NASA-CR-192924

Vol. 2/2

NAG-1-613
NCC2-481

COORDINATED SCIENCE LABORATORY

Computer Systems Group
University of Illinois at Urbana-Champaign

Presentation Slides
and
Publications

NASA Review of ICLASS:

Illinois Computer Laboratory
for
Aerospace Systems and Software

November 17, 1987

(NASA-CR-192924) PRESENTATION SLIDES AND PUBLICATIONS. NASA REVIEW OF ICLASS: ILLINOIS COMPUTER LABORATORY FOR AEROSPACE SYSTEMS AND SOFTWARE (Illinois Univ.) 520 p

N94-70418
--THRU--
N94-70427
Unclas

Z9/61 0158810

CONTENTS

CSG Presentation Slides

1. **Overview of ICLASS Research: Reliable and Parallel Computing.**
   Professor Ravi K. Iyer

2. **Reduction in Main Memory Traffic Through the Efficient Use of Local Memory**
   Geoffrey D. McNiven and Professor Edward. S. Davidson

3. **Parallel Algorithms for Design Automation**
   Professor Prith Banerjee and Randall Brouwer

4. **Parallel Architectures for Image Understanding**
   Professor Janak H. Patel and Alok N. Choudhary

5. **Efficient Reordering of Prolog Programs**
   Professor Benjamin W. Wah and Markian M. Gooley

6. **Concurrent Microprocessor Design and Implementation**
   Professor Wen-mei Hwu

7. **Experimental Methods for Performance/Reliability Evaluation**
   Professor Ravi K. Iyer and M. Devarakonda, K. Thearling, R. Llames

8. **Recovery from Failures in Parallel Architectures**
   Professor W. Kent Fuchs and A. Gupta, P. Chen, J. Li, K.L. Wu

9. **Performability Modeling Based on Real Data: A Case Study**
   Professor M.C. Hsueh

10. **Fault-Tolerant Matrix Computations on Processor Arrays**
    Professor J.A. Abraham and V.S.S. Nair

CONTENTS

CSG Publications

1. **Accurate Low Cost Methods for Performance Evaluation of Cache Memory Systems**
   Subhasis Laha, Janak H. Patel and Ravishankar K. Iyer

2. **Performance of a Parallel Algorithm for Standard Cell Placement on the Intel Hypercube**
   Mark Jones and Prithviraj Banerjee

3. **An Improved Simulated Annealing Algorithm for Standard Cell Placement**
   Mark H. Jones and Prithviraj Banerjee

4. **The "Rid-Redundant" Procedure in C-Prolog**
   Huo-yan Chen and Benjamin W. Wah

5. **Efficient Reordering of Prolog Programs**
   Markian M. Gooley and Benjamin W. Wah

6. **Resource Sharing on CSMA/CD Networks in the Presence of Noise**
   Duane Edward Dinschel (Masters Thesis)

7. **Applicative Architectures for Fault-Tolerant Multiprocessors**
   Madhumitra Sharma and W. Kent Fuchs

8. **A Measurement-Based Performability Model for a Multiprocessor System**
   M.C. Hsueh and R.K. Iyer and K.S. Trivedi

9. **A Measurement-Based Model of Software Reliability in a Production Environment**
   M.C. Hsueh and R.K. Iyer

10. **Recognition of Error Symptoms in Large Systems**
    R.K. Iyer, L.T. Young and V. Sridhar

11. **Reconfigurable Tree Architectures Using Subtree Oriented Fault Tolerance**
    Matthew B. Lowrie and W. Kent Fuchs

12. **Local Concurrent Error Detection and Correction in Data Structures Using Virtual Backpointers**
    C.C. Li, P.P. Chen, W.K. Fuchs

13. **Measurement-Based Reliability/Performability Models**
    Mei-Chen Hsueh (Ph.D Thesis)

14. **Design and Evaluation of Executable Assertions for Concurrent Error Detection**
    Kien A. Hua and Jacob A. Abraham

CSG PRESENTATION SLIDES

N94-70419

OVERVIEW OF ICLASS RESEARCH:

RELIABLE AND PARALLEL COMPUTING

Ravi K. Iyer

Computer Systems Group
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign

RELIABLE AND FAULT TOLERANT COMPUTING

- **Experimental Methods in Reliability/Performance Evaluation**
  - ☐ Model Extraction from Data
  - ☐ Adaptive Performability Modeling and Optimization
  - ☐ Learning-Based Approach for Load Allocation Under Failure
  - ☐ Failure Recognition and Diagnosis

- **Fault Tolerant Multiprocessor Architectures**
  - ☐ Recovery Cache Techniques for Shared Memory Architectures
  - ☐ Forward Recovery in Linked Data Structures
  - ☐ Reliable Garbage Collection for Checking System Integrity

RELIABLE AND FAULT TOLERANT COMPUTING

- **Fault Tolerant Matrix Computation**
  - ☐ Application of Algorithm-Based Fault Tolerance
  - ☐ Design of Linear Codes for Reliable Matrix Computation
  - ☐ Evaluation of Proposed Schemes Via Simulations

PARALLEL PROCESSING

- **General Applications**
  - ☐ Study of Memory Referencing Behavior
  - ☐ Investigation of Reduction in Communication Overhead
  - ☐ Use of Program Controlled Cache
- **Image Understanding (NETRA)**
  - ☐ Architecture for Integrated Vision System
  - ☐ Employ Flexible, Reconfigurable Programmable Interconnects
  - ☐ Case Study of Stereo Vision for 3-D Surface Extraction
  - ☐ Mapping and Performance Analysis of Several Algorithms on NETRA
- **Concurrent Microprocessor Design**
  - ☐ Single Chip High Concurrency Designs
  - ☐ Integration of Compiler and Hardware Design
  - ☐ Development of Design and Evaluation Tools
  - ☐ Development of Symbolic Debuggers

PARALLEL PROCESSING

- **Design Automation Applications**
  - ☐ Cell Placement, Wire Routing and Test Generation
  - ☐ Algorithm for Channel Routing
  - ☐ Parallel Implementation on the Intel Hypercube
- **AI Algorithms and Software**
  - ☐ Efficient Evaluation of AI Problems
  - ☐ Efficient Reordering of Logic Programs
  - ☐ Static versus Dynamic Restructuring
  - ☐ Learning the Design of Heuristics
  - ☐ Learning-by-Experimentation and Neural Networks

Reduction in Main Memory Traffic Through the Efficient Use of Local Memory

By
Geoffrey D. McNiven
University of Illinois at Urbana-Champaign

Summary of research conducted to satisfy requirements for Ph.D. in Electrical Engineering
October, 1987.

Problem Statement

- Performance is limited by traffic to main memory.
- Small quantity (<8k) of local memory available.
- Demand system (prefetching can only increase traffic).

? How should memory be used to effect maximum reduction in traffic?

- Knowing what to put in memory is easy: next item accessed.

☞ Knowing what to remove in order to make room is the hard part.

Overview

Two parts in this research

1. Study of memory referencing behavior.
    1. Flattening.
    2. Bounds.
    3. Histograms.
2. Recomendations for design of future memories.
    1. Program controlled cache.

Problems with classical techniques

- Classical trace analysis is a strong technique for measuring the performance of existing systems.
- Results are dependent on the underlying architecture and compiler.
    - This is not appropriate when attempting to measure the fundamental memory referencing behavior of a program.
- A memory location may hold several variables.
    - Each variable may be used for several purposes.
- the referencing behavior of a memory location is a composite of several different memory referencing behaviors.

☞ Remove some of these effects of compilation quality and memory allocation using a process called *flattening*.

Flattening

Effects

- Partition memory locations into *values*.
  - A value is written 1 time (0 for initialized data) and may be read any number of times.
  - Represents a single "use" of a memory location.
  - Provides specific data on referencing behavior of memory locations during live periods.
- Remove 2 level (register, main memory) memory hierarchy.
  - Remove effects of register allocation.
- Remove move instructions.
  - A typical product of poor compilers.
  - A byproduct of register allocation.

Bound on Max live

- The maximum number live in a class of values each of which is referenced at least R times and none of which has a lifetime longer than L is no greater than $N \leq \frac{2L}{R}$.
- The bounds for the classes with longer lifetimes become quite large.
- This bound is achieved if the following hold:
  1. All values have lifetime L.
  2. No value is referenced more than R times.
  3. All references over a period of $2L$ references are to values in this class.
- Typically, none of these conditions holds.

Bound on Average Live

- The average number live in a class of V values, none of which has a lifetime greater than L, from a trace of length T references, is no greater than $A \leq \frac{L \cdot V}{T}$.
- This bound considers referencing activity to the class of values.

Memory referencing statistics

- Once the trace has been processed, compute some statistics fo: each value and display the results in a histogram.
  - Histograms may show fraction of values with such statistics or fraction of references to values with such statistics.

AVERAGE INTERREFERENCE TIME VS. STANDARD DEVIATION
ALL TRACES, 14134430 REFERENCES, FRACTION OF 1975884 VALUES

Standard Deviation	Average Interreference Time 0-1	1-10	10-100	100-1000	1000-10000	10000-100000	100000-∞
0-1	0.16	**0.54**	0.08	**0.03**	<	<	0.02
1-10		0.05	0.01	<	<		
10-100		<	0.06	<	<		
100-1000			<	0.02	<		
1000-10000				<	<	<	
10000-100000					<	<	
100000-∞							

AVERAGE INTERREFERENCE TIME VS. STANDARD DEVIATION
ALL TRACES, FRACTION OF 14134430 REFERENCES, 1975884 VALUES

Standard Deviation	Average Interreference Time 0-1	1-10	10-100	100-1000	1000-10000	10000-100000	100000-∞
0-1	0.02	**0.15**	0.04	**0.28**	0.19	0.04	0.01
1-10		0.06	0.06	<	<		
10-100		0.02	0.07	<	<		
100-1000			<	0.04	<		
1000-10000				<	<	<	
10000-100000					<	<	
100000-∞							

More Histograms

- Consider histograms of lifetime vs. number of references.
- Fraction of references corresponds to fraction of traffic.

NUMBER OF REFERENCES VS. LIFETIME
ALL TRACES, 14134430 REFERENCES, FRACTION OF 1975884 VALUES

Lifetime	Number of References 0-1	1-10	10-100	100-1000	1000-10000	10000-100000	100000-∞
0-1		0.16					
1-10		**0.54**					
10-100		0.17	<				
100-1000		0.08	<	<			
1000-10000		<	<	<			
10000-100000		<	<	<	<		
100000-∞		0.02	<	<	<	<	<

NUMBER OF REFERENCES VS. LIFETIME
ALL TRACES, FRACTION OF 14134430 REFERENCES, 1975884 VALUES

Lifetime	Number of References 0-1	1-10	10-100	100-1000	1000-10000	10000-100000	100000-∞
0-1		0.02					
1-10		**0.15**					
10-100		0.07	<				
100-1000		0.04	0.01	<			
1000-10000		<	<	0.01			
10000-100000		<	<	<	<		
100000-∞		<	0.04	0.10	**0.26**	0.18	0.09

Memory Referencing Behavior

Average bounds

- From the information now available, we can calculate the uppe bound on the average live for each histogram class.

UPPER BOUNDS ON AVERAGE NUMBER LIVE PER CLASS
ALL TRACES, 14134430 REFERENCES, 1975884 VALUES

Lifetime	Number of References 0-1	1-10	10-100	100-1000	1000-10000	10000-100000	100000-∞
0-1		0.76					
1-10		2.32	0.01				
10-100		11.24	0.79	<			
100-1000		12.43	2.08	0.54			
1000-10000		67.25	18.06	1.92			
10000-100000		677.29	181.86	19.36	0.29		
100000-∞		17015	5354	1640	563	37	3

Max bounds

UPPER BOUNDS ON MAXIMUM LIVE PER CLASS

Lifetime	Number of References 1-10	10-100	100-1000	1000-10000	10000-100000
1-10	20				
10-100	200	20			
100-1000	2,000	200	20		
1000-10000	20,000	2,000	200	20	
10000-100000	200,000	20,000	2,000	200	20

Program Controlled Cache

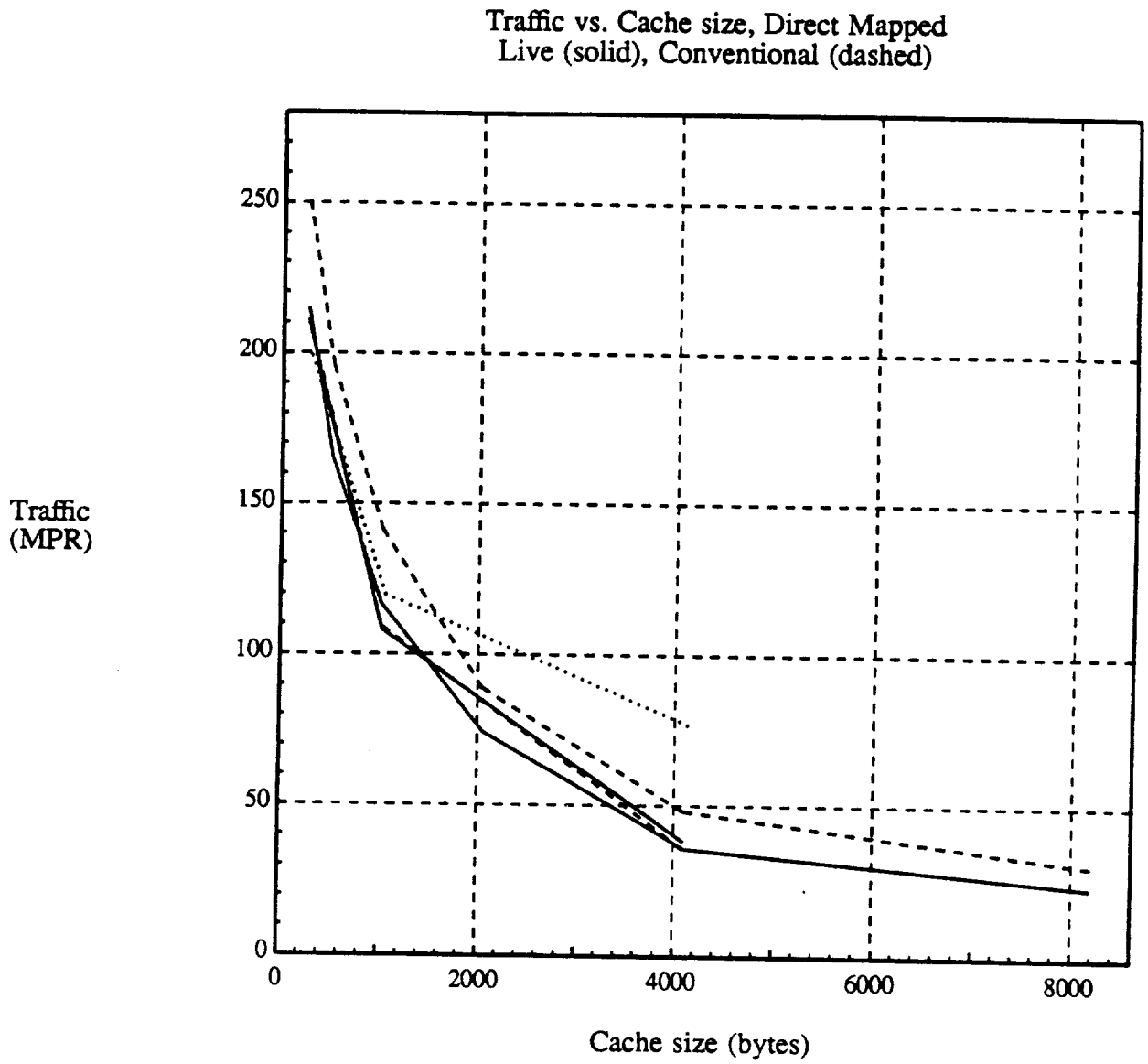
Justification

- Local memory becomes filled with values whose presence does not make the best contribution to reducing traffic.
  - Values that are not referenced for a long period of time.
    - Dead values.
- Realizing that a value is dead requires future knowledge.
  - Something computers do not have.
- During compilation, the compiler can determine approximate future knowledge.
  - When a variable becomes dead.
  - Relative times of next use.
- The compiler can insert directives to invalidate dead blocks and force the replacement of blocks that will not be referenced for a long period of time.

Program Controlled Cache

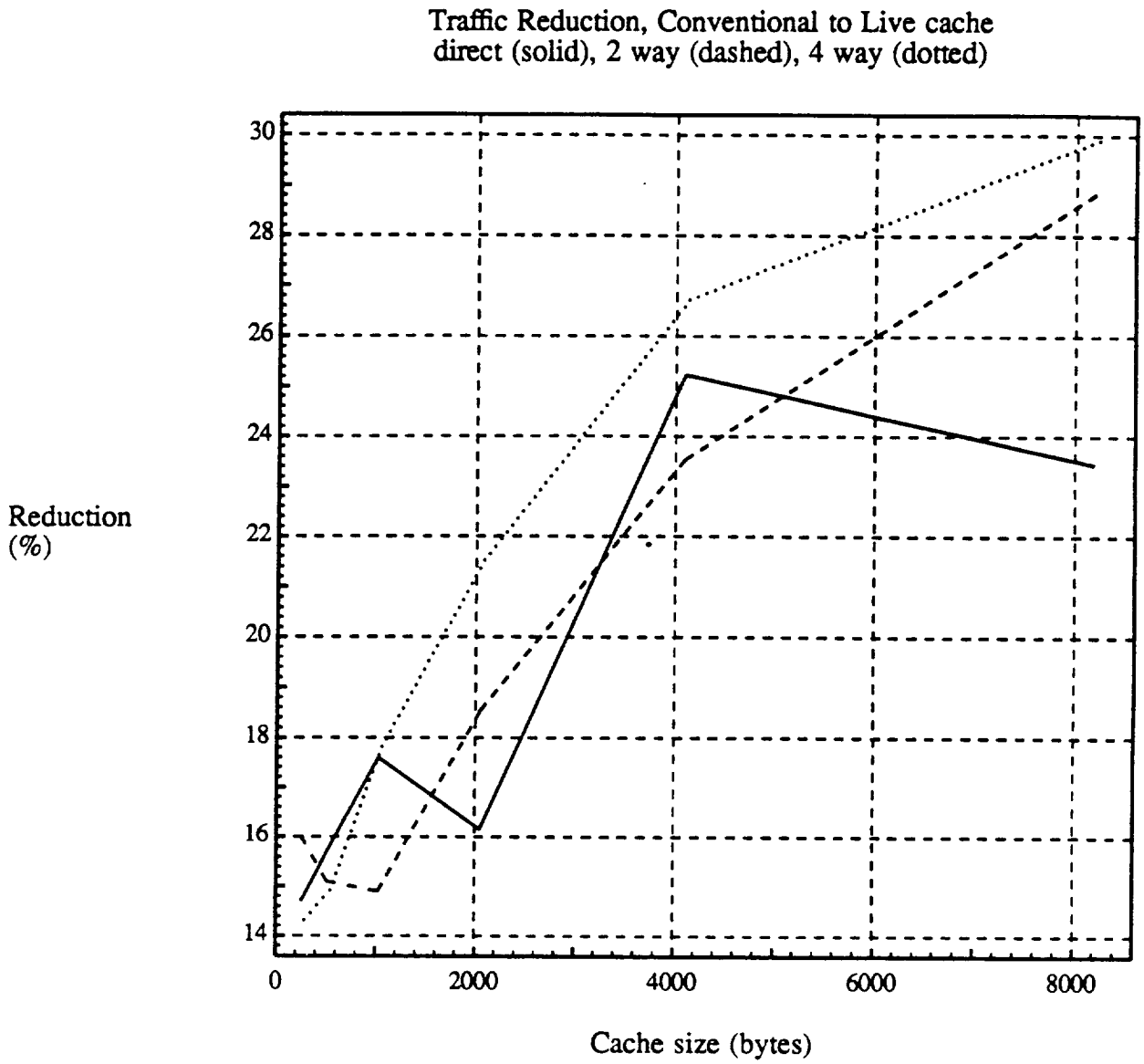
Performance

- Approximate action of compiler by trace-driven simulation.
  - Unflattened traces.



Performance

- How much improvement is achieved over a conventional cache.



Conclusions

Memory analysis

- Partitioning memory locations into contituent values provides specific information concerning memory referencing behavior.
- The bounds on memory use coupled with the histograms showing fraction of traffic per class provide the memory designer with a model for the cost in memory and benefit in performance from allocating various classes of values to local memory.

Program Controlled Cache

- Removing dead data from a cache provides a significant improvement in traffic from the back end of the cache.
  1. Read traffic.
  2. Write traffic.
  3. Write allocate traffic.
- Allows a smaller cache to be used to in place of a larger one with no performance degradation.
- Adding program control provides nearly the performance

PARALLEL ALGORITHMS

FOR DESIGN AUTOMATION

Faculty : Prith Banerjee
Student : Randall Brouwer

Computer Systems Group
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign

Outline

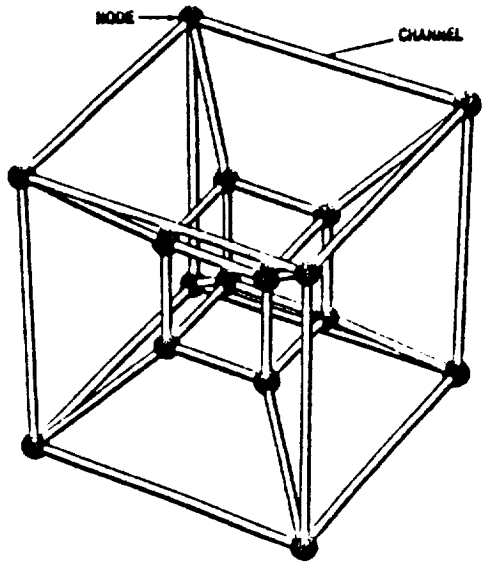
- Motivation
- Channel Routing Problem
- Previous Work
- Iterative Algorithm
- Serial Implementation
- Parallel Implementation
- Results
- Conclusions
- Future Research

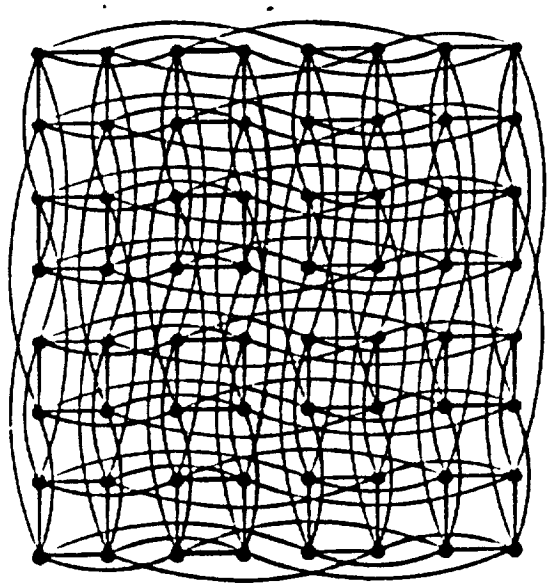
Motivation

- VLSI CAD tasks such as circuit, logic, fault simulation, and component extraction are computation intensive
- Tasks such as cell placement, wire routing, and test generation are NP-complete
- Uniprocessor algorithms take a long time
- Special-purpose hardware accelerators provide large performance improvements
- Not cost-effective, hence general-purpose multiprocessors
- Many commercial multiprocessors
  - † Shared memory, few processors (Alliant, Sequent)
  - † Shared memory, many processors (BBN Butterfly)
  - † Local memory with message-passing, many processors (Intel Hypercube, NCUBE, Ametek)

Hypercube Architecture

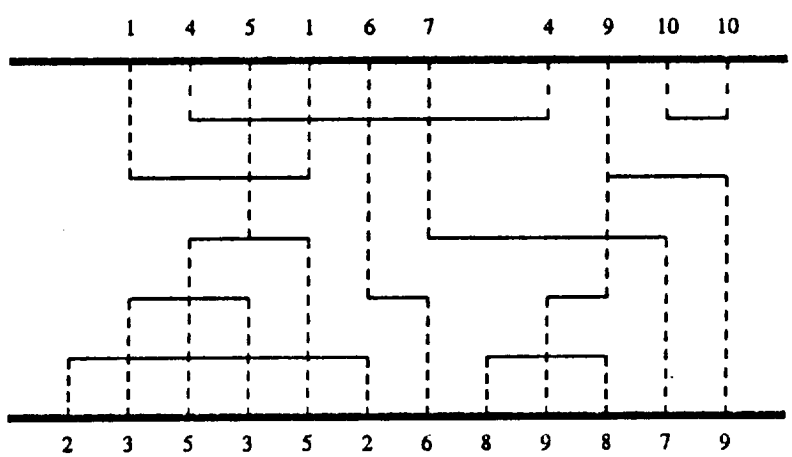
- Available hypercubes: intel iPSC, NCUBE, Ametek, Caltech MARK-III
- Binary cube interconnection
- Local memory, message passing





