for token
    token += my_id;                  // add my id to token
    use(3.0);
    CMMD_send((my_id+1)%n_procs,0,&token,sizeof(int));
                                     // send to next one
  }
}
```

Figure 4.3: Sample parallel program executed on the CM-5 model

model the time a number of statements would actually take to execute. A simple parallel program is included in Figure 4.3 as an example.

This program sends a message around a logical ring of processors. It could be run on a real CM-5 as is. The only modifications we have made are the create command at the beginning and use commands interspersed throughout the program. This program uses CMMD message passing library functions to send messages through the data network of the CM-5. In the

real machine, these functions create messages which are sent through the network interface hardware to the data network. In the simulated CM-5, these functions have been altered to send the messages through the simulated network interface to the simulated network. There is no external difference between the actual function calls in either syntax or semantics.

The model can also handle several applications running at the same time. A separate process is created for each, and the processing node object maintains context switching between processes based on a selectable scheduling policy. This provides a means for a user to model a sample workload consisting of a mix of applications arriving at various times throughout the simulation run.

4.2.3 Fault injection process

Fault injection experiments may be constructed in many ways. The basic steps that are necessary in a typical experiment are outlined below.

1. The size of the system, number of partitions, and a workload (if desired) must first be specified. The model can then be constructed.

2. The distribution and rate of fault injection for each object is set. Injection may be either automatic by the fault injector objects or may be manual. A single method allows any object in the system to be manually injected at any time.

3. Rates of repair or special repair functions have to be set. Once again, repairs are done either automatically by the fault injectors or manually.

4. Specify any additional measurements to be made (DEPEND objects keep most of the needed statistics). Also, criteria for ending the experiment must be established (time, number of faults, system crash).

5. Execute the model with a selected random number seed.

Execution of the workload begins with computation and message passing just as in the real CM-5. Unfortunately, since one workstation is doing the work of n processing nodes, the actual execution time of the workload is much greater. However, experiments running simple workloads have provided insight into the behavior of a CM-5-like system under faults. We present some of the more interesting results in the following section.

4.3 Experimental Results

A number of experiments were performed using the CM-5 simulation model. This section focuses on the results from one particular class, failures in the data network links. The results presented demonstrate the impact of faults on performance of massively parallel systems, as well as the need to simulate a system in the design phase to uncover overlooked problems that result from a combination of faults.

4.3.1 Effect of link faults on Data Network performance

The discovery of a faulty link in the Data Network requires the rerouting of messages through a properly functioning link. The obvious effect of rerouting is some loss of bandwidth and higher network congestion around the fault. It is expected that an application which sends

messages which normally traverse the faulty link will suffer some loss in performance due to the rerouting procedure. If this link failure cannot be repaired within the interval before another link fails (either because the machine cannot be brought down for repair or because the next fault occurs a short time later), the performance of the application may degrade further. Since typical maintenance periods are about one week for large machines, it is likely that several failed links may coexist for a period of time. The question for the operator is whether the system performance is so degraded that the system should be brought down for immediate maintenance, or whether the unplanned outage is too costly in terms of the amount of computation time lost. This problem is addressed by a series of experiments performed on the Data Network of our CM-5 model. While running a particular application, link faults were introduced into the system one at a time. After each injection, the times for the communication portions of the applications were measured to observe the effect of the faults on the performance. To determine which levels of links in the network have the greatest impact when they fail, injections were performed at each level of a 64-processor system.

Figure 4.4 shows the resulting percentage increase in communication time for each application for a number of faults injected in certain levels of the network. For each experiment, faults were injected at the target level until the network was no longer able to deliver the message to its destination.