Channel Routing Problem

- Final stage of Physical Design Process
- Rectangular channel with numbered terminals along top and bottom edges
- Nets connect terminals with the same terminal number
- Vertical segments routed in one layer, horizontal in the other
- Example



Previous Work – all found on uni processors

- Left-Edge Algorithm -- Hashimota & Stevens ('71)
- Dogleg Algorithm -- Deutsch ('76)
- Graph theory based -- Yoshimura & Kuh ('82)
- Greedy Algorithm -- Rivest & Fiduccia ('82)
- YACR -- Sangiovanni-Vincentelli & Santomauro ('83)
- Hierarchical -- Burnstein & Pelavin ('83)
- Simulated Annealing -- Leong & Liu ('85)
- Knowledge-based (Weaver) -- Joobbani & Siewiorek ('86)

Our Simulated Annealing Channel Router

- Goals:

  † High quality routings

    ☐ minimize number of tracks used

    ☐ minimize net length

  † Parallelizable

  † Easily extendible

    ☐ unrestricted doglegging

    ☐ obstacle avoidance

    ☐ switchbox routing

  † Efficient -- time & space

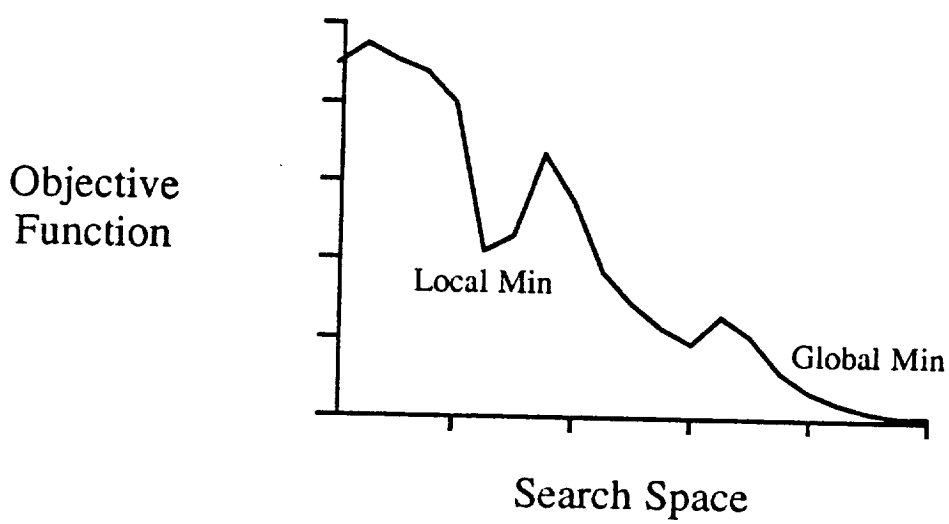
Simulated Annealing -- General

- Proposed by Kirkpatrick, et al
  - † Applied to general optimization problems
    - ☐ placement
    - ☐ traveling salesman problem
    - ☐ global routing
    - ☐ channel routing

- Analogous to physical systems
  - † system energy ≈ cost function
  - † molecule movement ≈ modeled moves

- Accept all downhill and some uphill moves

$$Prob(Accept) = e^{-\Delta / KT}$$



Algorithm

```
Set Initial Temperature & State
WHILE (Stopping Criteria Not Satisfied) DO
    Select Subnet Elements to Move
    Select Move Operation
    Calculate Change in Cost
    Evaluate Accept/Reject Based on Temperature &
    Cost
    IF (Accept) THEN
        Update State Information
    Adjust Temperature
END WHILE
Display Final Results
```

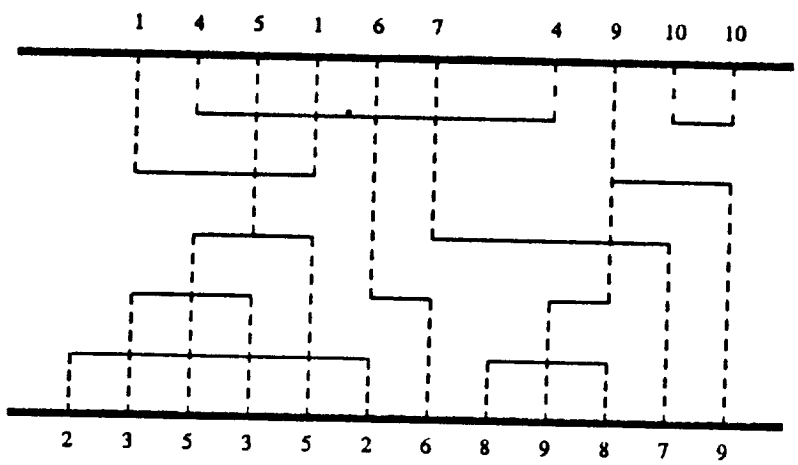
Cost Function

$$f\,(\,Tracks\,,\,Overlap\,,\,Goodness\,)$$

$$\Delta cost = (\alpha \times \Delta_{Channel\ Width})$$

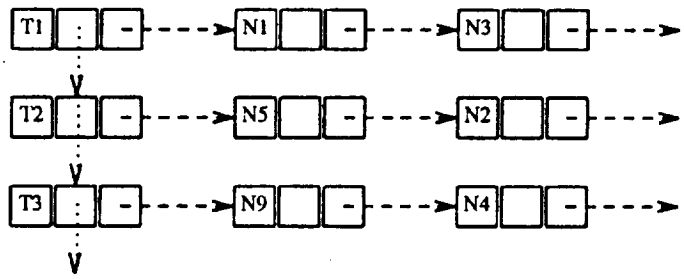
$$+\,(\beta \times \Delta_{Net\ Overlap})$$

$$+\,(\gamma \times \Delta_{Net\ Length})$$



Serial Implementation

- Data Structure
  - † Track list



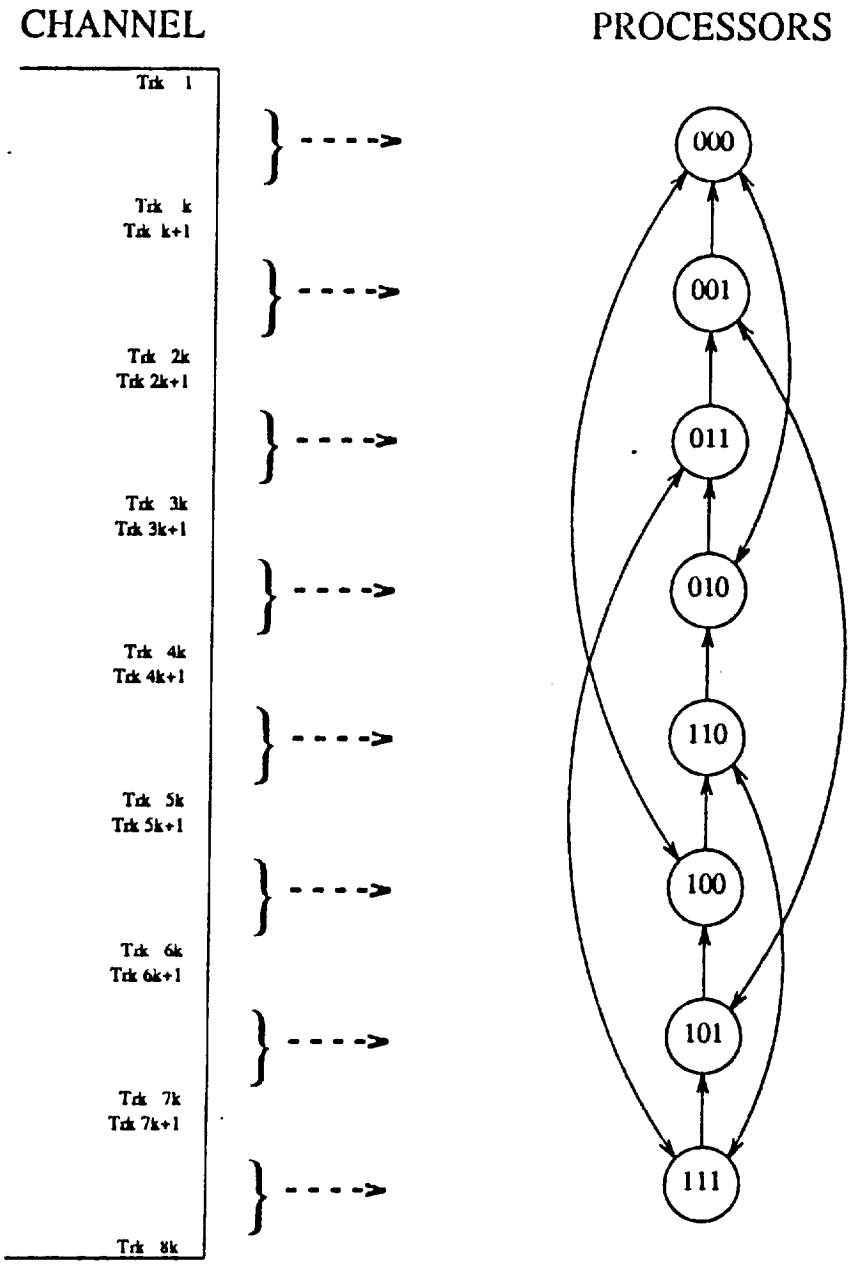
  - † Column list
    - □ Array of Column Structures

- Moves
  - † Displace -- Displace subnet to a new track
  - † Exchange -- Exchange two subnets

Parallel Implementation

- Target System: Intel iPSC Hypercube
- Partitioning
  - † Channel
    - □ Assign subsets of tracks to each node
    - □ Adjacent track subsets are assigned to adjacent nodes
  - † Data
    - □ Each node maintains data for its track subset
- Moves
  - † Intra-Node Displace
  - † Inter-Node Displace
  - † Intra-Node Exchange
  - † Inter-Node Exchange

Node Domain Mapping

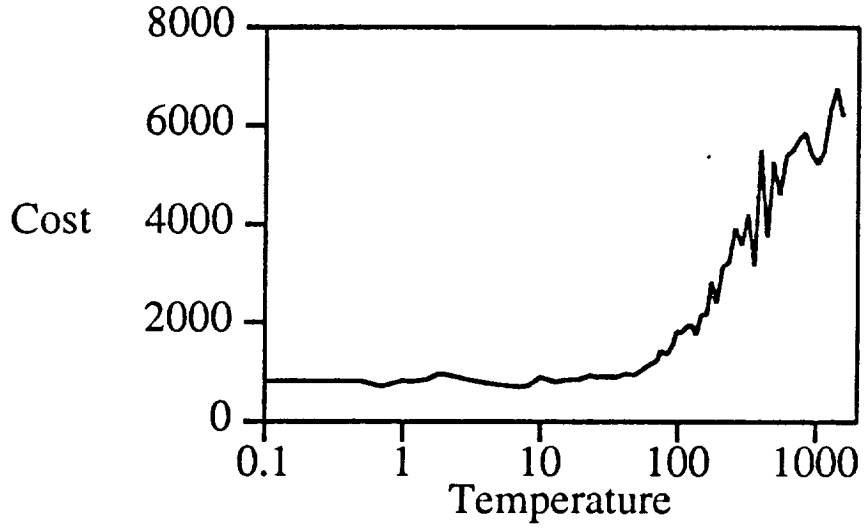


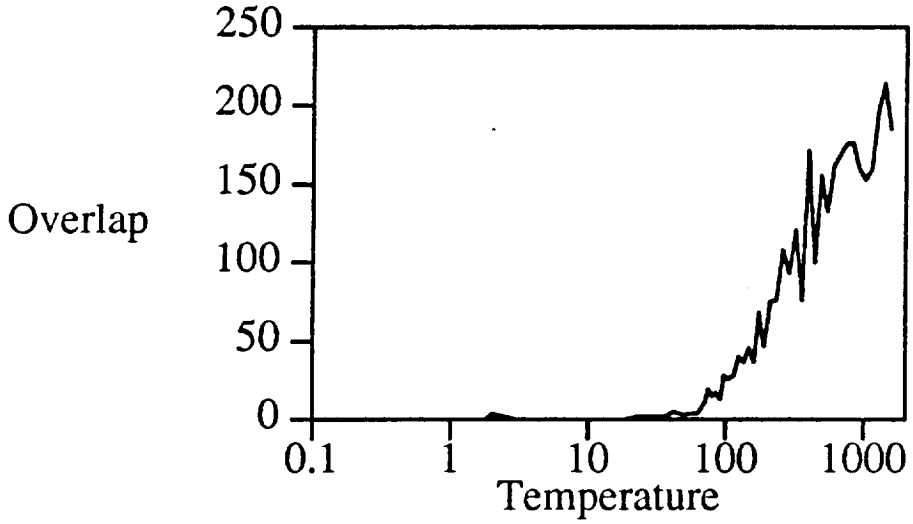
Heuristics

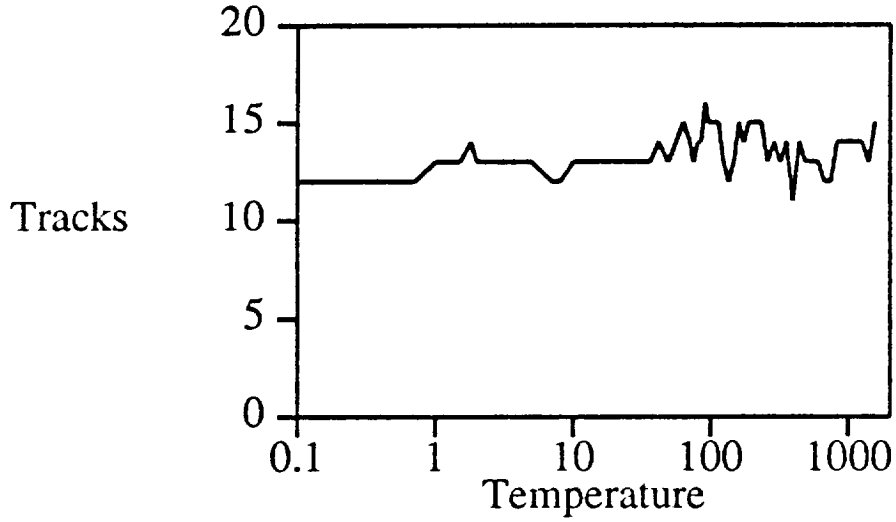
- No Inter-Node Exchange below $T_{Threshold}$
- More Iterations at Low Temperatures
- Penalize Excessive Length in Subnets
- Weighted Subnet Selection
- Weighted Displacement Track Selection
- Approximated Vertical Constraint Graph

  † Apply to Subnet Cost Calculation

Results


Cost
8000
6000
4000
2000
0
0.1
1
10
100
1000
Temperature


Overlap
250
200
150
100
50
0
0.1
1
10
100
1000
Temperature


Tracks
20
15
10
5
0
0.1
1
10
100
1000
Temperature

Conclusions

- Convergence of parallel algorithm dependent on annealing parameters

- Simulated annealing more suitable for certain problems like placement and partitioning rather than routing

- Issues in parallel algorithm

  † Computation/communication ratio

  † Cost of updating all nodes

Future Research

- Evaluate performance of standard cell placement and channel routing algorithms on real hypercube

- Investigate parallel algorithms for other tasks

  † Global Routing

  † Integrated placement and routing package

- Related research on parallel algorithms funded by other grants

  † Circuit extraction (IBM and NSF)

  † Automatic test generation (JSEP)

  † Timing simulation (NSF)

PARALLEL ARCHITECTURES FOR IMAGE UNDERSTANDING

Faculty : **Janak H. Patel**

Student : **Alok N. Choudhary**

Objective

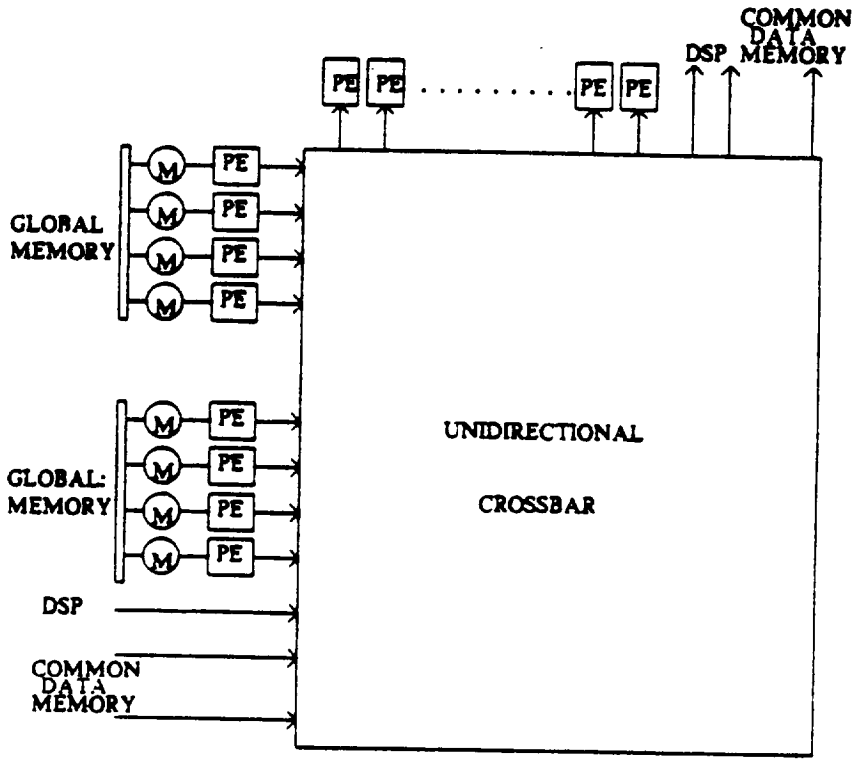
- Identify Architecture for Integrated Vision System
  - For Example
    - ☐ Stereo Vision
    - ☐ Motion Detection
    - ☐ Texture
- Approach
  - ☐ Large Grain Parallelism
  - ☐ Small Number of Powerful Processors
  - ☐ Flexible, Reconfigurable and Programmable Interconnect
  - ☐ Algorithms Mappings using Multi-Dimensional Divide-and-Conquer
  - ☐ Mappings independent of Input Data Size
- Integration of Algorithms
  - ☐ Data Transfer (including I/O)
  - ☐ Data Structures
  - ☐ Task Scheduling
  - ☐ Load Balancing

Current Research

- ☐ Stereo Vision for 3-D Surface Extraction
  - ☐ Involves Algorithms Suitable for
    - SIMD
    - MIMD
    - Hybrid Implementation
- Set of Tasks Mapped
  - ☐ 2-D Convolution
  - ☐ Matching
  - ☐ Surface Fitting
  - ☐ Locating Contours
  - ☐ Connected Components Labeling

An Image Processing Architecture

(Sharma, Patel and Ahuja)



PE : Processing Element
M : Local Memory

. Processor Cluster Organization

Stereo Vision

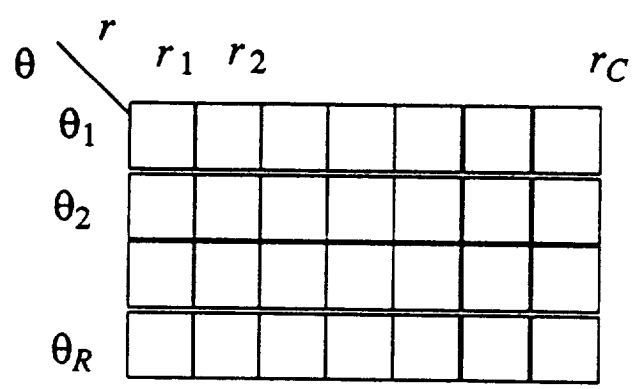
Stereo Vision Computation Flow				
Task	**Algorithm(s)**	**Type**	**Input**	**Output**
Feature Detection (e.g., edges)	2-D Convolution	SIMD	Image	Edges
Feature Match	Window Match Relaxation	Hybrid	Edges L & R image	3-D Points
Surface Fitting	Hough Transform Convex Hull	Hybrid	3-D Points	Surface Patches
Locating contours	Template Matching	MIMD	Surface Patches	Contour Description
Surface Interpolation	Weighted Average in varying size window	MIMD	Surface, Contour	3-D Object Description
Parameter Computation	Relaxation	MIMD	3-D Object	Parameters

Example : Hough Transform

- Algorithm to detect lines, curves, surfaces
- Computation done in parameter space
- Example : Line Segment
  - ☐ $r = x\cos\theta + y\sin\theta$
  - ☐ Parameters are r and θ
  - ☐ Count votes for quantized r values for all quantized θ values
  - ☐ Local maxima represent line segments
- Mappings onto Cluster
  - ☐ Data partitioned onto processors
  - ☐ Parameters partitioned onto processors
  - ☐ Parameters partitioned with data broadcast

Hough Transform Mapping

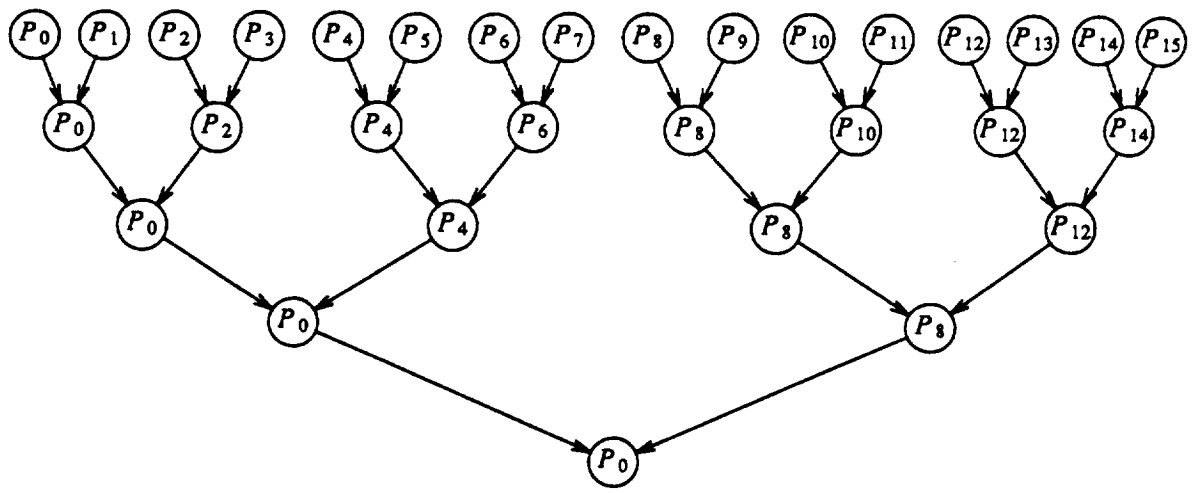
- Data Partitioning
- Each processor computes vote counts for $\frac{n^2}{p^2}$ image
- Steps
  - ☐ Compute Vote Counts (All processors in Parallel)
  - ☐ Combine partial results (Tree sum) (All Processors in Parallel)
  - ☐ Distribute the Accumulated Vote Count (One Processor)
  - ☐ Report Local Maxima (All Processors in Parallel)



ACCUM_ARRAY(i,j) FOR ONE PROCESSOR

Continued

- Tree Sum Method to Combine Partial Accumulator Arrays



An EXAMPLE WITH 16 PROCESSORS

- Time Complexity

$$\text{Floating Point Operations} = 3\times\left\lceil \frac{n^2}{p^2} \right\rceil \times f \times R$$

$$\text{Number of Comparisons} = \left\lceil \frac{R}{p^2} \right\rceil \times C \times f_m \times W$$

$$\text{Communications (in words)} = 2\times\log_2 p \times R \times C$$

$$\text{Memory Size} = \left\lceil \frac{n^2}{p^2} \right\rceil (pixels) + R \times C \ (words)$$

- f = fraction of Black Pixels
- f_m = No. of Local Maxima
- W = window size

Performance Analysis

- Computation, Communication and Storage Requirements

Hough Transform				
Mapping	Computation (T_{cp})	Communication (T_{com})	Total	Memory
Data	$3\times\left\lceil\frac{n^2}{p^2}\right\rceil\times f\times R+\left\lceil\frac{R}{p^2}\right\rceil\times C\times f_m\times W$	$2\times\log_2 p\times R\times C$	$T_{cp}+T_{com}$	$\left\lceil\frac{n^2}{p^2}\right\rceil+R\times C$
Parameter	$3\times\left\lceil\frac{n^2}{p^2}\right\rceil\times f\times R+\left\lceil\frac{R}{p^2}\right\rceil\times f_m\times W$	----	T_{cp}	$n^2+\left\lceil\frac{R}{p^2}\right\rceil\times C$
Alt. Parameter	$3\times\left\lceil\frac{n^2}{p^2}\right\rceil\times f\times R+\left\lceil\frac{R}{p^2}\right\rceil\times f_m\times W$	$f\times n^2$	$\mathrm{Max}(T_{cp},T_{com})$	$\frac{R}{p^2}\times C$
Connected Component Labeling				
	$(1+f\times(4+f_l))\times\left\lceil\frac{n^2}{p^2}\right\rceil$	$\frac{n}{p}\times(6\times\log_2 p-3)$	$T_{cp}+T_{com}$	$(1+f_l)\times\left\lceil\frac{n^2}{p^2}\right\rceil+l$

- f = Fraction of Significant Pixels
- f_m = No. of Local Maxima
- W = Appropriate Window Size
- f_l = Fraction of Significant Pixels with 0 as West Neighbor
- N_C = No. of Connected Components

Future Research

- Mapping of Additional Tasks
- Performance Analysis
  - ☐ Analytical
  - ☐ Simulation and Implementation
- Identify Memory Requirements
- Application of Stereo Vision for Real Time Motion Detection
  - ☐ Minimum Requirements on General Purpose Architecture
  - ☐ Specific Architecture Design
- Implementation of Algorithms on Available Architecture such as Intel Hypercube

OMIT

EFFICIENT REORDERING OF PROLOG PROGRAMS

Markian M. Gooley and Benjamin W. Wah

Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
Urbana, Illinois

This research was supported in part by the National Aeronautics and Space Administration under grants NAG 1-613 and NCC 2-481.

Motivation for this research

Why do it?

- Prolog programs are inefficient
- Execution:
  - traverse AND/OR graph
    - — depth-first
    - — left-to-right
- May traverse many large subgraphs (goals) before failing
- May try more-expensive alternatives first
- If search fails, backtrack
  - attempt to find solution in another way
  - must search again

Motivation for this research (continued)

Why try reordering, in particular?

- Past research on limited reordering (Warren)
  - speedups of 1 to 2 orders of magnitude on certain simple programs
  - worth trying on complex programs
- Programmers exploit order for efficiency
  - Put inexpensive, likely-to-fail goals first in clause
  - Put clauses last if heads unlikely to unify
- However:
  - Don't know cost, probability of success precisely
  - Won't change order if gain not obvious and readability suffers

⇒ Greater efficiency possible with automatic reordering.

A very short course in Prolog

⇒ *Pro*gramming in *Log*ic:

- Program: collection of logic sentences in clausal form
  - AND of zero or more goals implies conclusion
- Premises and conclusion take arguments
- *Predicate*: clauses with same name, arity (# of arguments)
  - Disjunction of its clauses

Logic		Prolog	
a(X)	*tautology*	a(X).	*fact*
a(X) → b(X)	a(X) *implies* b(X)	b(X) :- a(X).	b(X) *if* a(X)
a b → c	a *and* b *implies* c	c :- a, b.	c *if* a *and* b
premise		*goal*	
conclusion		*head*	
conjunction of premises		*body*	

Program execution:

- query: **:- a(X), b(X).**
- "**a(X)** and **b(X)** do not hold for any **X**."
- System searches depth-first, text-order for counterexamples
- **Backtracks** if it cannot, seeks alternative solutions

A very short course in Prolog (concluded)

Execution as traversal of an AND/OR tree:

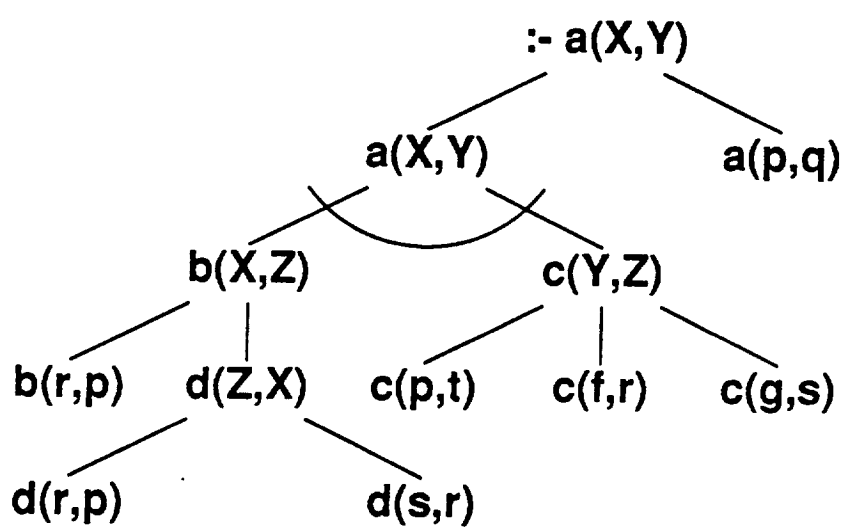
- Example: query **:- a(X,Y).** and clauses:

```
a(X,Y) :-  b(X,Z),  c(Y,Z).
a(p,q).

b(r,p).
b(X,Y)  :-  d(Y,X).

c(p,t).
c(f,r).
c(g,s).

d(r,p).
d(s,r).
```



The problem to be solved:

How can we reorder Prolog clauses and goals to reduce the expected cost of a query?

- Try to minimize number of predicate calls
  - unifications also a good measure; CPU time too coarse perhaps
- Process should require limited intervention
- Reordered program must be equivalent to original
  - Have found levels of equivalence
    1. Reflexive
    2. Set
    3. Tree
    4. Sub-tree

  ⇒ Set-equivalence most useful
    — preserved by reorderings presented here

Approach:

Generate possible reorderings:

- All valid in "pure", non-recursive Prolog
- Real Prolog: fixity, semi-fixity, control
- Demands for certain **modes**
  - especially from built-in predicates
  - demand that some arguments be constants
  - order must satisfy demands.
- Need for extended mode system
  - conventional systems too vague
  - have developed **legal modes**
- Recursion: avoid infinite loops

Find cheapest reordering of these:

- Find cost of each permitted reordering
  - model predicate or clause as a Markov chain
    - — delivers expected probability, cost
    - — given those of constituents
- Choose cheapest order
  - different orders for different modes

Restrictions on reordering:

Restriction	Cause(s)	Effect		Propagation
		on goals of clauses	on clauses of predicates	
Mode	Demands of built-in predicates; recursions	order must satisfy demands	none	demands may pass to ancestors
Fixity	Predicates with side-effects (*e.g.* I/O)	goal immobile within clause	clause immobile within predicate	ancestors become fixed
Semi-fixity	Certain success (failure) in some modes	fixed with respect to certain variables	none	ancestors semi-fixed (depends on variables)
Immobility	Cut (!) as goal	can't reorder goals before cut	clauses can't cross bearer of cut	descendants of goals before cut: immobile
Control	Explicit disjunction: **a ; b**	confined to halves of disjunction	none	none
Control	Implication: **if -> then ; else**	**if** immobile; **then,else** confined	none	none
Recursive	Recursions	avoid orders that cause infinite loops	same	none

Markov-chain model and method:

Description

- Chain estimates probability, cost for clause
  - given values of them for goals
  - need only give values for low levels
    - — values propagate upward
  - textbook mathematics
- Method, in brief:
  - generate legal orders of goals
  - feed probability, cost of goals into model
  - compare results, keep best version for each mode
- Model easily extended to special cases
  - control predicates
  - determinism, etc.

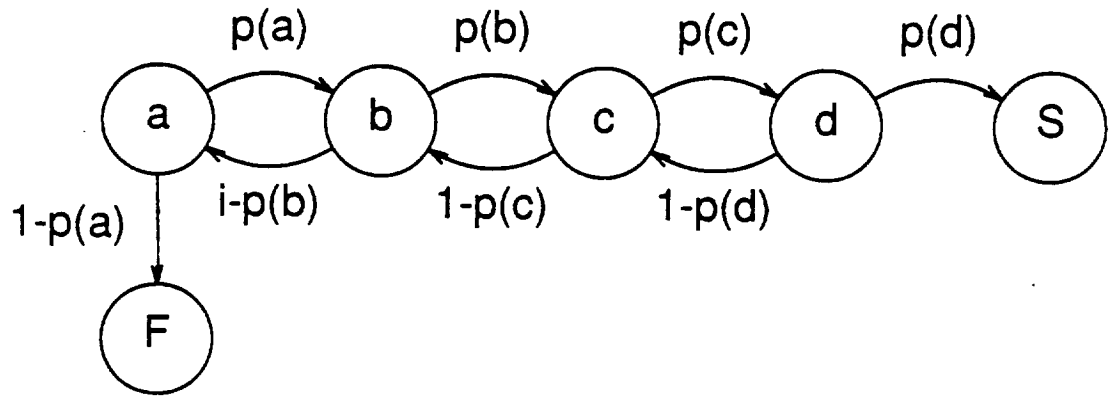
Limitations

- Potentially expensive
  - n! for n goals
    - — usually restricted (fixity, etc.)
    - — *i.e.*, 5! versus 2!2!
- Finding initial probabilities
  - problematic; depends on program

The Markov-chain model and method:

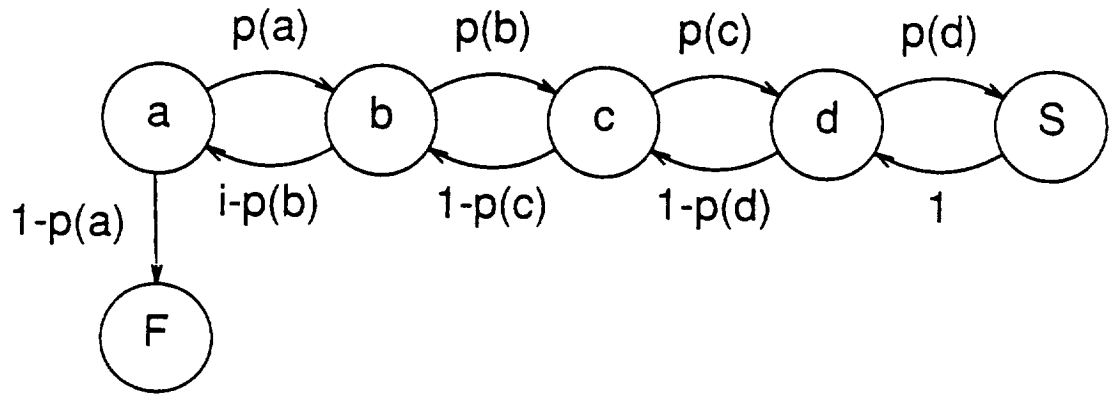
Example: h :- a , b , c , d .

- Single solution: general purpose



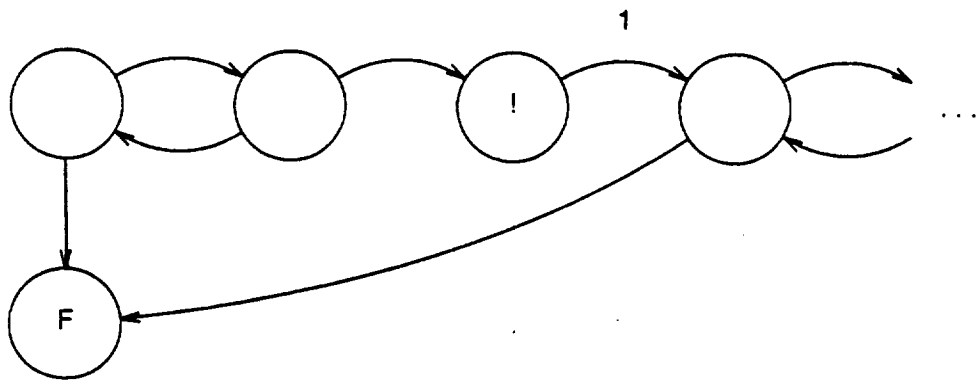
— **p(i):** success probability for goal **i**
— **S:** success state
— **F:** failure state

- All solutions

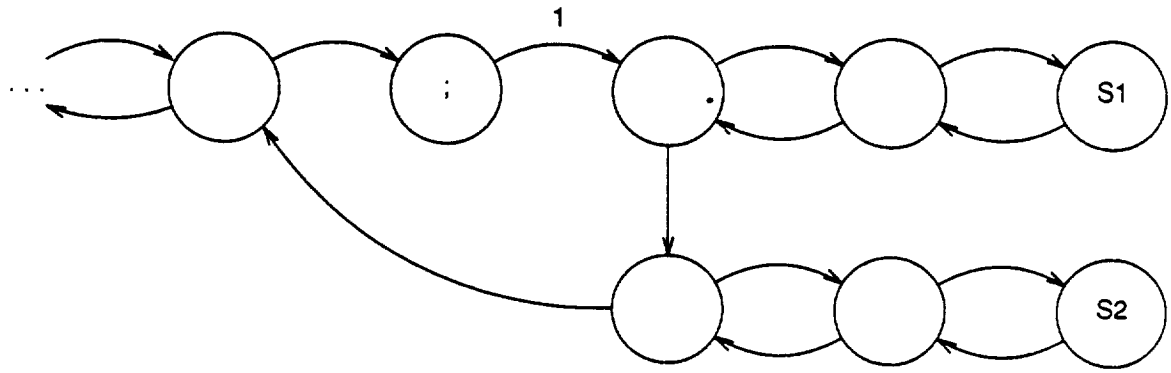


Special chains:

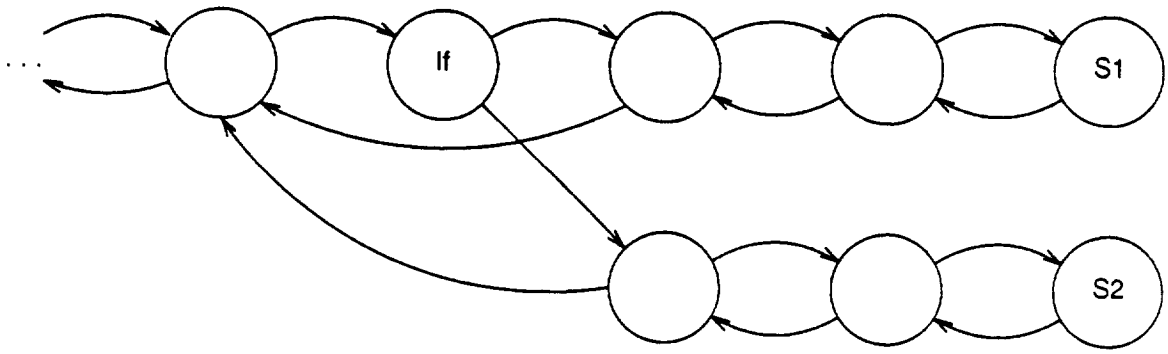
Cut



Disjunction



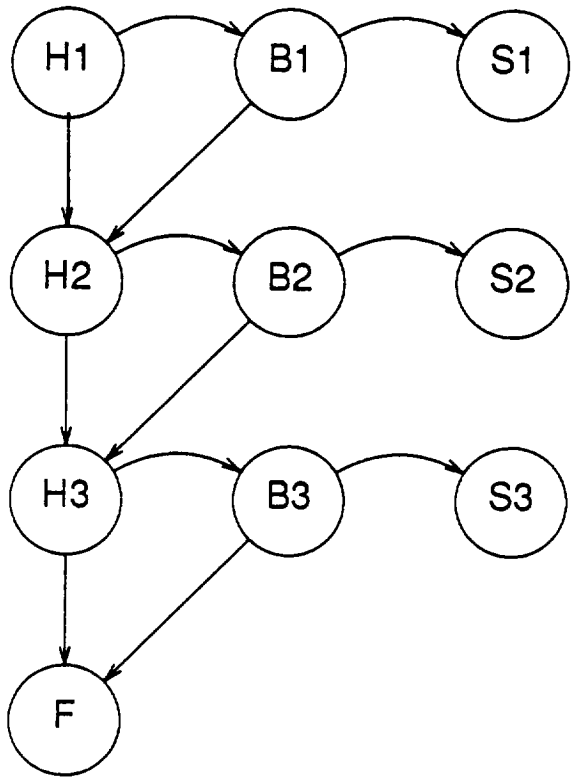
Implication with disjunction



Markov-chain model and method: clauses

Can model clauses of predicate as chain

- One state per head
  - outgoing arcs with probability that unification succeeds, fails
- One state per body
  - success/failure probabilities on arcs
- One state per success
- One state for failure



Conclusions

Summary

- Reordering Prolog for more efficient execution
  - speedups to two orders of magnitude (unrestricted)
  - lower but useful if restricted
- Levels of equivalence characterized
  - utility at each level
  - transformations which preserve a given level
- New mode system for Prolog
- Automatic reordering system
- Applicability to parallel processing

Future Work

- Better mode system
- Techniques for recursive predicates
  - improved analytical methods
  - expert system if necessary
- Exploiting other properties
- Integrated reordering/transformation system.

Concurrent Microprocessor Design and Implementation

Wen-mei Hwu
Coordinated Science Laboratory
University of Illinois
Urbana, IL 61801

(1) Design and Implement Microprocessors with Highly Concurrent Microarchitectures

(2) Integrate the Compiler and the Hardware Design to Exploit Concurrency

(3) Develop Tools for Designing, Implementing, Debugging, and Evaluating Microprocessors with Highly Concurrent Microarchitectures

(4) Develop Symbolic Debuggers and Exception Handlers for Microprocessors with Highly Concurrent Microarchitectures.

Microprocessor Design and Implementation

HPSm (Berkeley)

Single chip microarchitecture supporting out-of-order execution, fully pipelined FPU, and executing multiple instructions per cycle.

1.6 um COMS (HP) technology, 80ns cycle time, 4-phase clock

37.5MIPS/12.5MFLOPS peak and 22.5MIPS/3.0MFLOPS sustained

Next Chip

1.6 um CMOS (HP) technology, 40ns cycle time, 2-phase clock

50.0MIPS/25.0MFLOPS peak and 40.0MIPS/6.0MFLOPS sustained

C Compiler Development

"Sure, it can be done at Compile time."

(1) Allocate/Assign registers for concurrent Microarchitectures

(2) Schedule operations to reduce the hardware scheduling overhead

(3) Remove Output and Antidependencies to reduce hardware renaming overhead

(4) Support symbolic debuggers and exception handlers

(5) More conventional code optimizations

Tool Development (MATE)

(1) Sequential state consistency checker

(2) Microarchitecture state serializer

(3) Debugging Environment for Concurrent Microarchitectures

(4) Testability analysis and test pattern generator for Concurrent Microarchitectures

Symbolic Debuggers and Exception Handlers

(1) Constructing consistent execution states

(2) Handling exceptions with inconsistent states

floating-point operations
memory management
system calls
...

(3) Extension to symbolic debuggers and exception handlers in multiprocessor environments

EXPERIMENTAL METHODS FOR COMPUTER SYSTEM PERFORMANCE/RELIABILITY EVALUATION

Ravi K. Iyer

Coordinated Science Laboratory

University of Illinois at Urbana-Champaign

Students

M. Devarakonda
K. Thearling
R. Llames

EXPERIMENTAL METHODS FOR PERFORMANCE/RELIABILITY EVALUATION

Extraction of Behavioral Patterns from Measurements

- **Joint Modeling of Resource Usage/Failure/Recovery Process**
  - ☐ Model Identification Using Statistical Pattern Recognition
  - ☐ Application for Adaptive Performability Tuning
- **Adaptive Resource Usage Prediction**
  - ☐ Statistical Clustering to Identify System Resource Usage Patterns
  - ☐ Application for Load Re-allocation Under Failure

EXPERIMENTAL METHODS FOR PERFORMANCE/RELIABILITY EVALUATION

- **Fault Propagation in Shared Memory Multiprocessors**
  - ☐ Hardware Instrumentation for Trace Acquisition
  - ☐ Design of Fault Injection Experiments
  - ☐ Investigation of Multiple and Near Coincident Faults

- **Failure Diagnosis**
  - ☐ **Symptom Based**
    - Error Clustering
    - Symptom Extraction
    - Statistical Pattern Recognition
  - ☐ **Reasoning Based**
    - System Structure Representation Using Fault-Behavior Theorems
    - Resolution of Theorems with known Inputs
    - Partitioning to Reduce Diagnosis Complexity
    - Extensions to Probabilistic Inputs and Higher Level Diagnosis

EXPERIMENTAL METHODS FOR PERFORMANCE/RELIABILITY EVALUATION

- **Performance Measurements on Lisp Systems and Programs**
  - ☐ Low Interference Sampling of Virtual Memory and Garbage Collection Meters
  - ☐ Dynamic Evaluation Under Real Workload Conditions
  - ☐ Paging, Storage Allocation, Memory Space Usage, and Garbage Collection Activity
  - ☐ Applications to Adaptive, Self-Tuning Systems, Design of Lisp Architectures, Synthetic Workloads for Performance Evaluation

ADAPTIVE RESOURCE USAGE PREDICTION

- Measured program resource usage in UNIX environment
  - ☐ CPU used by a process
  - ☐ Bytes of IO performed
  - ☐ Memory required
- Performed cluster analysis on measured data
  - ☐ Used K-means clustering algorithm
  - ☐ Analysis resulted in seven clusters (i.e. found program resource usage to fall into seven categories)
- Clusters used to build state-transition models
  - ☐ Models represent the dynamics of resource usage of a program run several times
  - ☐ Models show the transition probabilites of a program
- Program based resource usage prediction
  - ☐ Scheme predicts resoruces required for a process at START of its life given:
    - the programs identity
    - past resource usage of the program

Cluster Analysis

- Three resources (CPU time, file I/O, and memory) form a 3D space.

- Processes are points in the resource space.

- The k-means algorithm identifies k clusters, $C_1, C_2, \ldots, C_k$, that minimize:

$$\sum_{i=1}^{k} \sum_{j} (\bar{x}_i - x_{ij})^2$$

where $x_{ij} \in C_i$ and $\bar{x}_i$ is the centroid of the cluster C_i.

- Results:

Cluster Number	Cluster Frequency	Cluster Statistics (median values of the resources)		
		CPU (seconds)	File I/O (Kbytes)	Memory (Kbytes)
1	11.26%	4.62	13.870	194.726
2	2.64%	0.25	0.000	446.461
3	6.43%	0.80	8.486	192.444
4	9.42%	0.25	0.732	117.294
5	29.76%	0.07	0.000	16.000
6	29.69%	0.25	2.000	50.238
7	10.77%	1.54	103.804	134.386

Resource Usage Model

- Build one for each program.

- When a process (a program in execution) terminates determine its cluster membership (or state).

- For each program, remember these cluster numbers in the terminating order of processes.
  - ☐ Ex: 5, 5, 5, 7, 7, 7, 5, ..., and 5.

- Compute:
  - ☐ Transition probability from state i to j:

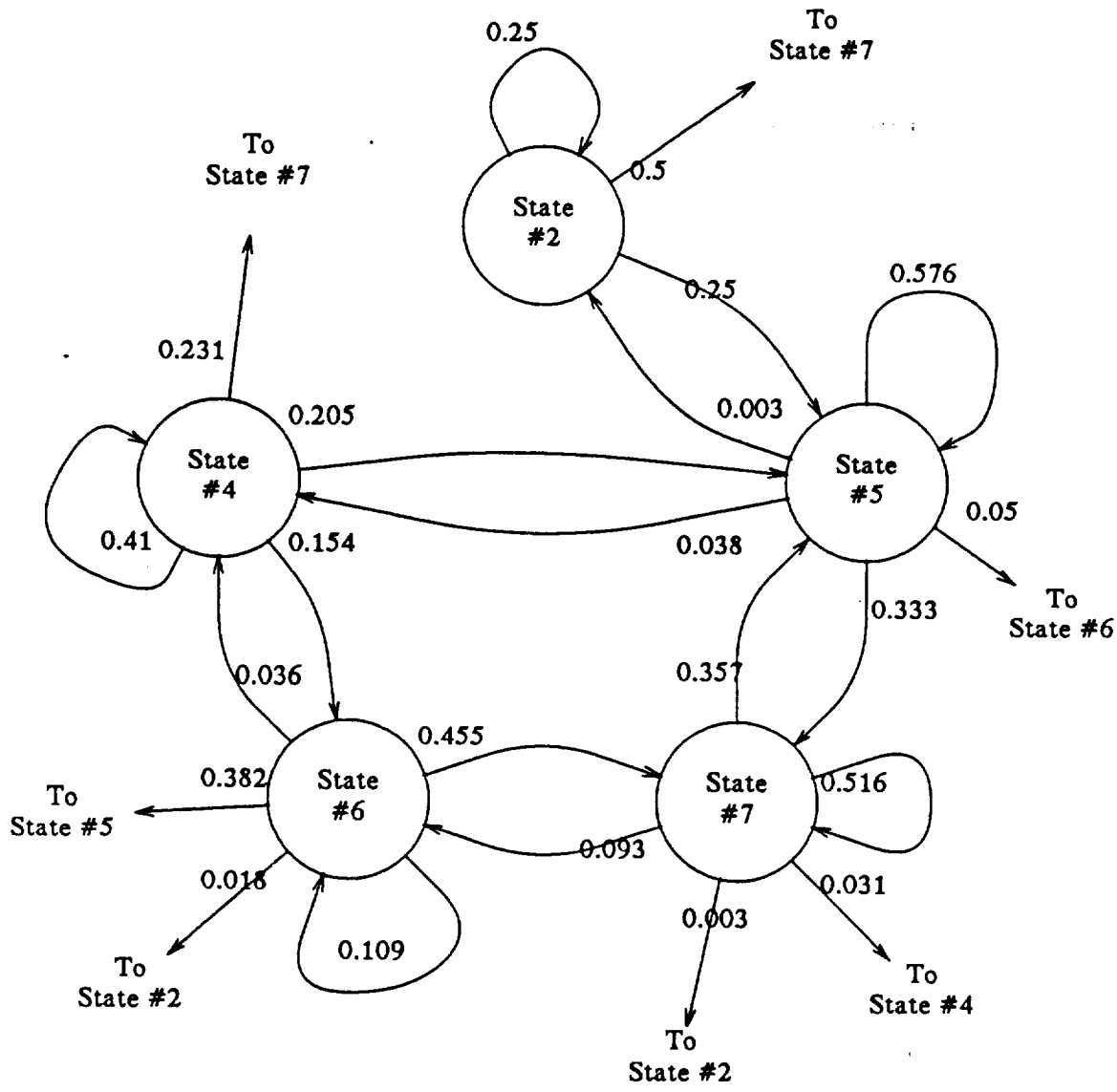
$$p_{ij} = \frac{transitions\ from\ state\ i\ to\ state\ j}{transitions\ from\ state\ i}$$

  - ☐ Visit ratio for state i:

$$v_i = \frac{occurrences\ of\ state\ i}{total\ occurrences\ of\ states}$$

Resource usage model — for a program from data

State-Transition Diagram



Visit Ratios

cluster#	1	2	3	4	5	6	7
ratio	-	0.005	-	0.056	0.450	0.077	0.412

Resource Requirements Computation

- Say, l is the state of the previous execution of a program

- Then, probability that next execution will be state j is:

$$p_{lj}, \quad j = 1, ..., N$$

- So, kth resource for the next process is:

$$r_k = \sum_{j=1}^{N} (p_{lj} \times d_{jk}), \quad k = CPU, I/O, MEM$$

where, d_{jk} is <u>typical</u> resource requirement for jth state.