Figure 4.4(a) contains the results for link fault injections while running the Total Exchange algorithm. In this algorithm, each processing node collects a value from every other processor one at a time. The largest number of collisions occurs at the routers right above the destination

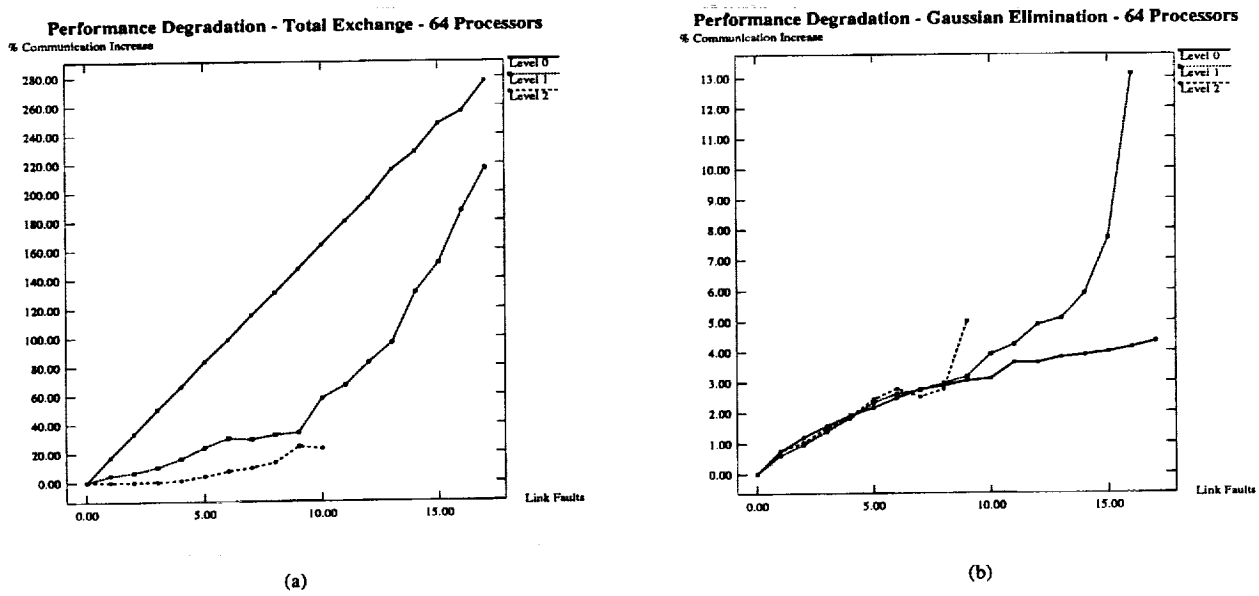

Figure 4.4: Effect of consecutive link faults on algorithm performance

node. Therefore, we expect communication time to be affected most by the loss of a link at the lowest level of the network, where it is needed most. We see that our simulations showed a nearly 20% increase in communication time for each fault at this level. Since congestion is less and the bandwidth is higher at the higher levels in the network, it is expected that fewer messages would be affected by link faults there. Our results verify this assumption, and provide quantitative information about the communication time increase.

Figure 4.4(b) shows the communication time increase for loss of links while running a Gaussian elimination program. Since this version of the program uses a *scattered* method of data distribution, we can expect to see sets of processors exchanging rows and columns of the matrix [24]. The broadcast of the pivot value can be handled by the Control Network so that that section of the algorithm is not affected by the link faults injected. The typical communication pattern for each iteration will involve disjoint sets of 8 processors. One processor from each

group communicates its row or column values to each member in its set. Since this method balances the load on the processors and on the Data Network, it is expected that a message could be rerouted around a link fault anywhere in the network without causing excessive collisions. Figure 4.4(b) shows this to be the case. With even seven or eight faults, little performance loss occurs (0-3%). Since the higher levels of the network are utilized frequently with algorithms, a high number of faults eventually begin to have an impact, as shown by the sudden rises in the level 2 and level 3 curves.

An ideal follow-up experiment would be to generate a realistic sample workload consisting of typically used programs, and repeat these experiments to gain an overall view of how this type of network performs under these faults. The experiments presented here stress the fact that some applications may not be affected at all by network faults, while others suffer greatly. In addition, the location of the fault is also shown to be of importance when considering these effects.

4.3.2 Deadlock due to link fault scenarios

When routing in a network is performed locally at each router, rerouting of messages may be performed by the router when it discovers or is informed of the failure of a connected link. The router chooses an alternative link through which the message may reach its destination. The routing algorithm used to achieve this is dependent on the network topology and the existing faults surrounding the router. Usually it is not difficult to devise such an algorithm which will successfully bypass the faulty link. However, when a number of links in the network have

suffered failures, a local routing scheme may result in deadlock if knowledge of failed links not connected to the router is not made available. During our link fault experiments, it was discovered that certain combinations of faulty links resulted in a deadlock situation, where messages would bounce between routers without ever reaching their destinations. The routing algorithm and fault scenario which triggered the deadlock are explained here. It should be clearly noted that this type of rerouting strategy *is not* the same one implemented in the real CM-5 machine. Implementation of the messages in the real machine does not allow this type of rerouting to exist, and therefore the deadlock problems described will not occur in reality. However, for general network problems, this strategy is not unreasonable, and the deadlock problems described below could exist in this type of system.

When a router determines that the message should be sent through a link which has failed, it must choose an alternative. In the Data Network, a message that should go up the tree can be sent to any other link and reach its destination in the same amount of time. Messages which are travelling down the network can be sent down only one path to the destination without taking a longer detour. If this path is blocked, a logical choice would be to accept this penalty by sending the message down another link.

Figure 4.5 shows a potential link fault scenario in which a rerouting algorithm, which is successful for most faults, fails. A message located in the router labeled START is trying to reach the processing node (or network interface) labeled DEST (node 11). Three links, labeled 1, 2, and 3, in the level separating the routers from the nodes have failed, as indicated by the X marks. Since the links are close in proximity, it is likely that they all could have been damaged

by a clumsy operator who scratched the network board the lines resided on, causing an open circuit for those connections. The first choice of the router would be to send the message along link 1 directly to the destination. When the router discovers (possibly after several retries) that it cannot send the message through link 1, it will select an alternate. The algorithm used simply chooses the first available link to the right of the failed link (wrapping around to the leftmost link if the rightmost link fails, as is the case here). In this example, the message would be sent to node 8. This node would realize that the message was not intended for it, and would try to send it up the network via an upward link other than the one it originated from. This is where the first problem occurs. The other upward link is link 3, which is failed. Node 8 has no alternative other than sending the message back up the link it came from, in hopes that another path can be found. When router START receives the message for the second time, it may try to send the message down link 1 again, since it does not know that this is not a message originating from node 8 unless some sort of costly state information is kept about each message which passes through the router. Since the message cannot be sent down link 1, the naive algorithm would send it back down to node 8. If we restrict messages from being sent back down their paths of origin, this easily solves this problem. If we skip that path and choose the next one, the message will be sent to node 9. Node 9 cannot send the message up its other upward link (link 2) because it has failed. It too sends it back to router START. Unfortunately, the router cannot send the message down link 1 as it wishes again, so it chooses to send it down to node 8. It does not know that some time ago it already tried this path, and to store this information is unreasonable. Thus, the whole cycle will start again and the message will never be delivered.

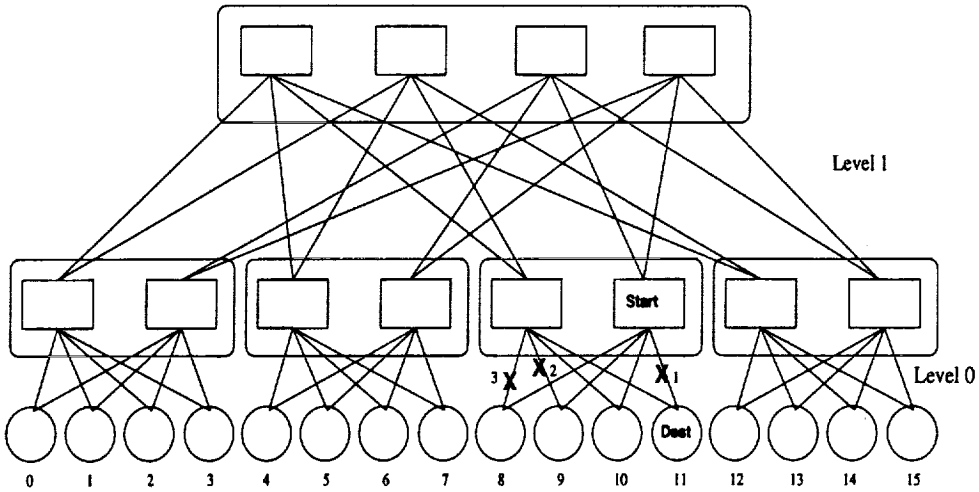

Figure 4.5: Link fault scenario resulting in deadlock

The unfortunate reality here is that a path *does* exist, through node 10. The method of selection chosen, however, results in deadlock when these faults exist.