- Best, if d_{jk} is the centroid of jth <u>subcluster</u>

  — A subcluster is a part of a cluster that consists of a few past executions of the program.

How Good is the Prediction?

- A trace-driven experiment:
  Predict & compare with actual

- Correlations between predicted and actual:

Resource	Correlation Coefficients	
	Rank	Product-Moment
CPU Time	0.8379	0.8406
File I/O	0.8105	0.1974
Memory	0.8925	0.8834

— A corr. coff. of 1.0 implies perfect prediction.

— Rank correlation allows <u>nonlinear</u> relationship.

Prediction Quality: A Note

- When a minimum of 3 past executions are used, the prediction scheme has no prediction for:

  - ☐ About 4% of processes;
  - ☐ About 8% of CPU time.

Conclusions

- A probabilistic scheme for predicting CPU time, file I/O, and memory requirements of a process.
  - — A program-based scheme.
- A cluster analysis identifies resource states.
- A state-transition model represents dynamics of resource usage in past executions of a program.
- Computes resource requirement for the next execution using:
  - ☐ Transition probabilities;
  - ☐ Typical resource usage of each state.
- Quality of Prediction:
  - ☐ Predicted & actual CPU times correlate at 0.84.
  - ☐ About 82% of errors in CPU time are less than 0.5 std devs.

RECOVERY FROM FAILURES IN PARALLEL ARCHITECTURES

W. Kent Fuchs

Coordinated Science Laboratory
University of Illinois

Students

A. Gupta (NASA ICLASS)
P. Chen
J. Li
K. L. Wu

OUTLINE

- Cache-based Recovery for Shared Memory Multiprocessor Systems
- Forward Recovery in Linked Data Structures
- Reliable Garbage Collection and Recovery

CACHE-BASED RECOVERY FOR SHARED MEMORY MULTIPROCESSOR SYSTEMS

Objective

- Recover from transient processor failures without a global restart

Motivation

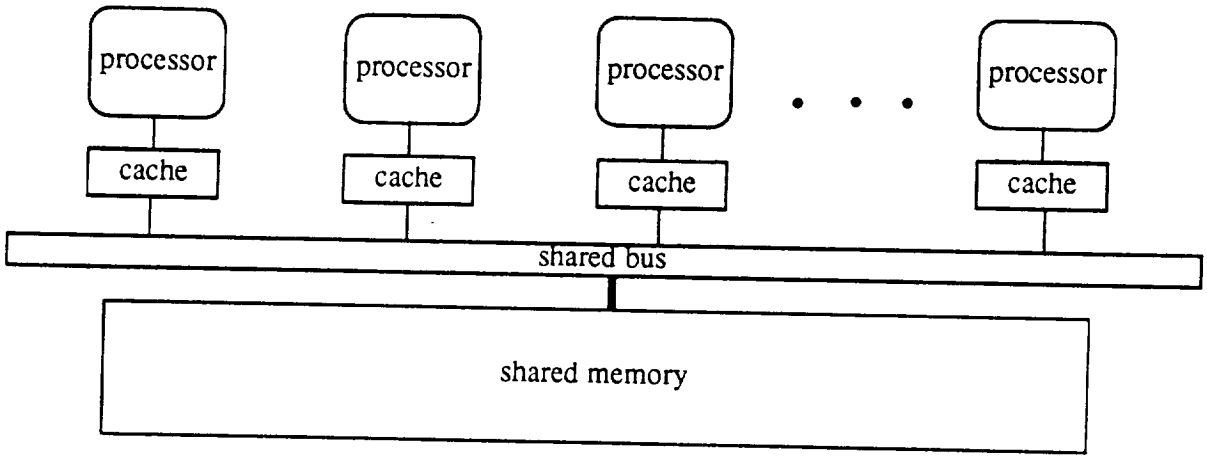
- Few known solutions to recovery in shared memory multiprocessors
- Checkpointing recovery schemes for distributed systems are not easily applicable nor cost effective

Approach

- A private cache-based checkpointing scheme to maintain a consistent global state
- A simple recovery technique without rollback propagation

System Architecture

- Write-back cache
- One process in each processor
- Interprocess communication through accessing shared variables
- Bus-based cache coherence protocol



Checkpointing Algorithm

if a block is to be written back, OR a dirty block is to be read by another processor

then begin

(1) copy the working register set into the back-up set;

(2) mark all dirty cache blocks as *Unwritable*;

end

Recovery Algorithm

if an error is detected

then begin

(1) invalidate cache blocks except those which are *Unwritable*;

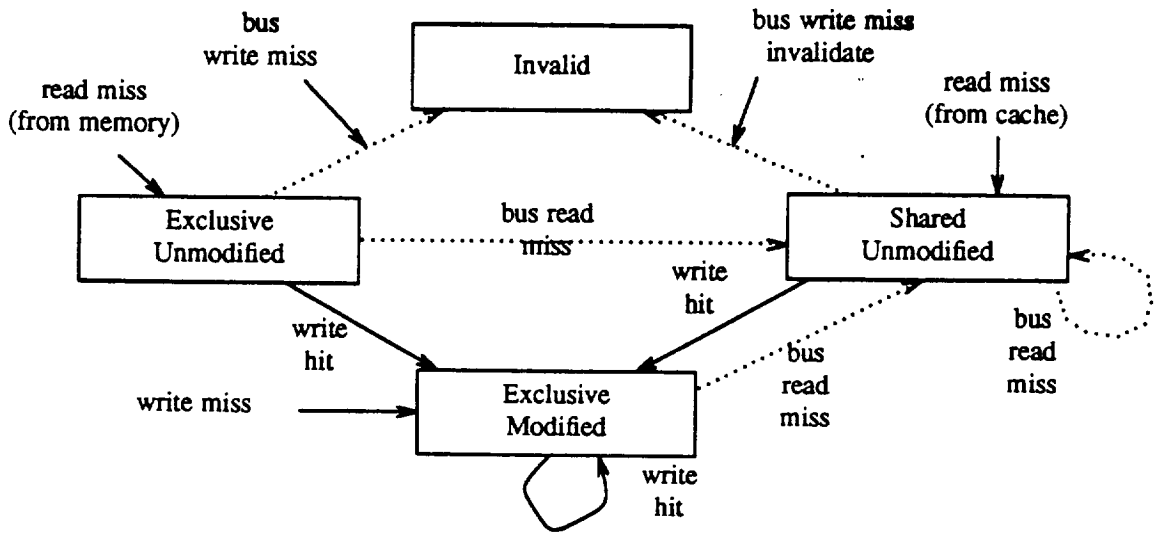
(2) reload the working register set from the back-up set;

(3) resume normal operation;

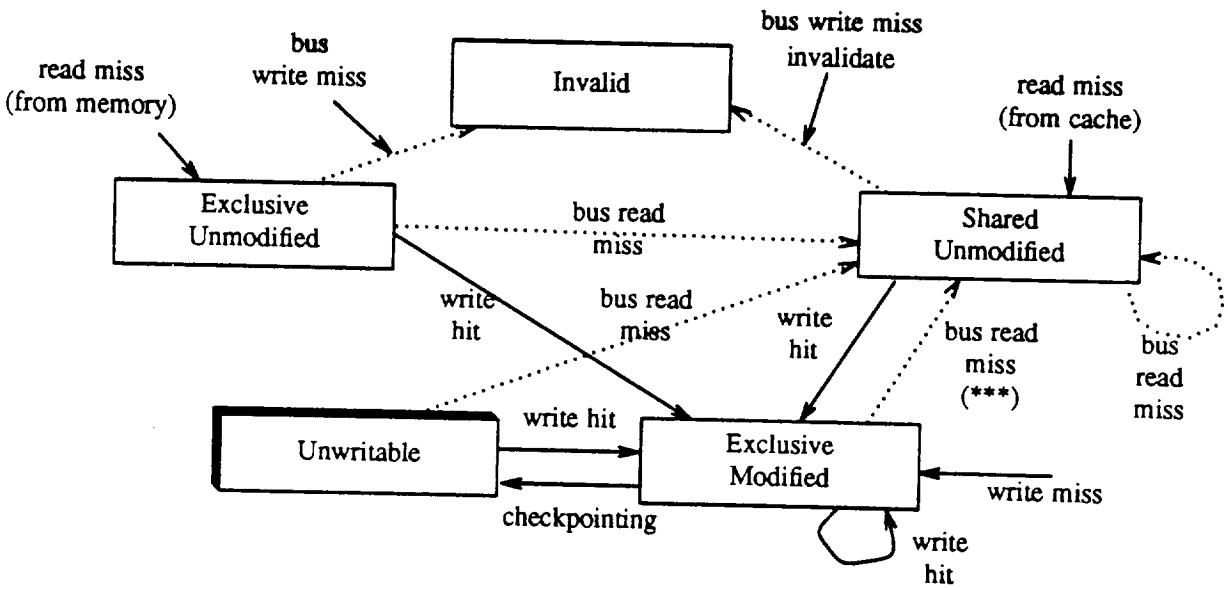
end

Implementation

- Illinois Protocol



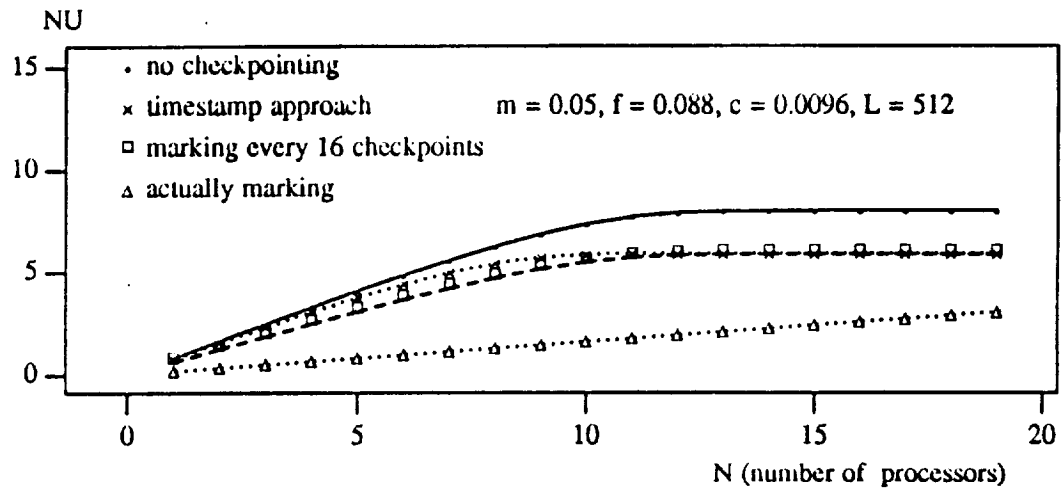
- Modified Illinois Protocol Integrating the Checkpointing Scheme

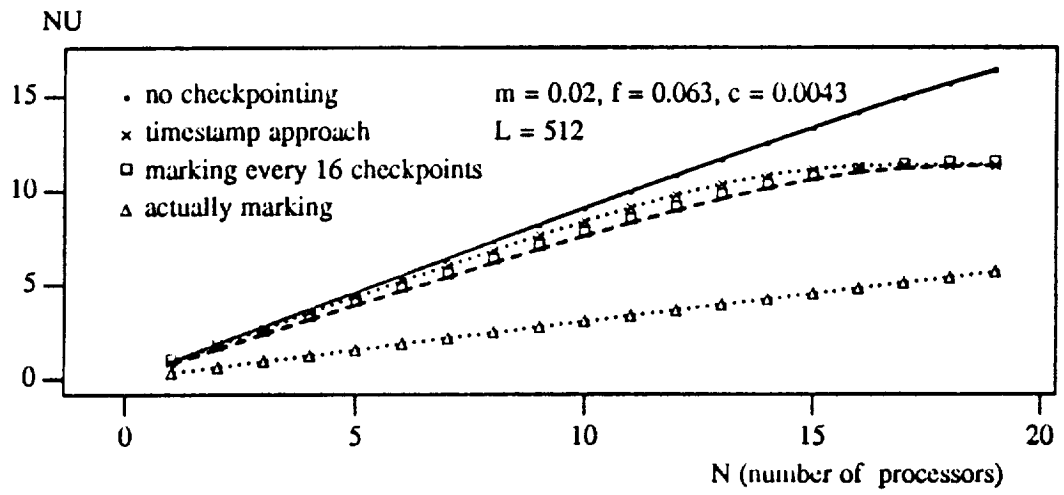


(***) a checkpointing session is initiated

Evaluation

- System Performance versus Number of processors





FORWARD RECOVERY IN LINKED DATA STRUCTURES

Objective

- To detect and correct errors to the structural information of data structures, concurrently with access

Approach

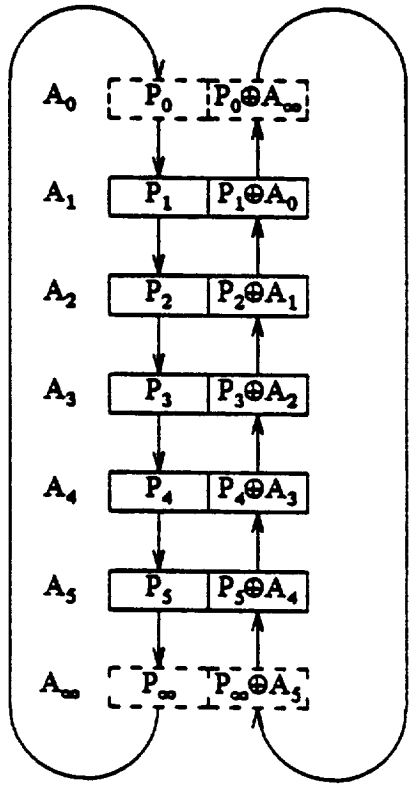
- Structural error detection/correction
- Low-overhead, constant-time concurrent checking
- Local concurrent error detection performs checking in a locality (a "checking window") of a current node
- Virtual pointers used as both backpointers and for error detection/correction

Virtual Backpointer

- $V_i = f(Q_i, A_{ancestor})$
- $A_{ancestor} = f^*(Q_i, V_i)$

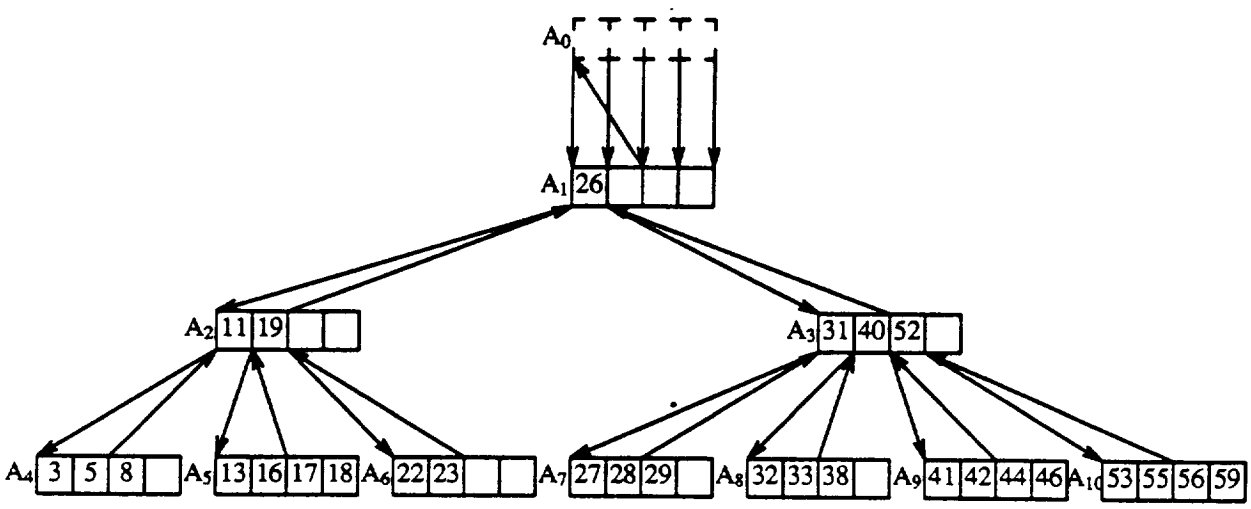
Virtual Double-Linked List (VDLL)

- Derived from double-linked list
- $V_i = P_i \oplus A_{i-1} = A_{i+1} \oplus A_{i-1}$



B-Tree with Virtual Backpointers (VBT)

- Derived from order-m B-tree
- $V_i = P_{i,0} \oplus P_{i,1} \oplus \cdots \oplus P_{i,2}m \oplus A_{parent,j}$



Evaluation

- Detection Latency

  - Sequent Balance 8000 (6 processors),
    4-users, VDLL-based DBMS system;
    Concurrent Auditor Process reduced
    latency by factor of 5

  - Sun 3/50,
    1-user, 1 VBT;
    Work in progress

- Detection and Correction Capability

  - Sun 3/110,
    1-user, 1 VDLL;
    5% erroneous pointers: 100% detection, 90% correctio
    30% erroneous pointers: 100% detection, 60% correctio
    Performance Characteristics

  - Network of Sun 3/110,
    Multi-user, VDLL- and VBT- based DBMS system;
    Work in progress

RELIABLE GARBAGE COLLECTION

Objective- reliable reclamation of memory

- Collect *only* garbage
- Collect *all* desired garbage
- Avoid corruption of objects
- Recovery in the event of a crash during garbage collection

Approach (also see poster)

- Reference count technique
- Arbiter used for resolving interprocess references
- Distributed checkpoints
- Use garbage collection to check structural integrity

CURRENT WORK

Cache-based recovery in shared memory multiprocesso

- Extended failures
- Hierarchical memory structures

Forward recovery in linked data structures

- Distributed implementation
- Evaluation

Reliable parallel/distributed garbage collection

- Detailed recovery procedures
- Version management- historical objects

PERFORMABILITY MODELING BASED ON REAL DATA
A CASE STUDY

M.C. Hsueh

Computer Systems Group
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

OUTLINE

- Objective
- Motivation/Related Research
- Approach
- Resource-usage/error/recovery model
  - model construction
  - model analysis
- Performability model
- Concluding Remarks

MOTIVATION

- Analytical models of hardware failures well investigated [Meyer 82, Trivedi 86]

- Most of current analytical models analysis are inadequate to represent the reliability of a system

- No systematic approach for modeling internal behavior from real data

- Only external behavior has been modeled based on the assumption of cyclostationary Gaussian process [Castillo 82]

- Developing models from real data valuable:
  - ☐ provide new insight into failure characterization
  - ☐ quantify interaction between different system components

APPROACH

- Instrument the system to collect data on error occurrences and recovery information (if nay)
- Employ clustering methods to identify important error states for the operational environment
- Determine types of recovery employed and their relationship to error states
- Use statistical methods to determine sojourn times in error and recovery states
- Evaluate key model characteristics

WORKLOAD MEASUREMENTS

- Workload Measurements
  - ☐ CPU utilization
  - ☐ Channel utilization and CPU wait
  - ☐ Channel start I/O rate
  - ☐ Disk I/O service requests rate
  - ☐ Paging rate

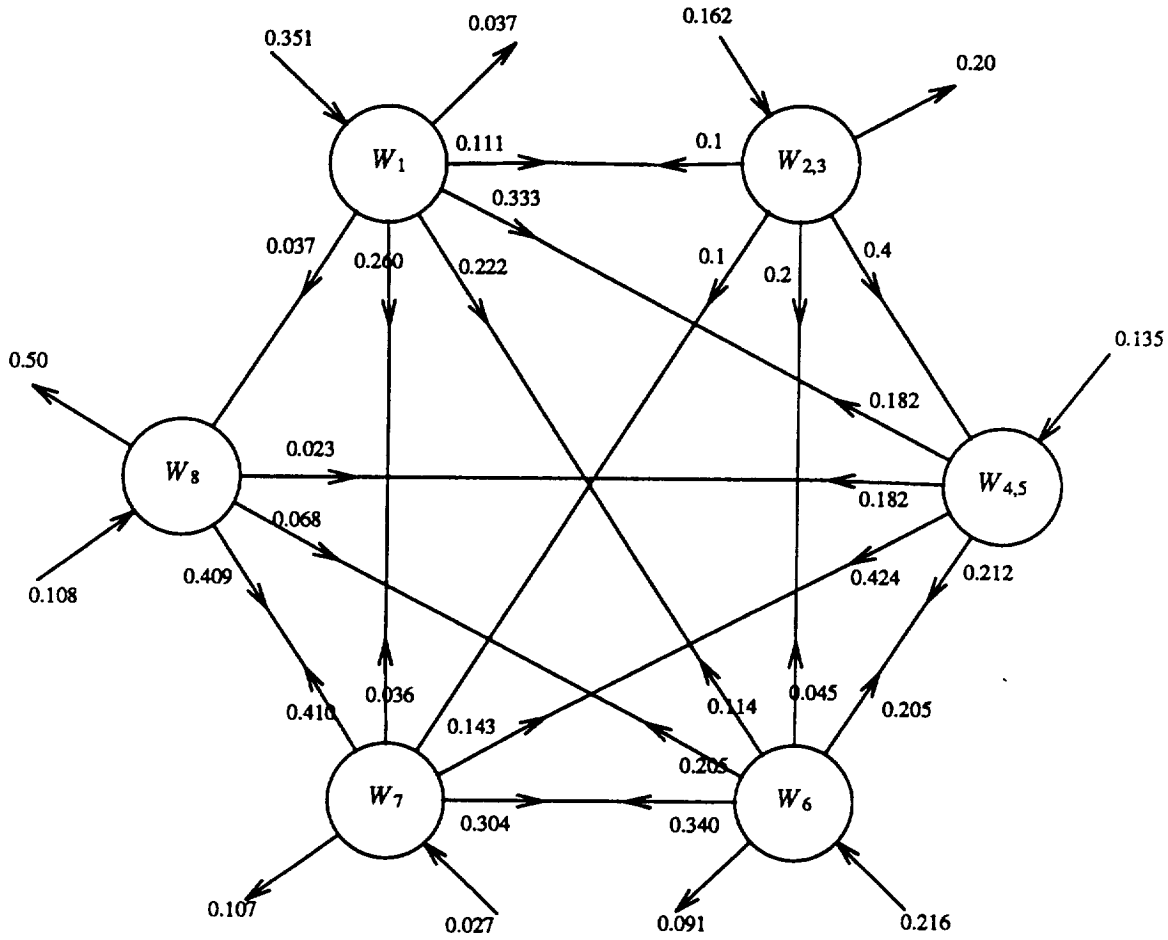
WORKLOAD CLUSTERS

- CPU-bound load

Cluster id	% of obs	Mean of CPU	Mean of CHB	Std dev of CPU	Std dev of CHB
W_1	7.44	0.0981	0.1072	0.0462	0.0436
W_2	0.50	0.1126	0.5525	0.0433	0.0669
W_3	2.73	0.1547	0.2801	0.0647	0.0755
W_4	12.41	0.3105	0.1637	0.0550	0.0459
W_5	0.74	0.3639	0.3819	0.0365	0.1923
W_6	17.12	0.5416	0.1287	0.0560	0.0511
W_7	22.58	0.7207	0.0848	0.0576	0.0301
W_8	36.48	0.9612	0.0168	0.0362	0.0143

R^2 of CPU = 0.9724
R^2 of CHB = 0.8095
overall R^2 = 0.9604

WORKLOAD MODEL

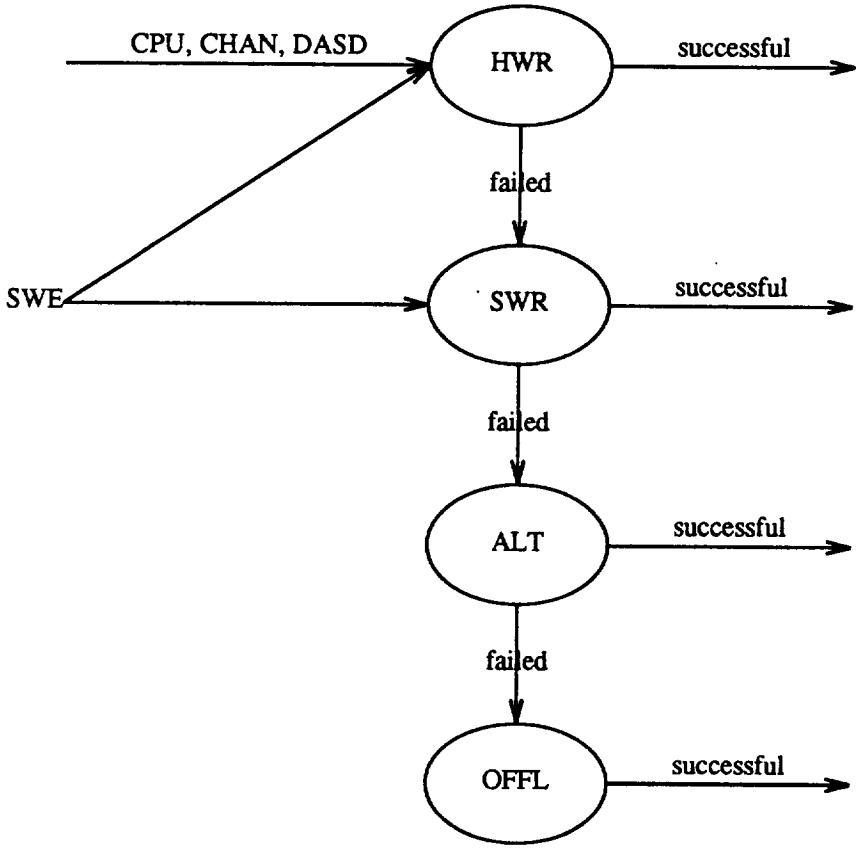


ERROR MEASUREMENTS

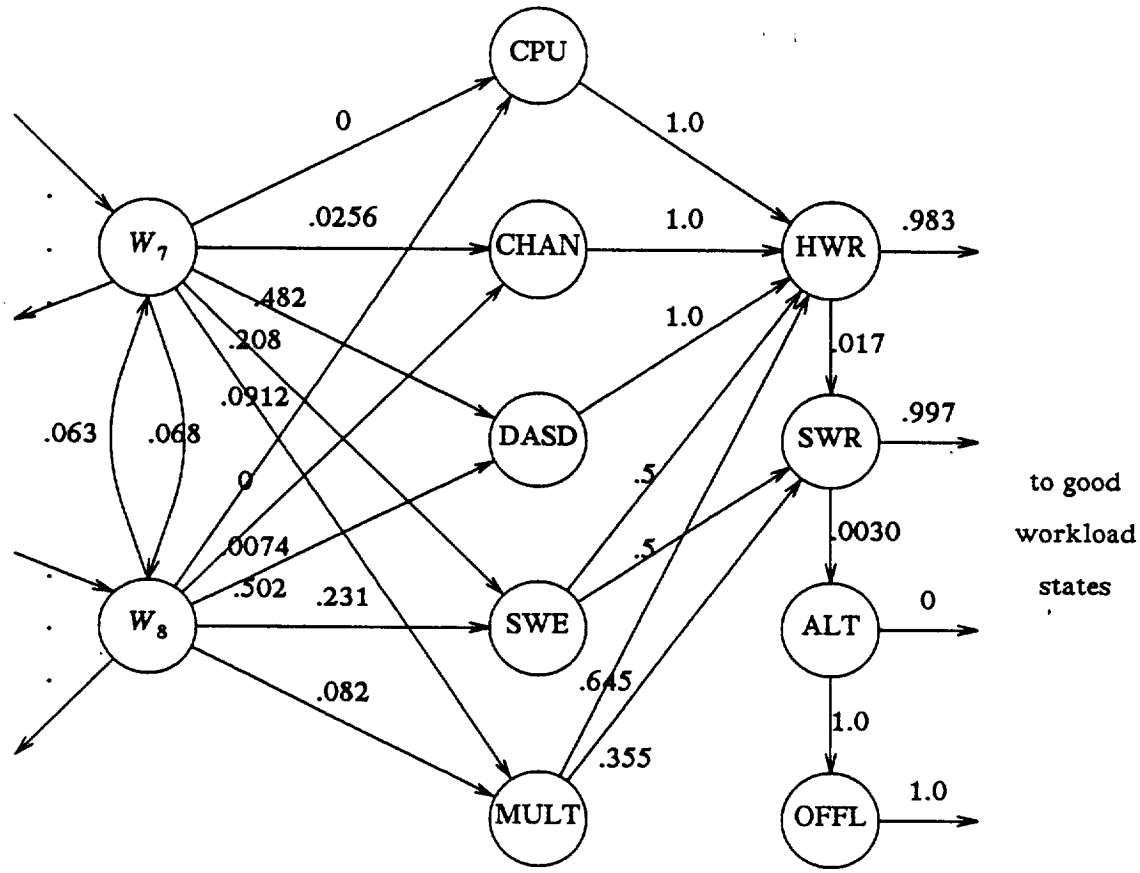
- Error Types
  - ☐ CPU error
  - ☐ Channel error
  - ☐ Software error (e.g., divide by zero, endless loop and program exceptions)
  - ☐ Disk error
  - ☐ Multiple error (errors affecting more than one type of component)

- Recovery Modes
  - ☐ Hardware recovery (HWR) — hardware instruction retry, ECC
  - ☐ Software recovery (SWR) — software retry, functional recovery routines
  - ☐ Alternative recovery (ALT) — alternative processor
  - ☐ Off-line repair (OFFL)

RECOVERY MODEL

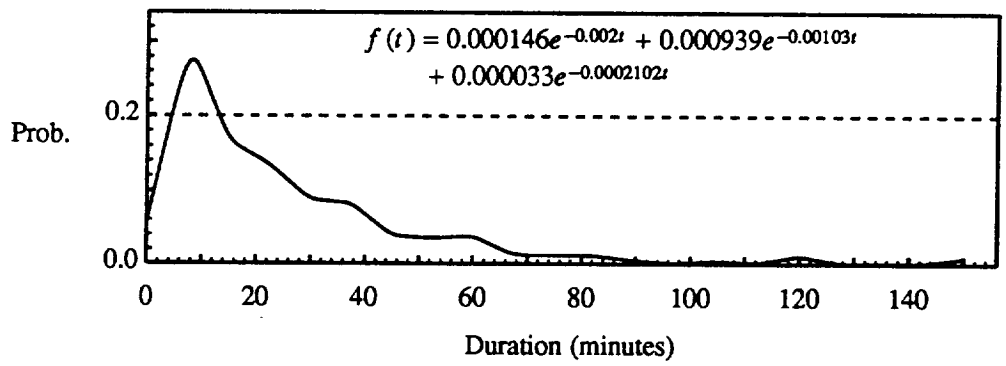


RESOURCE-USAGE/ERROR/RECOVERY MODEL

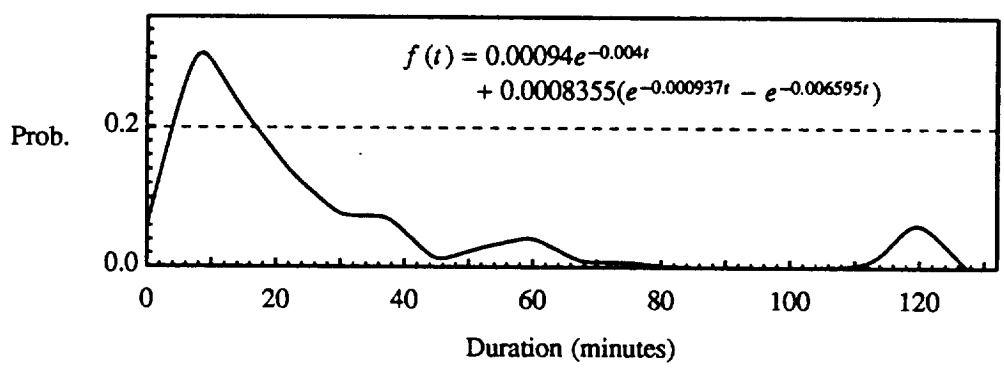


R-U/E/R MODEL (cont.)

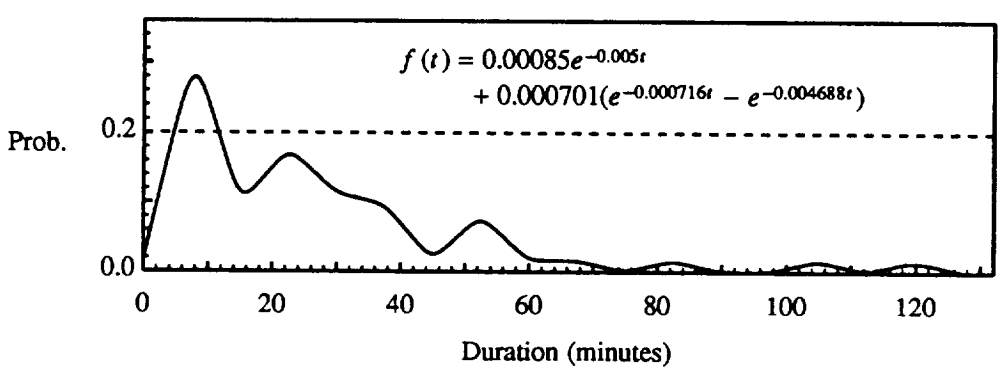
- Waiting and holding time densities for a CPU load state



Waiting time density of state W_8 (CPU = 0.96)



Holding time density from state W_8 to state DASD



Holding time density from state W_8 to state SWE

SUMMARY of R-U/E/R MODEL

- Workload dynamics explicitly described
- Error/recovery explicitly modeled
- Waiting time in some workload and error states not simple exponentials
- Transition from workload state to error states depend on destinations
- Resource-usage/error/recovery process modeled as *irreducible semi-Markov* process (OFFL recurrent)

EFFECT OF WORKLOAD

- CPU workload

CPU Load	Error state								Total
	CHAN		SWE		DASD		MULT		
	Time	Prob	Time	Prob	Time	Prob	Time	Prob	Prob
0.96	668.18	0.0011	1609.71	0.0786	1218.62	0.1296	1641.20	0.0285	0.2377
0.72	596.28	0.0032	1118.12	0.0492	971.62	0.0990	757.09	0.0146	0.1661
0.54	1304.96	0.0010	1507.92	0.0471	1070.10	0.0489	722.26	0.0052	0.1027

Time - in seconds.

- Summary
  - ☐ higher workload results in a higher error probability
  - ☐ error probability increases with increased holding time

MODEL VALIDATION

- Φ_i — occupancy probability evaluated from the model
- $\hat{\Phi}_i$ — actual occupancy probability estimated from observed data

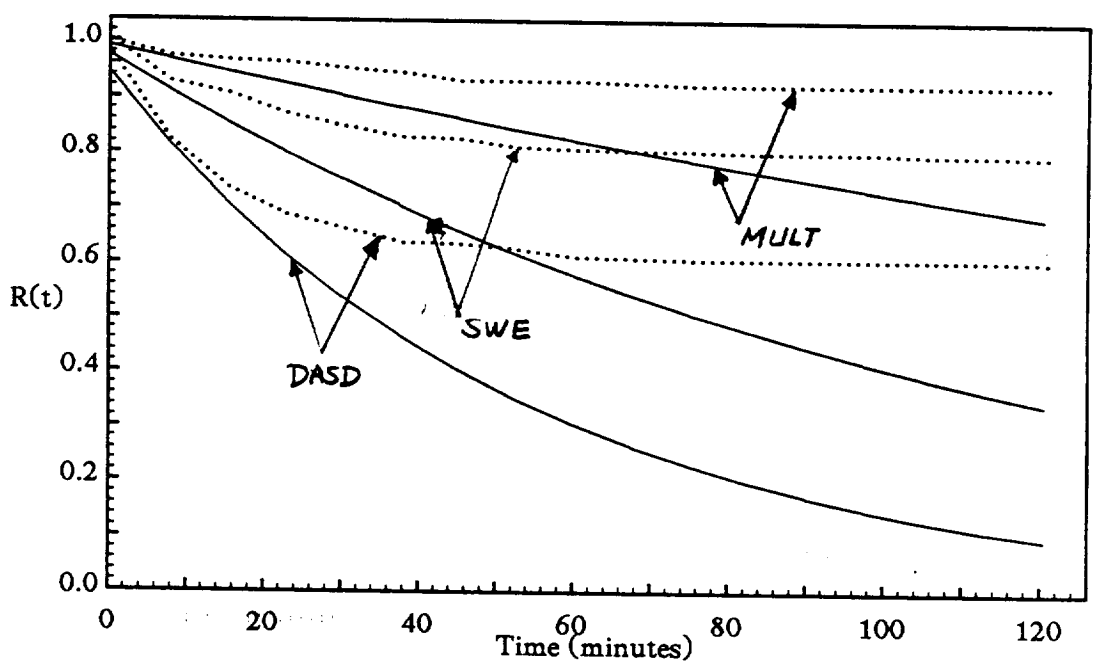
State	W_4	W_6	W_7	W_8	DASD
Φ	0.1258	0.1639	0.2255	0.3398	0.0383
$\hat{\Phi}$	0.1259	0.1634	0.2311	0.3452	0.0386
ϵ	0.0001	0.0005	0.0056	0.0054	0.0003
$\frac{\epsilon}{\hat{\Phi}}$	0.0008	0.0031	0.0242	0.0156	0.0078

ϵ : the absolute error, $|\Phi - \hat{\Phi}|$

- Semi-Markov assumption provides a fairly accurate prediction

SEMI-MARKOV VS. MARKOV

- Markov process is
  - ☐ independent semi-Markov process
  - ☐ exponential distribution in each state
- Reliabilities of two models



solid line — Markov

dotted line — semi-Markov

SUMMARY of MODEL ANALYSIS

- Semi-Markov model provides a fairly accurate prediction.

- Constant error (or failure) probability assumption may not be valid because that
  - ☐ higher resource-usage results in higher error probability
  - ☐ longer holding time also results in higher error probability

- Markov process assumption results in
  - ☐ inaccurate estimation of the reliability

REWARD FUNCTION

- Reward rate for each state i

$$r_i = \begin{cases} \dfrac{s_i}{s_i + e_i} & \text{if } i \in S_N \cup S_E \\ 0 & \text{if } i \in S_R \end{cases}$$

where s_i : service rate of state i
e_i : error rate of state i

- Calculated Reward rates for different error states

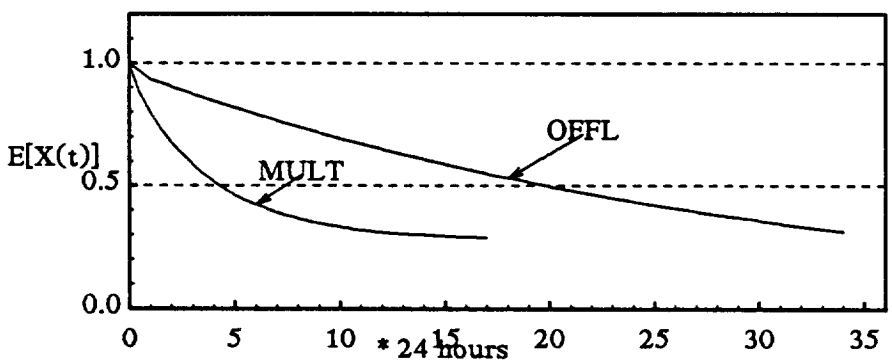
State	DASD	SWE	CHAN	MULT
r_i	0.5708	0.2736	0.9946	0.2777

PERFORMABILITY ANALYSIS

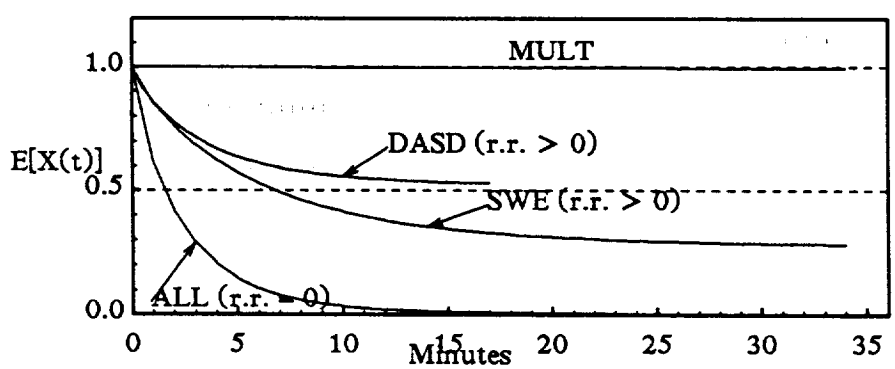
- Reward rate at time t, $X(t)$

$$X(t) = \begin{cases} r_i & \text{the process is in state } i \text{ at time } t \\ 0 & \text{otherwise} \end{cases}$$

- The expected reward rate at time t (using SHARPE)



(a).



(b)

SUMMARY of PERFORMABILITY ANALYSIS

- Explicitly described the performability model based on the error data

- Investigated the impact due to different errors
  - SWE error degrades the system performance more severely than the DASD error
    - due to the lower reward rate

CONCLUSIONS

- Established a methodology to construct a workload/performability model based on real data

- Modeled both normal and erroneous behavior

- Investigated the impact due to different errors on the system reliability and performance

- Can be used as an adaptive tuning tool during design phase

FAULT-TOLERANT MATRIX COMPUTATIONS ON PROCESSOR ARRAYS

V. S. S. Nair

J. A. Abraham

Computer Systems Group
Coordinated Science Laboratory
University of Illinois
Urbana, IL-61801

ALGORITHM BASED FAULT-TOLERANCE

- Low cost techniques
- Applicable to processor arrays.

PRIMARY CHARACTERISTICS

- Encode data at a high level.
- Design algorithm to operate on encoded input data and produce encoded output data.
- Distribute computation tasks among multiple computation units, so that failure of a unit affects only a portion of the output data, enabling the correct data to be recovered.

APPLICATION TO MATRIX OPERATIONS

PREVIOUS WORK

- Checksum Matrix Encoding (CC)
- Weighted Checksum Encoding (WCC)

Checksum Encoding Example.

$$A = \begin{bmatrix} 2 & 3 & 5 \\ 6 & 7 & 8 \\ 9 & 10 & 11 \end{bmatrix} \qquad A_c = \begin{bmatrix} 2 & 3 & 5 \\ 6 & 7 & 8 \\ 9 & 10 & 11 \\ 17 & 20 & 24 \end{bmatrix}$$

$$A_r = \begin{bmatrix} 2 & 3 & 5 & 10 \\ 6 & 7 & 8 & 21 \\ 9 & 10 & 11 & 30 \end{bmatrix} \qquad A_f = \begin{bmatrix} 2 & 3 & 5 & 10 \\ 6 & 7 & 8 & 21 \\ 9 & 10 & 11 & 30 \\ 17 & 20 & 24 & 61 \end{bmatrix}$$

NUMERICAL PROBLEMS

- Roundoff Errors
- Overflow/Underflow errors
- These errors may be misconstrued as an error due to a physical fault.

Previous solutions for roundoff errors

- Define a bound and treat all errors less than the bound as numerical errors.
  - ☐ The bound depends on the data set.
  - ☐ If the bound is small, it leads to incorrect fault detection.
  - ☐ If the bound is high, the fault detectability will be poor.

OVERFLOW PROBLEMS

- When computing row/column checksums, we are storing sums of **n** elements.
- The checksum element, (n+1,n+1) can be as large as n^2 times the largest element in the original matrix.
- Need increased word size for overflow protection.

Modulo Arithmetic to prevent Overflow

- For integers perform residue Arithmetic with modulus $M = 2^r$

$$a_{n+1,j} = \Sigma\, a_{i,j} \; mod \; M$$

- Encoding still holds.
- But not possible for floating point numbers.

AIM

- To keep the numerical errors as small as possible.

Solution

- Identify various stages which are prone to numerical errors in the fault-tolerance technique.
- Devise techniques to reduce the error in each stage.
- Stages identified are
  - ☐ Coding phase
  - ☐ Actual data computation phase
  - ☐ Check element computation phase.

CODING PHASE

- It is impossible to find a coding technique which will give minimal numerical error independent of the data set on which it is applied.
- The approach should be to choose different algorithms for different set of data.
- Instead of confining the fault-tolerant encodings to the checksums and the weighted checksums we may use different encoding schemes for different sets of data.

GENERAL LINEAR CODES (GLC)

DEFINITION 1: An encoder vector is a vector whose inner product with a column/row vector will produce a column/row check element.

- A valid Encoder vector will produce check elements whose properties will be preserved during matrix multiplication, addition, transposition and LU-decomposition.

THEOREM 1: An Encoder vector is a valid encoder vector if and only if it is a linear operator.

ERROR ANALYSIS

- Maximum roundoff error $\leq u\, m \;||\, a \,||_2 \;||\, x \,||_2$ where u is the machine dependent unit round off error and m is the mantissa, $|\, m \,| \leq 1$.

- If we select $||\, a \,||_2$ to be very small, the absolute error in the code element will be small, and observability of the actual error at the output will be very poor.

- As a compromise select a constant such that
$$C = ||\, a \,||_2 \;||\, x \,||_2.$$

- When $||\, x \,||_2$ is high select a small $||\, a \,||_2$ and vice versa.

EXAMPLE CODES

- Average Checksum Codes
- Weighted Average Checksum codes
- Codes produced by Periodic Encoder vectors
- Codes produced by Normalized Encoder vectors.

AVERAGE CHECKSUM ENCODING (ACC)

- Instead of summing the column/row vector, find the average

Example

$$A = \begin{bmatrix} 2 & 3 & 4 \\ -1 & 6 & -2 \\ 2 & 0 & 1 \end{bmatrix} \qquad A_c = \begin{bmatrix} 2 & 3 & 4 \\ -1 & 6 & -2 \\ 2 & 0 & 1 \\ 1 & 3 & 1 \end{bmatrix}$$

$$A_r = \begin{bmatrix} 2 & 3 & 4 & 3 \\ -1 & 6 & -2 & 1 \\ 2 & 0 & 1 & 1 \end{bmatrix} \qquad A_f = \begin{bmatrix} 2 & 3 & 4 & 3 \\ -1 & 6 & -2 & 1 \\ 2 & 0 & 1 & 1 \\ 1 & 3 & 1 & \frac{5}{3} \end{bmatrix}$$

PERIODIC ENCODER VECTORS

Advantages

- Less memory space required to store the encoder vector elements.
- Easy expansion of the system.
- If the average weight of the encoder vector is zero, there will not be any overflow error during coding.

EXAMPLE 1: $[1 \; -1 \;\; 1 \; -1 \; \ldots]$

EXAMPLE 2: $[\cos\theta \;\; \cos 2\theta \;\; \ldots \; \cos n\theta]$

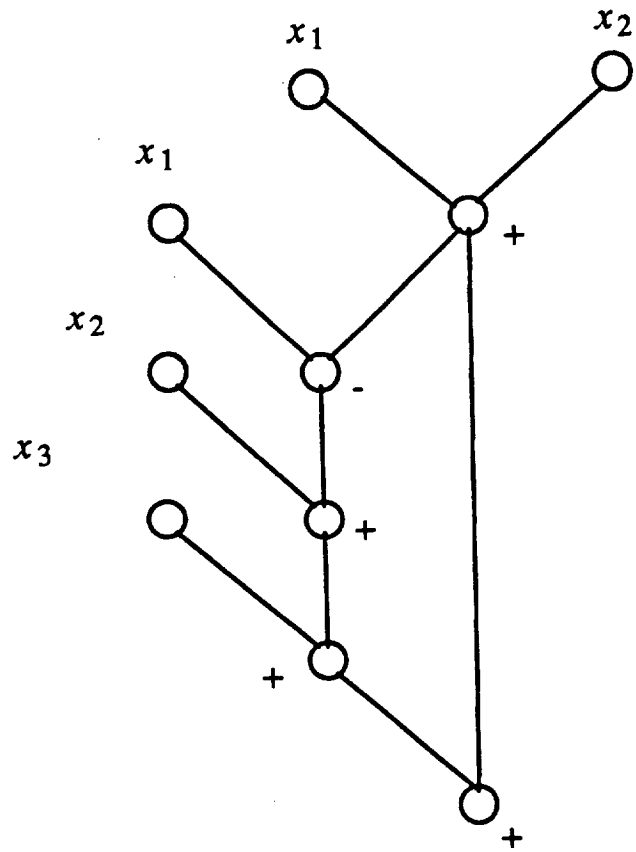
CHECK ELEMENT COMPUTATION PHASE

- Modify the algorithm or/and hardware to reduce the numerical errors.