The discovery of deadlock under multiple link faults demonstrates the need for simulation in the design phase of massively parallel systems. As systems become more complex, the difficulty in enumerating all possible fault scenarios increases. If the algorithms implemented in the hardware are susceptible to a particular set of faults which are not considered, the resulting effects may be disastrous. Proper simulation will reveal such cases before the hardware is put into production.

4.4 Conclusions and Future Work

In this section, it has been shown how dependability of a system affects its performance. The link failure example demonstrates the effect of rerouting strategies in the data network, and provides insight as to when an operator should decide to bring down a crippled machine for

repairs. Furthermore, the experiments showed an obscure shortcoming in the rerouting strategy used when a certain pattern of link faults exists. An experiment like this performed in the early design phase could prevent costly problems later.

Obviously, this class of link faults represents only a small part of the total possible faults in the system. The effects of router and processor failures may also produce interesting results. The question of efficient software execution is also important, since only small algorithms running for a small period of time are possible. Simulating systems larger than 64 processors tends to be troublesome as well, due to the large computing resources required. Parallel simulation methods, which map the model of the system across the nodes of a parallel machine, could help alleviate this. Initial tests indicate that models of this type are highly parallel, especially when executing code that does not involve a high degree of message passing.

5. CONCLUSIONS

5.1 Summary

The three significantly different case studies presented in this thesis provide strong evidence of the useful results that can be obtained using the simulation facilities provided by DEPEND. By modeling the actual algorithms and protocols governing these systems, as well as realistic workloads executing on them, we can obtain insight into the system behavior under fault conditions. This is especially important when the system is too large and complex to allow for enumeration of all possible faulty outcomes. Typically this information can be used to develop design modifications to improve fault tolerance, as is evident in the DASH example.

Also demonstrated in this work are methods of creating proper simulation models for the problem at hand. Modeling a system at the right level of detail and with a clear objective at the start of the modeling process proved to be the most effective method. While the simulation approach may take a fair amount of development and execution time, the benefits in terms of

increased possibilities for analysis and accuracy of results should be clear from the case studies presented in this work.

5.2 Future Work

Future work in the analysis of the systems modeled has been described at the end of each chapter. In this section, a number of suggestions are made for further study in the area of dependability analysis through a simulation model approach.

A major area in need of exploration is software fault analysis. With increasing component reliabilities, a greater percentage of critical failures occur in the system software. This work did not examine faults in the software. Large amounts of real software can not be included in the actual simulation due to simulation time constraints. Therefore, some type of abstraction capturing the behavior of the software under faults, without executing it, is necessary to do this analysis. Performing this abstraction is not a simple task, however, and must be the subject of further study.

Another interesting problem that occurred in the model development stage involved specifying the relationships among the various components in the system. One example of this results from the physical connections between components. In real machines, architectural components exist on the same circuit board, and this board is the minimal unit of repair. Such a *field replaceable unit* or FRU, as they are commonly called, has the characteristic that if one chip or subunit fails, all components on the FRU are replaced, not just the failed unit. This relationship must be specified and managed within the dependability simulation package.

Other dependency relationships include those centering on power or connectivity. A group of components will not function properly unless their power source is available. When the power source fails, all dependent components must be inactivated. DEPEND does not address this class of intercomponent dependencies, and a way to remove the burden of incorporating this into the simulation must be removed from the user, once specified.

A final class of problems focuses on reducing simulation execution time. Large systems, such as those analyzed in this work, are requiring more and more computing resources to generate accurate models. It was shown in the RAID example how simple acceleration techniques can drastically improve performance of the simulation. Traditionally, however, these techniques are incorporated on a case by case basis. Formal techniques which will allow automation of this process must be researched. One possibility may lie in hierarchical modeling. If a dependability model can be specified in various levels of detail, slow executing detailed component models can be used only at critical times in the simulation, while higher-level models can be used the majority of the time. Also, if information from lower level models can be extracted to higher-level models, precious execution time can be saved. Without further work in this area, it is unlikely that this type of modeling will be feasible for larger machines developed in the future.

REFERENCES

[1] SES, Inc., Austin, TX, *SES/Sim Simulation Language Reference Manual*, Mar. 1989.