Algorithm modification

- When the dimension of the matrices are small use binary type addition.
- For larger dimensions (>50) use Kahan's Algorithm for adding up the elements.
  - ☐ In Kahan's method we determine an approximation r to the roundoff error during addition of two numbers a and b by computing in floating point arithmetic.
    $$r = a - (a + b) + b.$$
  - ☐ Kahan's algorithm when used with single precision, produces roundoff errors as small as that in double precision.

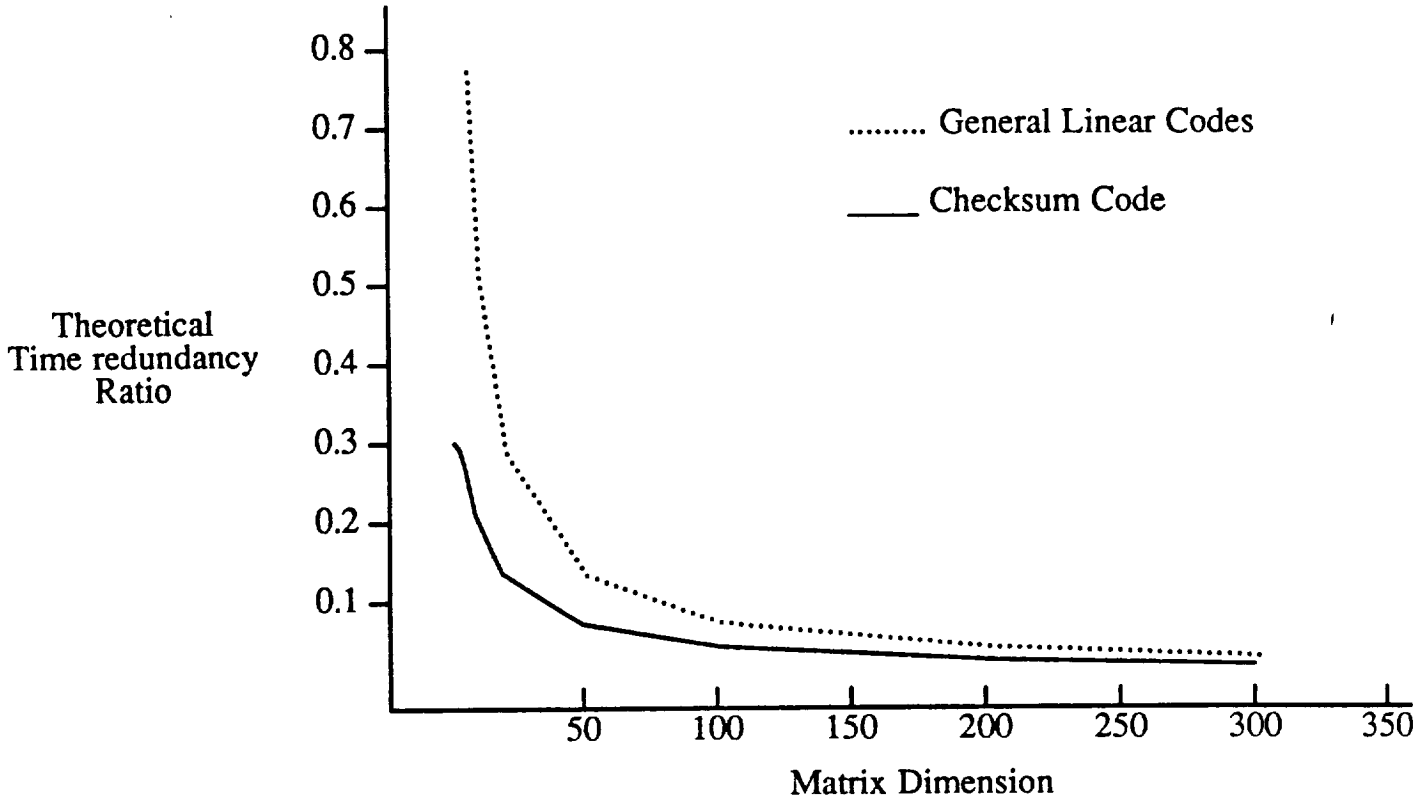
ADDITION TREE USING KAHAN'S ALGORITHM



OVERHEAD COMPARISON

- Hardware overhead in terms of number of processors is same as in the case of Checksum encoding, $(2n + 1)$. But each processor needs an extra register to hold the encoder vector element.
- Time Overhead for CC is $2 * r * \log_2 n$.
- GLC requires three additional multiplication cycles.
- For large arrays, the overhead required for both techniques is almost the same.

THEORETICAL TIME REDUNDANCY GRAPH



SIMULATION RESULTS

SIMULATION RESULTS				
Matrix Dimension	Time with out Checking	Time with Checking	TTR	OTR
2 by 2	6220	10629	58.76%	70.88%
3 by 3	8870	13976	42.59%	57.56%
4 by 4	11520	17143	33.77%	48.81%
5 by 5	14170	20189	28.14%	42.47%
6 by 6	16820	23463	24.22%	39.49%

- TTR - Theoretical Time Redundancy
- OTR - Observed Time Redundancy.
- Time is measured in number of clock ticks.

SIMULATION RESULTS

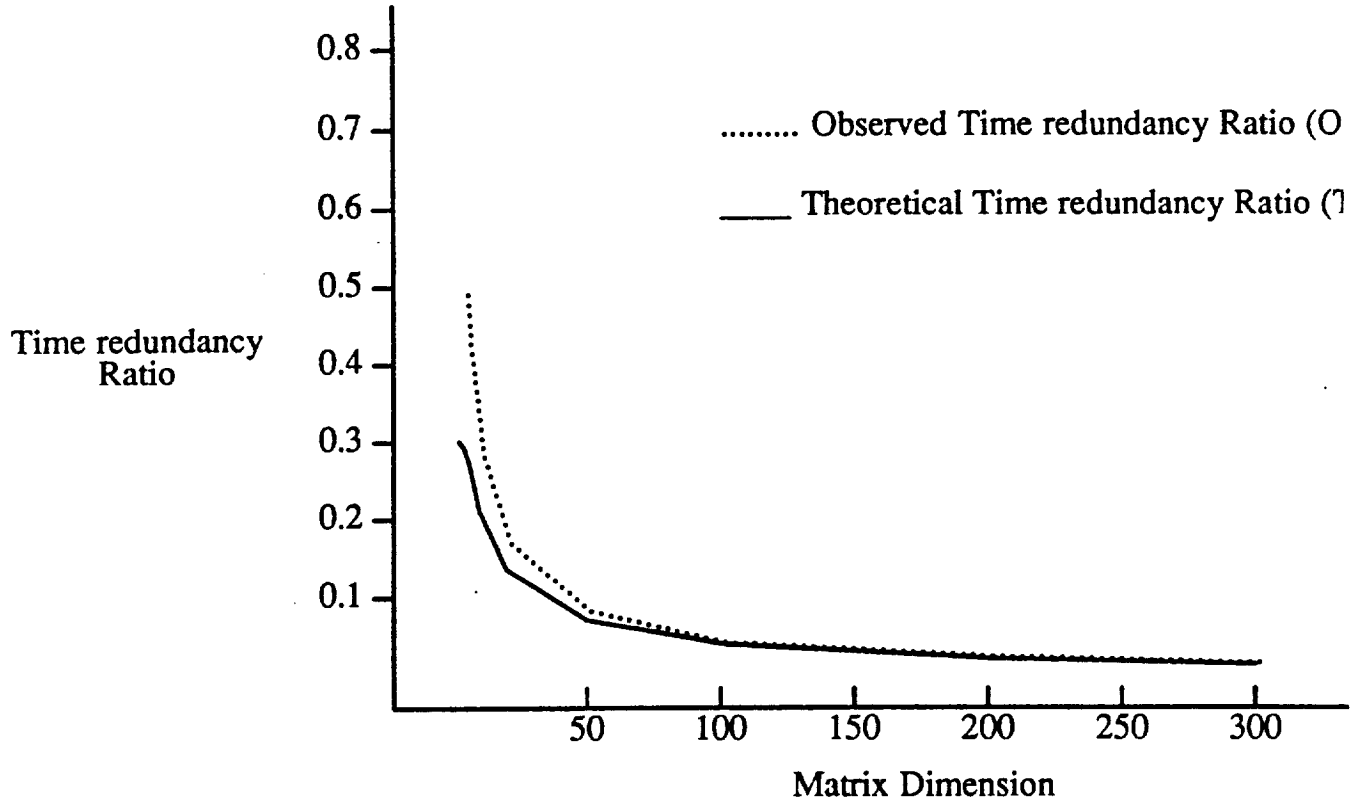
- Observed the time redundancy required for the algorithms, applied to arrays of small sizes.
- Figures obtained for computation time with out checking (C) and with checking (C ') fit very well into equations

  $C = K + l * n$

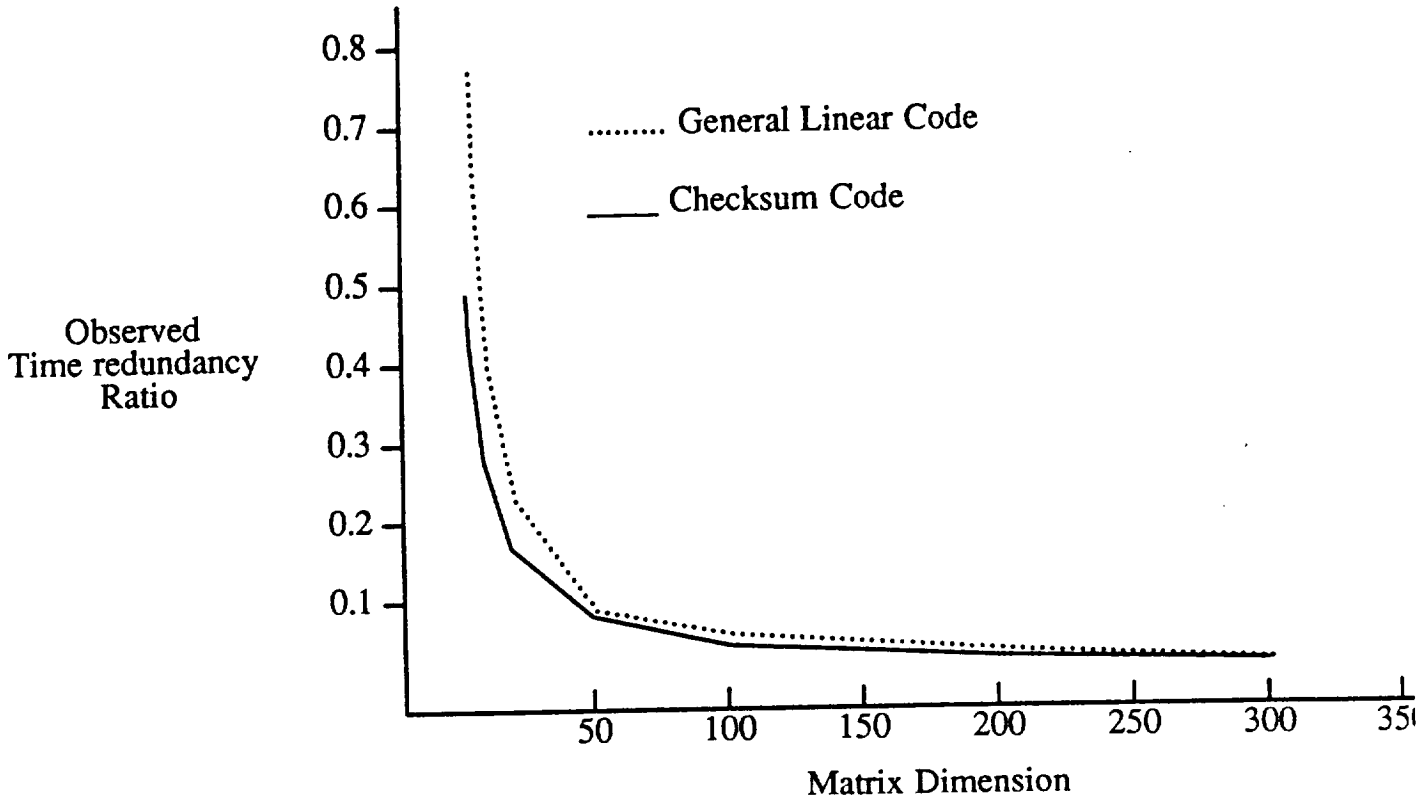
  $C' = K' + l' * \log_2 n$

  where K, K', l,l', are simulator-dependent parameters, and **n** is the size of the matrix
- Since the simulation is extremely computation intensive, the above equations were used to predict the overhead for larger numbers of processors.

TTR-OTR GRAPHS FOR CHECKSUM CODE



OTR GRAPHS FOR CC AND GLC



CONCLUSION

- Identified a general set of linear codes for fault-tolerant matrix computations.
- When roundoff is the larger problem use normalized encoder vectors. For overflow problems use Average checksum codes or codes produced by periodic encoder vectors.
- Hardware and time redundancy required for the proposed codes are of the same order as that of the previous techniques.
- Actual overhead associated with the proposed schemes found by simulation, and was observed to be quite small.

CSG PUBLICATIONS

To appear IEEE Transactions on Computers

ACCURATE LOW-COST METHODS FOR PERFORMANCE EVALUATION OF CACHE MEMORY SYSTEMS

Subhasis Laha, Janak H. Patel and Ravishankar K. Iyer

Coordinated Science Laboratory
University of Illinois
1101 W. Springfield Ave.
Urbana, IL 61801

Abstract : Trace-driven simulation is a simple way of evaluating cache memory systems with varying hardware parameters. But to evaluate realistic workloads, simulating even a few millions of addresses is not adequate and such large scale simulation is impractical from the consideration of space and time requirement. In this paper, new methods of simulation based on joint usage of statistical techniques and the principles of computer architecture are proposed for decreasing the need for large trace measurements and to predict true program behavior. In our method, sampling techniques are applied while collecting the address trace from a workload; this drastically reduces the space needed to store the trace. New simulation techniques are developed to use the sampled data to predict not only the mean miss rate of the cache, but also its true distribution. Finally, a model is proposed to statistically project the results to different context-switch intervals from only one simulation of a small number of samples of a fixed size.

Acknowledgements : This research was supported in part by the Joint Services Electronics Program (US Army, US Navy, US Air-force) under contract N00014-84-C-0149 and in part by NASA grant NAG-1-613.

ACCURATE LOW-COST METHODS FOR PERFORMANCE EVALUATION OF CACHE MEMORY SYSTEMS

1. Introduction

It is well known that the use of a cache in memory hierarchy has considerable influence on the overall performance of a computer system. Although, cache memory systems have been extensively measured and simulated for traditional high level languages like Fortran, Pascal, Algol and PL1 [1], studies on LISP programs are few and far between [2, 3, 4].

The goal of this project was to develop new techniques for cache evaluation. For various reasons, discussed in the next section, trace driven simulation was selected as the method of evaluation. But, soon it was found that conventional simulation suffers from a lot of limitations; the results, so obtained, do not represent true program behavior. Some of these aspects have already been pointed out by Smith [3]. Since results on LISP program architecture are few, LISP programs were employed in our case studies. The proposed methodology, however, is equally applicable to other programs.

This paper establishes a new method of trace driven simulation for cache memory systems, based on sampling techniques. This method decreases the need for large trace measurements and simulation, and is based on joint usage of statistical techniques and the principles of computer architecture. It is shown that, unlike conventional techniques, the proposed approach is able to characterize a complete program with low overhead. Thus, it not only allows a prediction of the *mean miss rate*, but also of its *true distribution*. In the next section, different aspects of conventional trace driven simulation is presented along with its limitations.

2. Different Methods of Evaluation

Among the various methods of performance evaluation of cache based systems, *Hardware Monitoring* is the most accurate method. But it is very difficult to instrument and it takes up a lot of time to develop. Even after implementing it for a particular machine, it is hard to vary the parameters of the architecture. *Analytical Methods* can be quiet cheap in most cases. But it involves so many simplifying assumptions that the results may be far from being true. This leaves us with *Simulation* which is simple and quiet easy to develop. It can be used to evaluate any existing or proposed architectures. The parameters of the architecture can be varied very easily simply by changing the input parameters of a simulator.

2.1. Trace Driven Simulation

A program address trace consists of a sequence of the virtual (usually) addresses, referenced by a program or programs, along with other relevant information. Trace driven simulation is the process of running the simulation model of a system with the trace. A simulator can also be driven by a random number generator; but, because of the lack of existence of an accurate behavioral model of programs in the context of cache, such simulation can never represent realistic program behavior. On the other hand, a trace represents the behavior of at least one program and so, trace driven simulation will characterize that correctly. For all these reasons, this has been chosen by most people, including us for evaluation of memory systems.

2.1.1. Limitations

In spite of all these nice features, there are several limitations of trace driven simulation which make it inadequate not only for our purpose, but also as a tool of performance

evaluation in general.

(1) It simulates only a very small part of a program. For example, if a program runs for a thousand seconds of cpu time at the rate of one million memory references per second, simulating a trace of one million addresses will only represent 0.01% of the whole program.

(2) Traces are usually collected from the initial portion of a program, rather than from the core part of it. This practice of taking a single section of a program and projecting it for the entire program is highly questionable. Smith [1, 3] has reported that an enormous degree of variability exists among traces in their measured miss ratios in cache. Our own experience with simulation of cache memories has shown that there is a large degree of variation in program behavior in one sample of half million references with another of the same program at a different time. Even if the mean miss rate, measured this way, happens to be the same as that of the whole program, this mean value is not adequate to accurately reflect the performance of the system. We will show that the distribution of this measured miss ratio is far from the real one. Such experience suggests that simulation with a section of program chosen on an ad-hoc basis with a few thousand references cannot accurately reflect the overall program behavior.

(3) Any large scale simulation with a few millions of addresses requires an enormous space to store the trace. It is also very expensive from the consideration of computer time. Each such simulation with one set of hardware parameters as input typically takes up several minutes of cpu time on a minicomputer (e.g., VAX-11/780). So, running a bunch of simulation jobs with several combinations of architectural parameters for a single trace is computationally very expensive.

(4) Most machines switch context every few thousands of instructions, especially in a multiprogramming environment. The Context switch can occur due to I/O interrupts or time out of a fixed time slice. This effect of context switch on cache performance has been studied by others [5, 6, 7]. But, their methods involve simulating every different switch intervals or a curve fitting on individual program traces. Our objective is to perform only one simulation, based on our sampling technique and statistically project the results for other switch intervals.

To overcome these limitations, we propose a new sampling-based technique for studying cache performance in LISP systems. The methodology is based on statistical methods and is capable of characterizing a program without extensive simulation.

3. Sampling Technique

In conventional simulation, the effect of task switches on a program can be estimated by assuming the cache to be flushed totally during a context switch [5, 8]. This is essentially true for smaller caches. For larger caches, the miss ratio obtained in this way provide a lower limit to the cache performance. The preliminary aim of our sampling technique is to provide a cost-effective approach to estimate the distribution of this cold-start miss ratio.

Instead of simulating all consecutive intervals of address references, as is done conventionally, we sample the address trace (Fig. 1) and run the simulation only on the sampled data.

Our Methodology

(1) Choose a sample size which is typically a few thousands of consecutive addresses. In the initial study, this corresponds to the task interval considered.

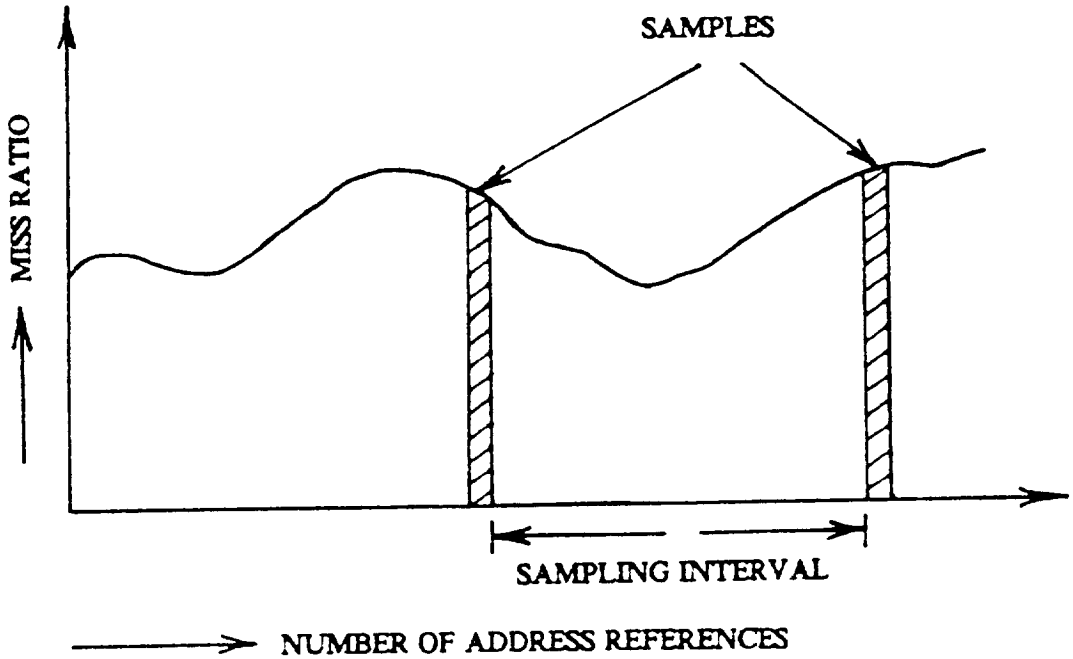


Fig. 1. Sampling technique

(2) Depending on the size of the whole trace, determine the uniform sampling interval to attain a certain number of samples.

(3) Sample the continuous trace accordingly (or, preferably collect only the sampled part of the trace at the beginning itself) and store the sampled addresses only.

(4) Assume cache to be empty at the beginning of each sample and simulate cache memory system for each sample.

Claim

The distribution of the average miss ratio for the samples gives an estimate of that of the cold-start miss ratio over the entire trace for the same context switch interval as the sample size.

Validation Procedure

Simulating one sample of consecutive addresses represents the behavior of that particular context switch interval. Because of the cold-start, the miss rate is very high at the beginning of the interval and usually decreases as the cache gets more and more filled up. We consider the average value of the miss ratio at the end of the sample. To validate this methodology, it is essential to establish that the miss ratios of the samples truly represent the behavior of the continuous trace. Our objective is to determine the number of samples which is adequate to represent any program, irrespective of the nature of the program or the length of the trace. We will also show that the number of samples is the factor to be considered for sampling, rather than the sampling interval. Therefore, to validate this technique, different number of samples are considered over different LISP programs and statistical techniques are applied to compare the distribution of the sampled data with that of the actual distribution of the miss ratio. In the next section, different aspects of the simulations performed are outlined.

4. Simulations Performed

4.1. Traces Used

The traces of virtual addresses were collected by running the compiled code of *Franz LISP programs* on a VAX-11/780. These are continuous traces which were later sampled at various intervals and sizes and then simulated. The programs chosen are realistic workloads of varied nature. The nature of the programs and the lengths of the corresponding traces are listed below:

(1) FSIM : Fault Simulation program : Length $\simeq$ 2.9 million.

(2) DRC : Design Rule Check program for VLSI layout : Length $\simeq$ 5.7 million.

(3) ELI : English Language Interpreter program : Length $\simeq$ 1.8 million.

(4) RRL : Mathematical Rewrite Rule Lab program : Length $\simeq$ 2.0 million.

These traces are collected by running only part of these programs because of practical limitations. But they are of varied length and we will show that our methodology holds good independent of this variation of trace length, so that this method can be applied to traces of entire programs also.

4.2. Size and Number of Samples

Three different sample sizes were used : 5K, 10K and 20K of addresses. They are chosen to be of the same order as the typical task interval on VAXes [9].

It is known that about 30 samples give a good estimate of any normally distributed data. At the beginning, we started with the assumption that the distribution of the cache miss ratio is normal which is found later not to hold good in general. But, nonetheless, the numbers of samples selected are (1) about 20, (2) about 35. The results of these two sampled cases are compared with that of the continuous trace. We will show that about 35 samples are always found adequate to predict the actual miss distribution with reasonable

accuracy.

4.3. Architecture Simulated

The architectural model has three level memory hierarchy : virtual, main and cache. We consider *real address cache*: the virtual address is first translated to real address of the main memory which is subsequently mapped to the cache address space.

The main memory is organized into pages. The physical page frames are numbered sequentially as they are allotted. FIFO (First In First Out) policy is used for page replacement. The cache memory is *set-associative* and its blocks are replaced according to *Least Recently Used (LRU)* policy. Cache is updated by *no-allocate* policy i.e., during a write-miss, only the main memory is updated and it is not counted as a cache miss [10].

The goal of this paper is to develop a new method of simulation which is applicable to all LISP programs, rather than to evaluate the behavior of LISP programs for various architectures. For that reason, it is sufficient to simulate this architecture only with one set of hardware parameters. The sizes chosen for the three levels of memory, virtual, main and cache are respectively 64KB, 16KB and 4KB. The page size in the main memory is 128 bytes. The block size and set size of the cache are kept fixed at 8 bytes and 2.

4.3.1. Approximation

Both the instructions and the data in VAX-11 are of variable length. It has been shown in other studies [9] that most of the times, these lengths are within 4 bytes . Also, the VAX machine always fetches a block of 4 bytes together from the memory. So, it is quite reasonable to approximate the length of all the references to 4 bytes. This simplifies the process of gathering the trace and others [4, 11] have also made similar assumptions . During this simulation, if the address referenced fits exactly in an integral 4-byte boundary, only one block (of size at least 4 bytes) is fetched. Otherwise if the 4-byte field,

starting from the current address in the trace, does not coincide with a 4-byte boundary and it also surpasses the end of the current block of the cache, then the next block is also fetched. In the next section, the results of simulation are presented and discussed.

5. Results

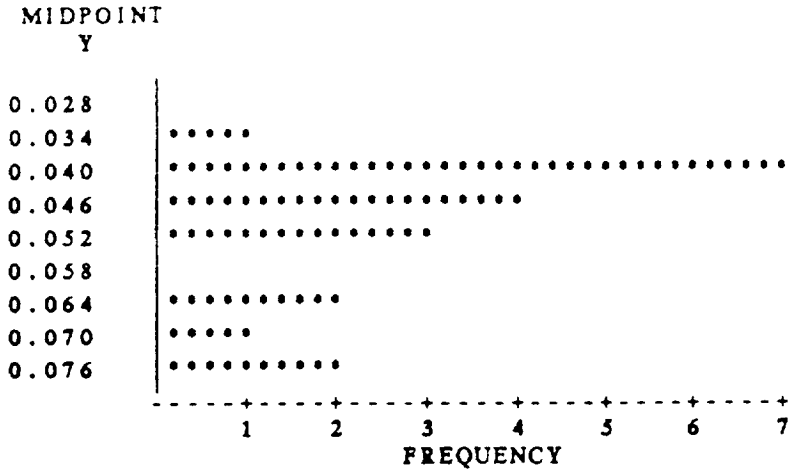
Recall that the goal is to show that a sampling scheme can successfully predict not only the true mean value but also the distribution of miss ratio. Note that it is assumed that a program is running with context switches. To establish the effectiveness of the sampling scheme, we compare the distribution of the sampled data with that of the continuous trace. As a first step, the mean and the standard deviation of the miss ratios of sampled data are compared with actual values (Table 1). The term "all" for the number of samples in the table signifies the results for the continuous trace without any sampling. It is found that the maximum error in mean miss ratio for number of samples $\simeq$ 35 is 5.5%; this error for about 20 samples can be as high as 13.8%. The standard deviation of miss ratio for about 35 samples is always very close to its actual value. So, just from the consideration of mean and standard deviation of the samples, it can be concluded that with approximately 35 samples a good estimate of the mean and the standard deviation is achieved.

The distribution of the continuous trace is compared with that obtained from sampled trace in Figures 2 through 5. All these figures contain the histograms for the case of about 35 samples and the continuous trace. Figures 2 and 3 also include the case for number of samples $\simeq$ 20. Of all the 12 cases considered, only four are given in these figures; remaining of them appear in the Appendix. It is very apparent from these figures that in general, the distribution profile is far from normal. Initially, these distributions can be just checked for similarity in appearance. The cumulative percentages of the sample points in the different levels, included in the figures, provide additional way of comparing the distribu-

Table 1. Mean & Standard Deviation of Sampled Result

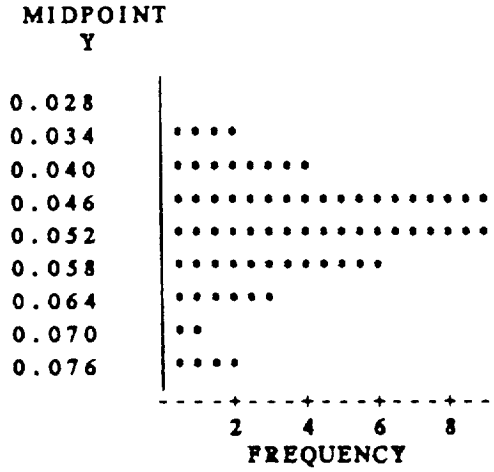
Program	Sample size	Mean miss ratio (for #samples)			%Error in mean (for #samples)		Std. dev. of miss ratio (for #samples)		
		≃ 20	≃ 35	All	≃ 20	≃ 35	≃ 20	≃ 35	All
FSIM	5K	.0743	.0767	.0781	4.9	1.8	.0158	.0177	.0160
	10K	.0576	.0609	.0600	4.0	-1.5	.0137	.0115	.0131
	20K	.0492	.0513	.0505	2.6	-1.6	.0125	.0098	.0107
DRC	5K	.0768	.0639	.0675	-13.8	5.3	.0539	.0443	.0430
	10K	.0630	.0550	.0582	-8.2	5.5	.0425	.0386	.0389
	20K	.0576	.0512	.0528	-9.1	1.9	.0389	.0349	.0353
ELI	5K	.0819	.0775	.0792	-3.4	2.1	.0452	.0379	.0406
	10K	.0680	.0691	.0673	-1.0	-2.7	.0359	.0351	.0343
	20K	.0606	.0616	.0604	-0.3	-2.0	.0295	.0311	.0307
RRL	5K	.0424	.0417	.0416	-1.9	-0.2	.0354	.0332	.0329
	10K	.0349	.0356	.0346	-0.9	-2.9	.0283	.0293	.0286
	20K	.0318	.0315	.0308	-3.2	-2.3	.0267	.0258	.0255

(a) Number of samples = 20



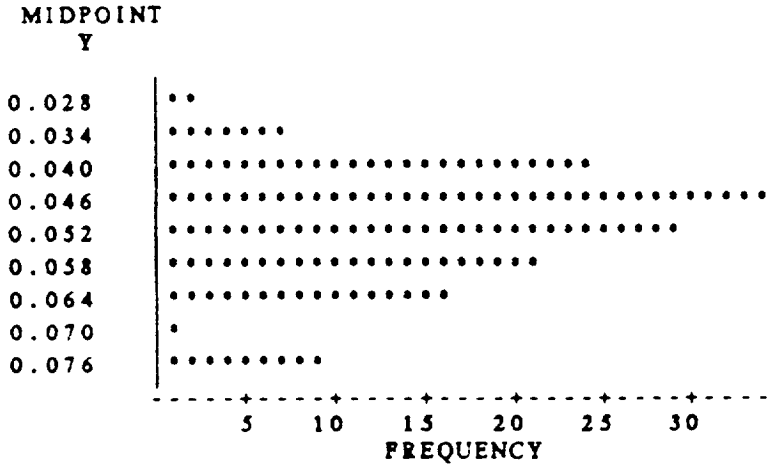
MIDPOINT Y	FREQ	CUM. FREQ	PERCENT	CUM. PERCENT
0.028	0	0	0.00	0.00
0.034	1	1	5.00	5.00
0.040	7	8	35.00	40.00
0.046	4	12	20.00	60.00
0.052	3	15	15.00	75.00
0.058	0	15	0.00	75.00
0.064	2	17	10.00	85.00
0.070	1	18	5.00	90.00
0.076	2	20	10.00	100.00

(b) Number of samples = 36



MIDPOINT Y	FREQ	CUM. FREQ	PERCENT	CUM. PERCENT
0.028	0	0	0.00	0.00
0.034	2	2	5.56	5.56
0.040	4	6	11.11	16.67
0.046	9	15	25.00	41.67
0.052	9	24	25.00	66.67
0.058	6	30	16.67	83.33
0.064	3	33	8.33	91.67
0.070	1	34	2.78	94.44
0.076	2	36	5.56	100.00

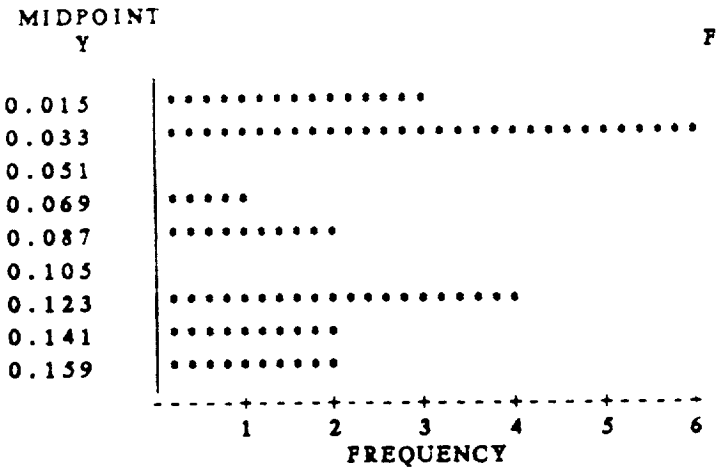
(c) Continuous Trace



MIDPOINT Y	FREQ	CUM. FREQ	PERCENT	CUM. PERCENT
0.028	2	2	1.40	1.40
0.034	7	9	4.90	6.29
0.040	24	33	16.78	23.08
0.046	34	67	23.78	46.85
0.052	29	96	20.28	67.13
0.058	21	117	14.69	81.82
0.064	16	133	11.19	93.01
0.070	1	134	0.70	93.71
0.076	9	143	6.29	100.00

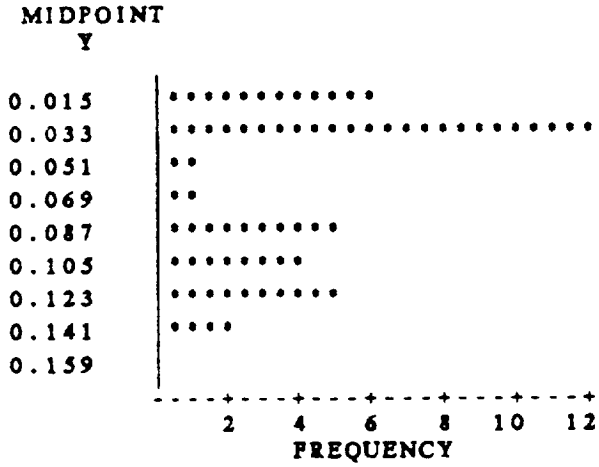
Fig. 2. Frequency bar chart of miss ratio for FSIM : Context-switch interval = 20K

(a) Number of samples = 20



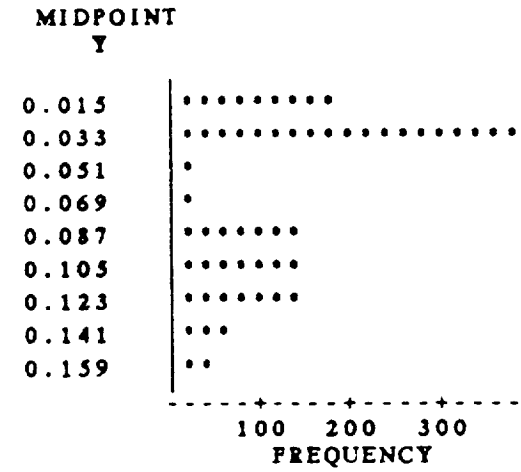
MIDPOINT Y	FREQ	CUM. FREQ	PERCENT	CUM. PERCENT
0.015	3	3	15.00	15.00
0.033	6	9	30.00	45.00
0.051	0	9	0.00	45.00
0.069	1	10	5.00	50.00
0.087	2	12	10.00	60.00
0.105	0	12	0.00	60.00
0.123	4	16	20.00	80.00
0.141	2	18	10.00	90.00
0.159	2	20	10.00	100.00

(b) Number of samples = 36



MIDPOINT Y	FREQ	CUM. FREQ	PERCENT	CUM. PERCENT
0.015	6	6	16.67	16.67
0.033	12	18	33.33	50.00
0.051	1	19	2.78	52.78
0.069	1	20	2.78	55.56
0.087	5	25	13.89	69.44
0.105	4	29	11.11	80.56
0.123	5	34	13.89	94.44
0.141	2	36	5.56	100.00
0.159	0	36	0.00	100.00

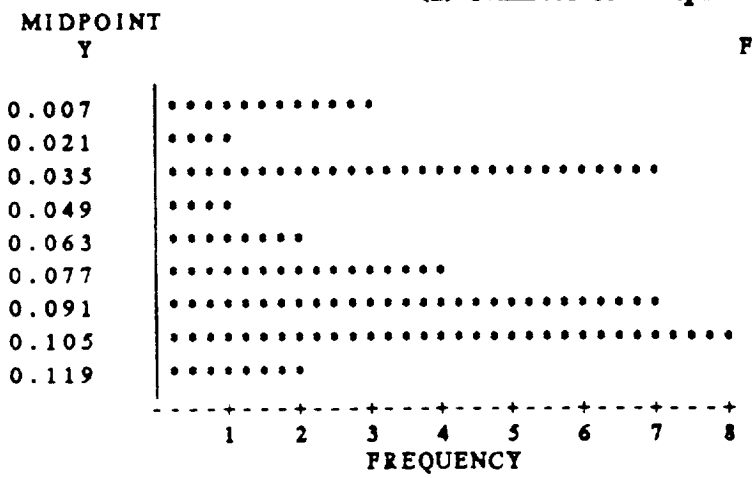
(c) Continuous Trace



MIDPOINT Y	FREQ	CUM. FREQ	PERCENT	CUM. PERCENT
0.015	174	174	15.34	15.34
0.033	384	558	33.86	49.21
0.051	17	575	1.50	50.71
0.069	25	600	2.20	52.91
0.087	148	748	13.05	65.96
0.105	148	896	13.05	79.01
0.123	139	1035	12.26	91.27
0.141	67	1102	5.91	97.18
0.159	32	1134	2.82	100.00

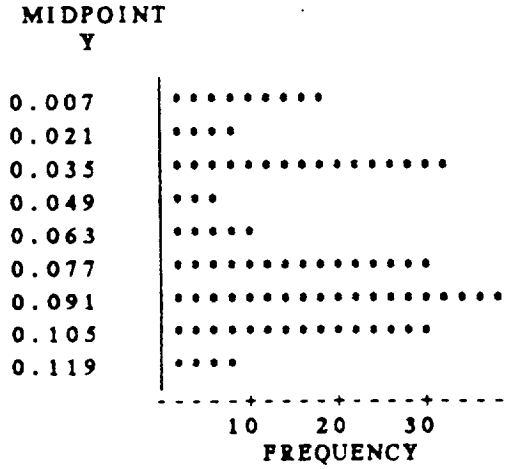
Fig. 3. Frequency bar chart of miss ratio for DRC : Context-switch interval = 5K

(a) Number of samples = 35



MIDPOINT Y	FREQ	CUM. FREQ	PERCENT	CUM. PERCENT
0.007	3	3	8.57	8.57
0.021	1	4	2.86	11.43
0.035	7	11	20.00	31.43
0.049	1	12	2.86	34.29
0.063	2	14	5.71	40.00
0.077	4	18	11.43	51.43
0.091	7	25	20.00	71.43
0.105	8	33	22.86	94.29
0.119	2	35	5.71	100.00

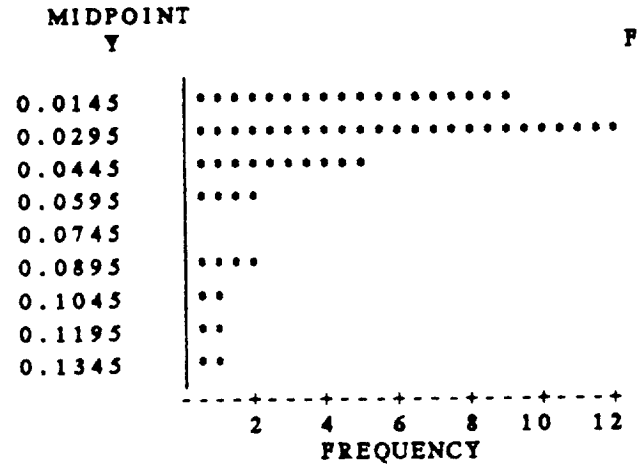
(b) Continuous Trace



MIDPOINT Y	FREQ	CUM. FREQ	PERCENT	CUM. PERCENT
0.007	17	17	9.71	9.71
0.021	7	24	4.00	13.71
0.035	32	56	18.29	32.00
0.049	5	61	2.86	34.86
0.063	9	70	5.14	40.00
0.077	30	100	17.14	57.14
0.091	38	138	21.71	78.86
0.105	29	167	16.57	95.43
0.119	8	175	4.57	100.00

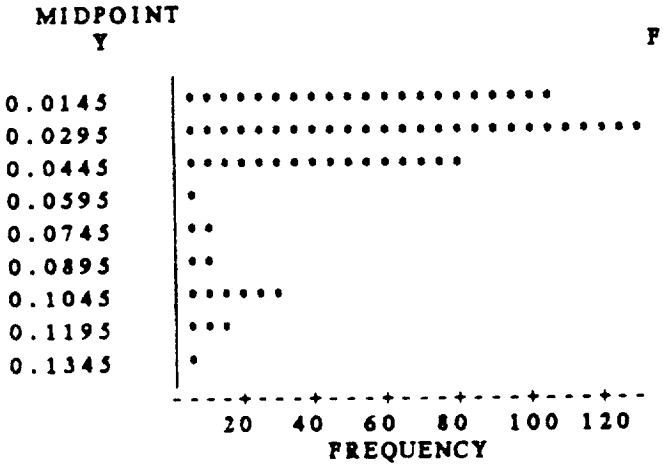
Fig. 4. Frequency bar chart of miss ratio for ELI : Context-switch interval = 10K

(a) Number of samples = 33



MIDPOINT Y	FREQ	CUM. FREQ	PERCENT	CUM. PERCENT
0.0145	9	9	27.27	27.27
0.0295	12	21	36.36	63.64
0.0445	5	26	15.15	78.79
0.0595	2	28	6.06	84.85
0.0745	0	28	0.00	84.85
0.0895	2	30	6.06	90.91
0.1045	1	31	3.03	93.94
0.1195	1	32	3.03	96.97
0.1345	1	33	3.03	100.00

(b) Continuous Trace



MIDPOINT Y	FREQ	CUM. FREQ	PERCENT	CUM. PERCENT
0.0145	107	107	27.23	27.23
0.0295	129	236	32.82	60.05
0.0445	80	316	20.36	80.41
0.0595	5	321	1.27	81.68
0.0745	10	331	2.54	84.22
0.0895	12	343	3.05	87.28
0.1045	31	374	7.89	95.17
0.1195	13	387	3.31	98.47
0.1345	6	393	1.53	100.00

Fig. 5. Frequency bar chart of miss ratio for RRL : Context-switch interval = 5K

tions. It is found that the distributions of nearly 20 samples have significant difference from the true ones. But, the profiles of about 35 samples closely resemble the actual shape in all different cases. It is easily shown that the distributions of the samples are statistically equivalent to the actual ones with a high degree of confidence. Therefore, it is concluded that 35 samples are adequate for this method.

The distribution of the miss ratios provides a lot of information about the program behavior which is lost when only the mean value is considered. For example, the distribution of miss ratio is useful in performance analysis of cache memories in multiprocessors or cache memories with a finite write-back buffer. In both cases, a high miss rate imposes a relatively higher penalty *per miss* than the penalty per miss at low miss rates. These distributions for the four programs are quite different from one another; but, for a particular program, it follows similar trend for the various context switch intervals (i.e., for different sample sizes in our method). The miss ratio distribution of the DRC program has two distinct modes while for the FSIM program, it is nearest to the normal distribution among the four programs. This indicates that the miss ratio of the DRC program is spread over two regions which are apart from each other; the region with smaller value corresponds to the tight loop structures in the program. The miss ratio versus time plot (Fig. 6) for this program corroborates this feature. The FSIM program on the other hand has its miss ratio randomly spread over a single band as found in its miss ratio versus time plot.

5.1. Comparison with Conventional Simulation

In conventional simulation, the measured distribution of the miss ratio can be quite different from the actual one. The conventional method for cold-start miss ratio can be considered as simulating consecutive samples with no interval in between. To compare conventional simulation with our method, the distribution of miss ratio of 35 consecutive context switch intervals of size 5K addresses is checked for closeness with the actual

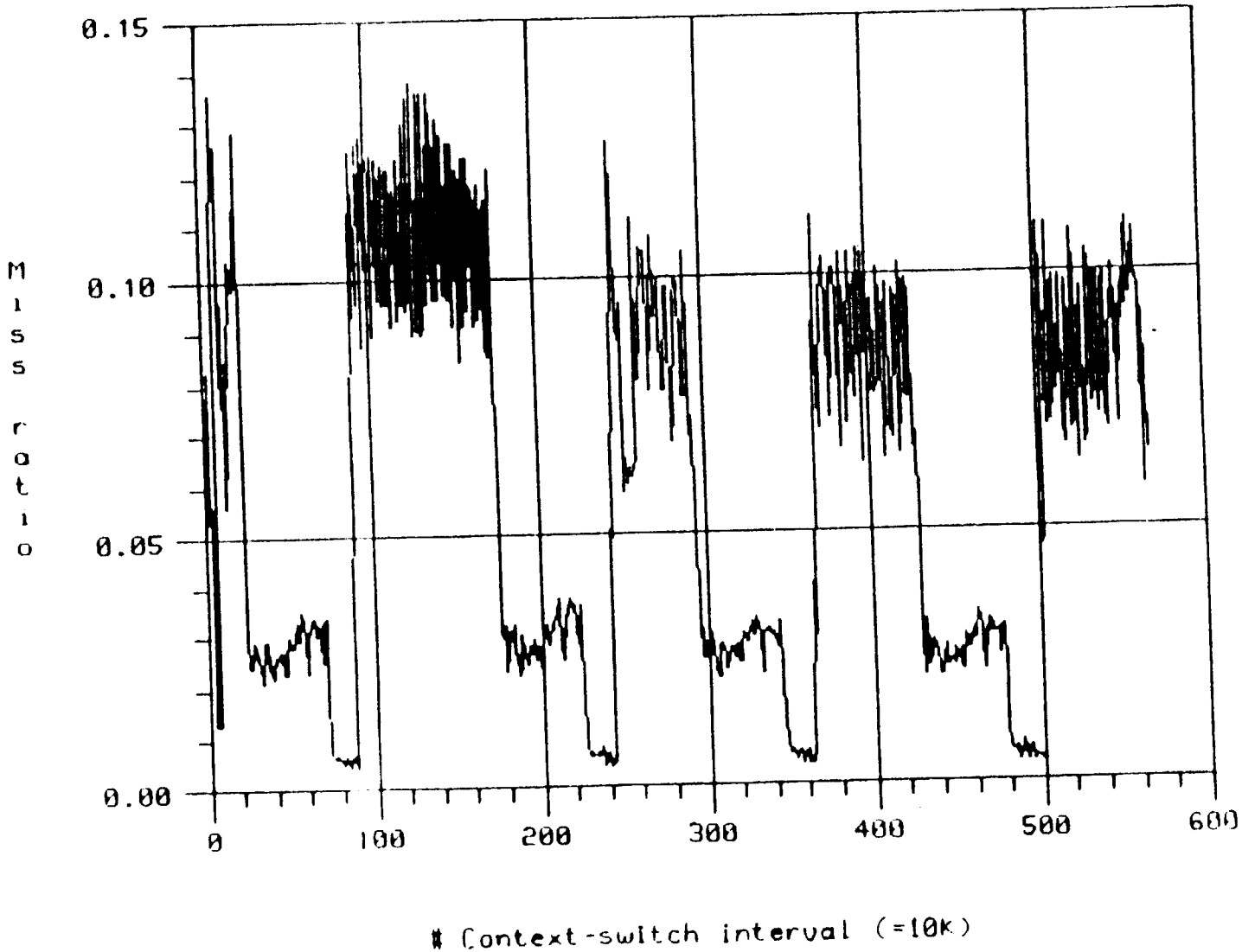


Fig. 6. Miss ratio versus time : DRC program : Context switch interval = 10K

distribution of miss ratio for DRC program, because this simulation has the same space-time cost as our method. These two distributions are found to differ considerably in mean value, standard deviation and the shape (Fig. 7). This is expected from Figure 6 which shows that 35 consecutive intervals, each of size 5K cover only a small, initial part of the program which is very different from the rest. This shows the inadequacy of conventional method to represent true program behavior.

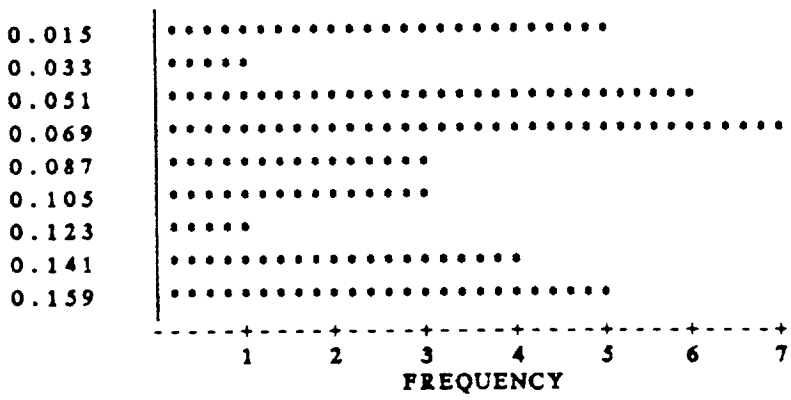
6. Evaluation of Other Context Switch Intervals

In this section, we propose a model of cache behavior for larger context switch intervals, based on the performance measures of smaller intervals. Our aim is to perform only one simulation on samples of one size and statistically project the results for other context switch intervals. Once simulation is performed for samples of a particular size, information about smaller task intervals is already present in the results. So, only those intervals which are greater than the sample size need to be projected. To do that, we will first consider how the mean value of miss ratio over the samples vary within the duration of the sample.