[2] M. H. MacDougall and J. McAlpine, "Computer simulation with ASPOL," in *Symposium on the Simulation of Computer Systems, ACM/SIGSIM*, pp. 93–103, 1973.

[3] C. Sauer, E. MacNair, and J. Kurose, "RESQ: CMS user's guide," Tech. Rep. RA-139, IBM T.J. Watson Research Center, Yorktown Heights, N.Y., Apr. 1982.

[4] J. Lala, "Fault detection isolation and reconfiguration in FTMP: Methods and experimental results," in *5th AIAA/IEEE Digital Avionics Systems Conference*, pp. 21.3.1–21.3.9, 1983.

[5] K. G. Shin and T. Lin, "Modeling and measurement of error propagation in a multimodule computing system," *IEEE Transactions on Computers*, vol. 37, pp. 1053–1066, Sept. 1988.

[6] L. Young, R. K. Iyer, K. K. Goswami, and C. Alonso, "A hybrid monitor assisted fault injection environment," in *Third IFIP Conference on Dependable Computing for Critical Applications*, Sept. 1992.

[7] Z. Segall et al., "FIAT - fault injection based automated testing environment," in *The Eighteenth Annual International Symposium on Fault-Tolerant Computing*, pp. 102–107, June 1988.

[8] G. Kanawati, N. Kanawati, and J. Abraham, "FERRARI: A fault and error automatic real-time injector," in *The Twenty-Second Annual International Symposium on Fault-Tolerant Computing*, July 1992.

[9] E. W. Czeck, "On the prediction of fault behavior based on workload," Ph.D. dissertation, Carnegie Mellon University, Department of Electrical Engineering, Apr. 1991.

[10] K. K. Goswami and R. K. Iyer, "A simulation-based study of a triple modular redundant system using depend," in *5th International Tests, Diagnosis, Fault Treatment Conference*, Sept. 1991.

[11] K. K. Goswami and R. K. Iyer, "DEPEND: A simulation-based environment for system level dependability analysis," Tech. Rep. CRHC Report #92-11, CRHC, University of Illinois, June 1992.

[12] K. K. Goswami and R. K. Iyer, "DEPEND: A design environment for prediction and evaluation of system dependability," in *9th Digital Avionics Systems Conference*, pp. 87–92, Oct. 1990.

[13] K. K. Goswami and R. K. Iyer, *The DEPEND Reference Manual.* University of Illinois, Center for Reliable and High Performance Computing, Urbana, Illinois 61801, Oct. 1990.

[14] K. K. Goswami and R. K. Iyer, "Simulation of software behavior under hardware faults," in *The Twenty-Thrid Annual International Symposium on Fault-Tolerant Computing*, pp. 218–227, June 1993.

[15] Parsytec Computer GmbH, *Parsytec GC Technical Summary*, Jan. 1991.

[16] R. H. Katz, G. A. Gibson, and D. A. Patterson, "Disk system architectures for high performance computing," *Proceedings of the IEEE*, vol. 77, no. 12, pp. 1842–1858, Dec. 1989.

[17] G. A. Gibson and D. A. Patterson, "Designing disk arrays for high data reliability," *Journal of Parallel and Distributed Computing*, vol. 17, pp. 4–27, Jan. 1993.

[18] D. Lenoski et al., "The directory-based cache coherence protocol for the DASH multiprocessor," in *Proceedings of the 17th International Symposium on Computer Architecture*, 1990.

[19] D. Lenoski et al., "The DASH prototype: Logic overhead and performance," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, pp. 41–61, Jan. 1993.

[20] Thinking Machines Corporation, *The Connection Machine CM-5 Technical Summary*, Jan. 1992.

[21] C. E. Leiserson et al., "The network architecture of the Connection Machine CM-5," *ACM Symposium on Parallel Algorithms and Architectures*, 1992.

[22] C. E. Leiserson, "Fat-trees: Universal networks for hardware-efficient supercomputing," *IEEE Transactions on Computers*, vol. C-34, pp. 892–901, Oct. 1985.

[23] Thinking Machines Corporation, *CMMD 3.0 Reference Manual*, May 1993.

[24] R. P. Brent, "Parallel algorithms in linear algebra," Tech. Rep. TR-CS-91-06, Computer Sciences Laboratory, Australian National University, Aug. 1991.