Figure 8 shows the typical plot of mean miss ratio (over different samples) with the number of address references during the period of the sample size. The value of mean miss ratio at each point is calculated by considering the number of hits and misses from the end of the interval, rather than from the beginning for each sample. This is done to avoid the effect of cold start at the beginning of the interval on the later segments of the task interval. This enables us to check if the instantaneous mean miss rate in an interval reaches a steady value as the cache gets more and more filled up. It is observed in all our experiments that the miss ratio really attains a steady state for sufficiently long samples (as small as 5K

(a) Conventional method with 35 intervals

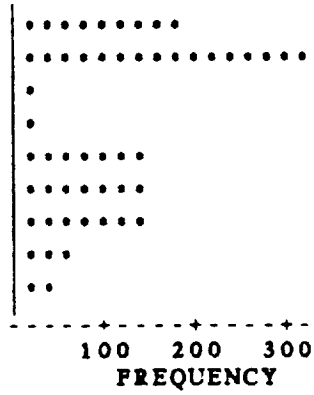
(Mean : 0.083; Standard deviation : 0.048)



MIDPOINT Y	FREQ	CUM. FREQ	PERCENT	CUM. PERCENT
0.015	5	5	14.29	14.29
0.033	1	6	2.86	17.14
0.051	6	12	17.14	34.29
0.069	7	19	20.00	54.29
0.087	3	22	8.57	62.86
0.105	3	25	8.57	71.43
0.123	1	26	2.86	74.29
0.141	4	30	11.43	85.71
0.159	5	35	14.29	100.00

(b) Complete Trace

(Mean : 0.068; Standard deviation : 0.043)



MIDPOINT Y	FREQ	CUM. FREQ	PERCENT	CUM. PERCENT
0.015	174	174	15.34	15.34
0.033	384	558	33.86	49.21
0.051	17	575	1.50	50.71
0.069	25	600	2.20	52.91
0.087	148	748	13.05	65.96
0.105	148	896	13.05	79.01
0.123	139	1035	12.26	91.27
0.141	67	1102	5.91	97.18
0.159	32	1134	2.82	100.00

Fig. 7. Comparison of conventional method for DRC : Context-switch interval = 5K

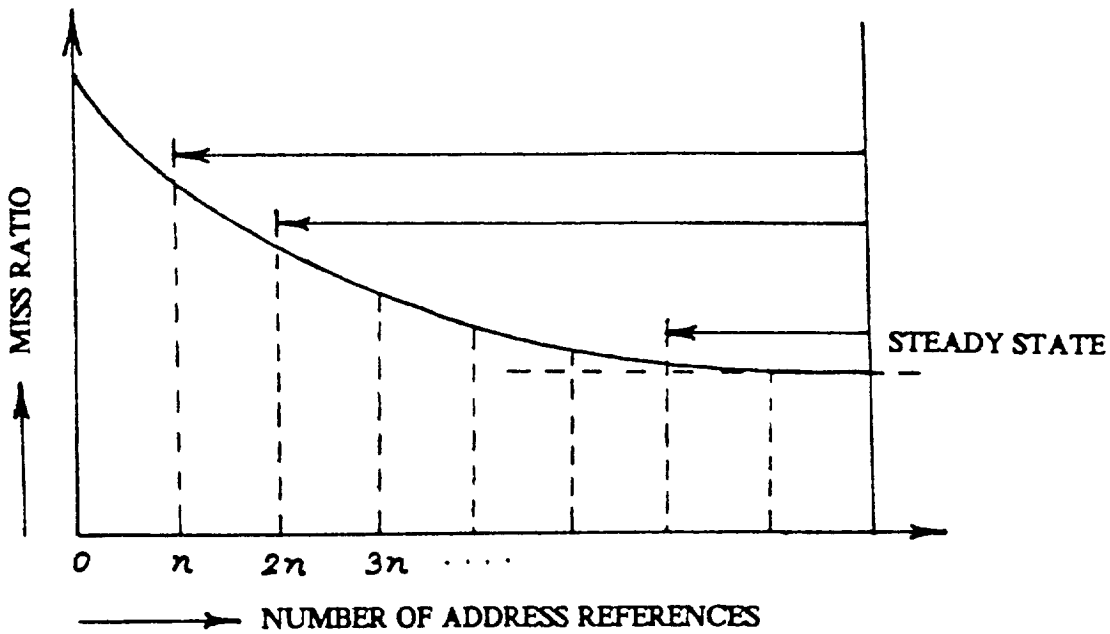


Fig. 8. Mean miss ratio versus number of addresses

for all the traces).

6.1. Our Model

Given a sample size n_0 which is long enough for the miss ratio to reach a stable value, this steady state will continue for the rest of a larger task interval n (Fig. 9).

Based on this assumption, the value of the miss ratio for a larger context switch interval n can be easily calculated and is given by :

$$m(n) = \frac{n_0}{n} m(n_0) + (1 - \frac{n_0}{n}) m_\infty$$

where, m(n) : miss ratio for interval n

$m(n_0)$: miss ratio for interval n_0

m_∞ : steady state value of miss ratio

6.2. Results

Based on the formula above, the miss ratio for 20K interval is predicted from that of 5K and 10K samples for all the traces; also, the miss ratio for 10K context switch interval is calculated from that of 5K samples (Table 2). It is found that the projected value is reasonably close to the actual value, measured from the continuous trace. However, we are still in the process of comparing the distributions of the projected miss ratio with the actual ones. More exhaustive statistical analysis needs to be done to find out how much can the samples be extended to larger context switch intervals with reasonable accuracy and to resolve other related issues.

7. Concluding Remarks

In this paper, we have established a sampling-based technique for cache memory simulation. This method provides insight into the true program behavior by giving the

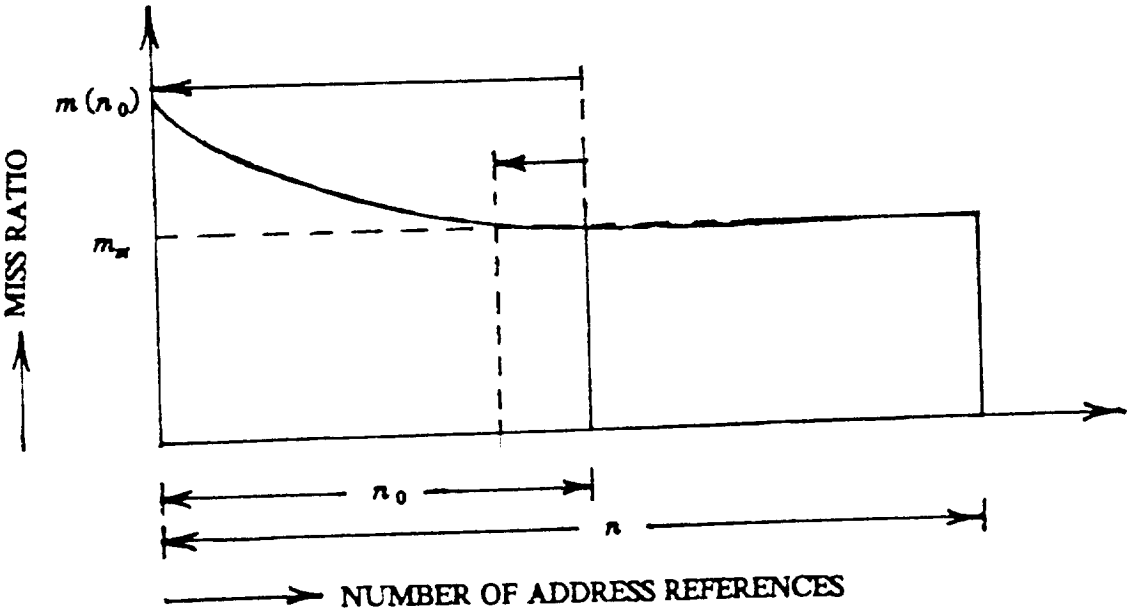


Fig. 9. Model for larger context switch intervals

Table 2. Calculated Miss Ratios for Different Context Switch Intervals

Program	Context switch interval	Actual measured miss ratio	Miss ratio calculated from context switch interval	
			5K	10K
FSIM	10K	.060	.062	
	20K	.051	.053	.054
DRC	10K	.057	.061	
	20K	.052	.056	.0525
ELI	10K	.067	.067	
	20K	.061	.0615	.0635
RRL	10K	.034	.035	
	20K	.030	.0315	.032

accurate estimates of both the mean value of miss ratio and its distribution. In comparison, conventional techniques just measure the mean miss ratio for one small segment of a program; thus, they may not be representative of actual performance. The cost of the sampling-based method in terms of both memory space and computer time is small. There is one pathological case in which this technique might not give satisfactory result: this will happen very rarely when (1) the miss ratio versus time plot of the program has a step-function; as in Figure 6, (2) the period of this variation is constant and (3) the sampling interval happens to be the same as this period and all the samples are taken predominantly at the top or bottom level of the step-function. This can be avoided by taking the samples at random intervals or by using two sets of samples having two different uniform sampling intervals. Hence the paper shows that the sampling-based simulation is an accurate approach. More measurements with other programs are essential to establish guidelines for general usage.

Simulating only one set of samples and projecting the result for other context switch intervals look very promising. But, still some more analysis has to be done regarding this issue. Another direction in which this research needs to be extended is to separate the cold-start effect at the beginning of a task interval from the characteristics of the rest of the interval. Finally, these new simulation methods can be very effectively used not only for cache memory simulation of LISP programs, they can also be easily extended for other programming languages.

Acknowledgements

We would like to thank Hoichi Cheong, Susheel Chandra, Ram Kunda, Suresh Rajamoney and G. Sivakumar for their assistance in collecting the traces. Many thanks to

Ram Chillarege and Sandy Hsueh for helping in the statistical analysis.

References

[1] A. J. Smith, "Cache Memories," *ACM Computing Surveys*, vol. 14, 3, pp. 473-530, September, 1982.

[2] D. W. Clark, "Measurements of Dynamic List Structure Use in Lisp," *IEEE Trans. Software Eng.*, vol. SE-5, 1, pp. 51-59, January, 1979.

[3] A. J. Smith, "Cache Evaluation and the Impact of Workload Choice," *Proc. 12th Annual Int'l Symp. on Computer Architecture*, vol. 13, 3, pp. 64-73, June, 1985.

[4] A. J. Smith, "Line (Block) Size Choice for CPU Cache Memories," UC Berkeley CS Report, UCB/CSD85/239, June, 1985..

[5] W. D. Strecker, "Cache Memories for PDP-11 Family Computers," *Proc. Third Annual Conf. on Computer Architecture*, pp. 155-158, 1976.

[6] W. D. Strecker, "Transient Behavior of Cache Memories," *ACM Trans. Computer Systems*, vol. 1, 4, pp. 281-293, November, 1983.

[7] I. J. Haikala, "Cache Hit Ratios with Geometric Task Switch Intervals," *Proc. 11th Annual Int'l Symp. on Computer Architecture*, vol. 12, 3, pp. 364-371, June, 1984.

[8] M. Easton, "Computation of Cold Start Miss Ratios," *IEEE Trans. Comp.*, vol. C-27, 5, pp. 404-408, May, 1978.

[9] J. S. Emer and D. W. Clark, "A Characterization of Processor Performance in the VAX-11/780," *Proc. 11th Annual Int'l Symp. on Computer Architecture*, vol. 12, 3, pp. 301-310, June, 1984.

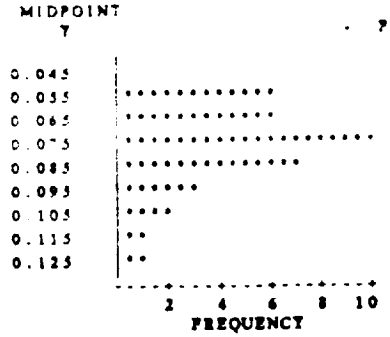
[10] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, 1984.

[11] M. D. Hill and A. J. Smith, "Experimental Evaluation of On-Chip Microprocessor Cache Memories," *Proc. 11th Annual Int'l Symp. on Computer Architecture*, vol. 12, 3, pp. 158-166, June, 1984.

Appendix

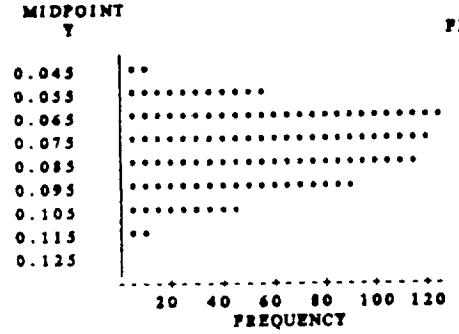
(Frequency bar charts for remaining
context switch intervals)

(a) Number of samples = 36



MIDPOINT Y	FREQ	CUM FREQ	PERCENT	CUM PERCENT
0.045	0	0	0.00	0.00
0.055	6	6	16.67	16.67
0.065	6	12	16.67	33.33
0.075	10	22	27.78	61.11
0.085	7	29	19.44	80.56
0.095	3	32	8.33	88.89
0.105	2	34	5.56	94.44
0.115	1	35	2.78	97.22
0.125	1	36	2.78	100.00

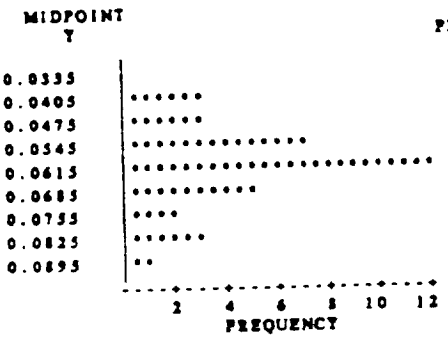
(b) Continuous Trace



MIDPOINT Y	FREQ	CUM. FREQ	PERCENT	CUM. PERCENT
0.045	12	12	2.09	2.09
0.055	57	69	9.93	12.02
0.065	125	194	21.78	33.80
0.075	118	312	20.56	54.36
0.085	115	427	20.03	74.39
0.095	88	515	15.33	89.72
0.105	45	560	7.84	97.56
0.115	12	572	2.09	99.65
0.125	2	574	0.35	100.00

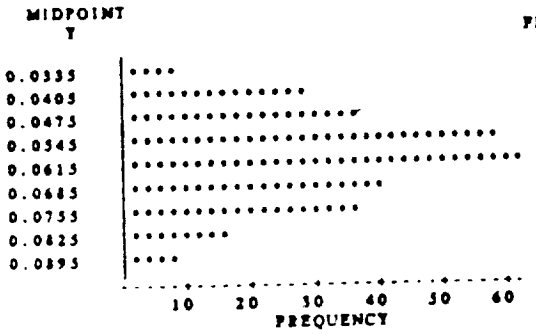
Fig. 10. Frequency bar chart of miss ratio for FSIM : Context-switch interval = 5K

(a) Number of samples = 36



MIDPOINT Y	FREQ	CUM. FREQ	PERCENT	CUM. PERCENT
0.0335	0	0	0.00	0.00
0.0405	3	3	8.33	8.33
0.0475	3	6	8.33	16.67
0.0545	7	13	19.44	36.11
0.0615	12	25	33.33	69.44
0.0685	5	30	13.89	83.33
0.0755	2	32	5.56	88.89
0.0825	3	35	8.33	97.22
0.0895	1	36	2.78	100.00

(b) Continuous Trace

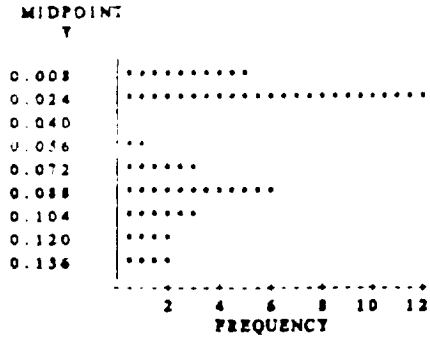


MIDPOINT Y	FREQ	CUM. FREQ	PERCENT	CUM. PERCENT
0.0335	8	8	2.79	2.79
0.0405	27	35	9.41	12.20
0.0475	36	71	12.54	24.74
0.0545	57	128	19.86	44.60
0.0615	62	190	21.60	66.20
0.0685	39	229	13.59	79.79
0.0755	35	264	12.20	91.99
0.0825	15	279	5.23	97.21
0.0895	8	287	2.79	100.00

Fig. 11. Frequency bar chart of miss ratio for FSIM : Context-switch interval = 10K

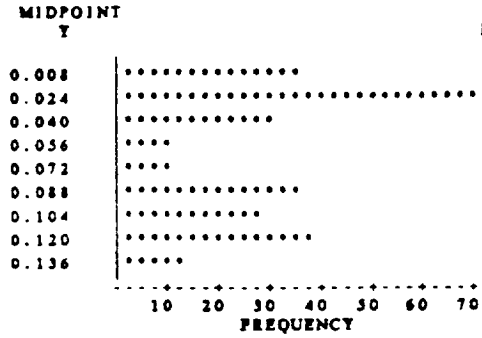
ORIGINAL PAGE IS
OF POOR QUALITY

(a) Number of samples = 36



MIDPOINT Y	FREQ	CUM FREQ	PERCENT	CUM PERCENT
0.008	5	5	14.71	14.71
0.024	12	17	35.29	50.00
0.040	0	17	0.00	50.00
0.056	1	18	2.94	52.94
0.072	3	21	8.82	61.76
0.088	6	27	17.65	79.41
0.104	3	30	8.82	88.24
0.120	2	32	5.88	94.12
0.136	2	34	5.88	100.00

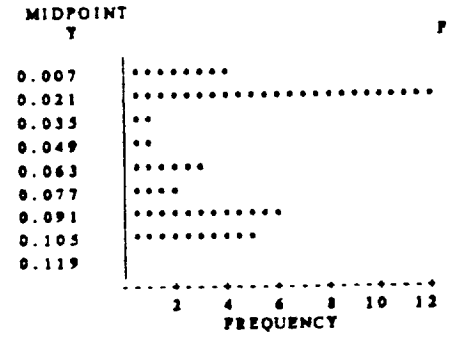
(b) Continuous Trace



MIDPOINT Y	FREQ	CUM. FREQ	PERCENT	CUM. PERCENT
0.008	35	35	13.01	13.01
0.024	71	106	26.39	39.41
0.040	30	136	11.15	50.56
0.056	11	147	4.09	54.65
0.072	9	156	3.35	57.99
0.088	35	191	13.01	71.00
0.104	28	219	10.41	81.41
0.120	38	257	14.13	95.54
0.136	12	269	4.46	100.00

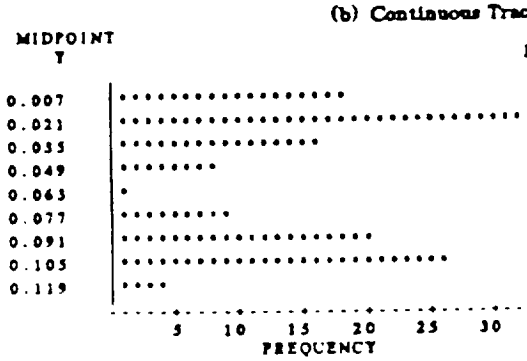
Fig. 12. Frequency bar chart of miss ratio for DRC: Context-switch interval = 10K

(a) Number of samples = 36



MIDPOINT Y	FREQ	CUM. FREQ	PERCENT	CUM. PERCENT
0.007	4	4	11.76	11.76
0.021	12	16	35.29	47.06
0.035	1	17	2.94	50.00
0.049	1	18	2.94	52.94
0.063	3	21	8.82	61.76
0.077	2	23	5.88	67.65
0.091	6	29	17.65	85.29
0.105	5	34	14.71	100.00
0.119	0	34	0.00	100.00

(b) Continuous Trace

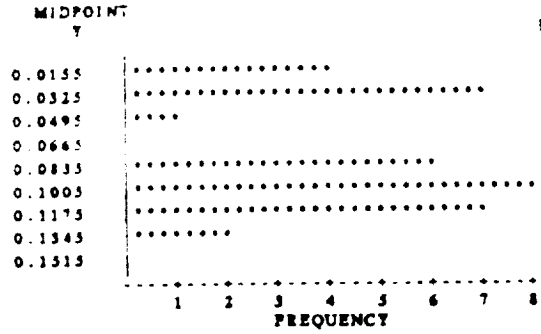


MIDPOINT Y	FREQ	CUM. FREQ	PERCENT	CUM. PERCENT
0.007	18	18	13.43	13.43
0.021	32	50	23.88	37.31
0.035	16	66	11.94	49.25
0.049	8	74	5.97	55.22
0.063	1	75	0.75	55.97
0.077	9	84	6.72	62.69
0.091	20	104	14.93	77.61
0.105	26	130	19.40	97.01
0.119	4	134	2.99	100.00

Fig. 13. Frequency bar chart of miss ratio for DRC: Context-switch interval = 20K

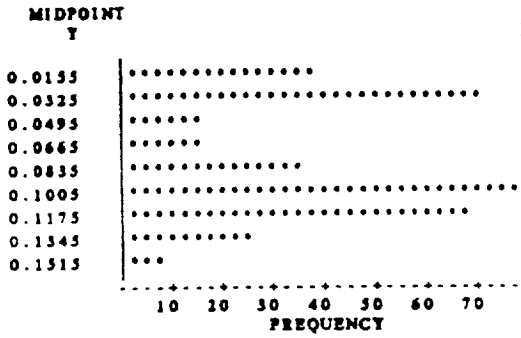
ORIGINAL PAGE IS
OF POOR QUALITY

(a) Number of samples = 35



FREQ	CUM. FREQ	PERCENT	CUM PERCENT
4	4	11.43	11.43
7	11	20.00	31.43
1	12	2.86	34.29
0	12	0.00	34.29
6	18	17.14	51.43
8	26	22.86	74.29
7	33	20.00	94.29
2	35	5.71	100.00
0	35	0.00	100.00

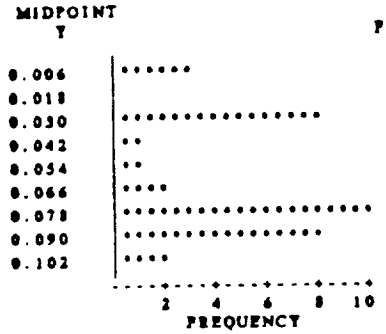
(b) Continuous Trace



FREQ	CUM. FREQ	PERCENT	CUM. PERCENT
37	37	10.57	10.57
69	106	19.71	30.29
15	121	4.29	34.57
15	136	4.29	38.86
36	172	10.29	49.14
77	249	22.00	71.14
68	317	19.43	90.57
26	343	7.43	98.00
7	350	2.00	100.00

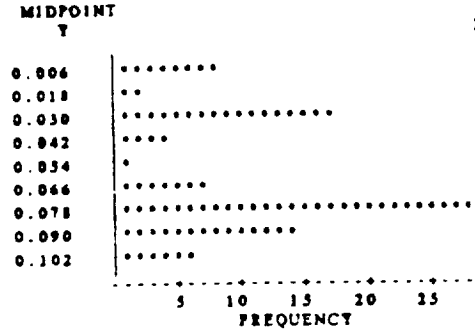
Fig. 14. Frequency bar chart of miss ratio for ELI : Context-switch interval = 5K

(a) Number of samples = 35



FREQ	CUM. FREQ	PERCENT	CUM. PERCENT
3	3	8.57	8.57
0	3	0.00	8.57
8	11	22.86	31.43
1	12	2.86	34.29
1	13	2.86	37.14
2	15	5.71	42.86
10	25	28.57	71.43
8	33	22.86	94.29
2	35	5.71	100.00

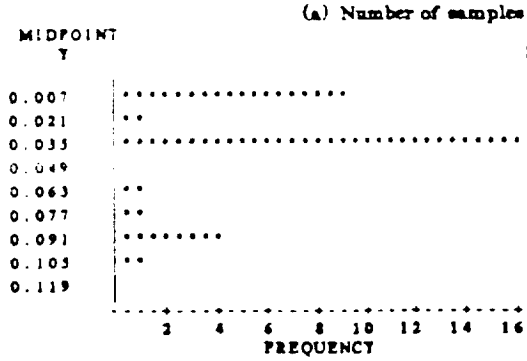
(b) Continuous Trace



FREQ	CUM. FREQ	PERCENT	CUM. PERCENT
8	8	9.20	9.20
2	10	2.30	11.49
17	27	19.54	31.03
4	31	4.60	35.63
1	32	1.15	36.78
7	39	8.05	44.83
28	67	32.18	77.01
14	81	16.09	93.10
6	87	6.90	100.00

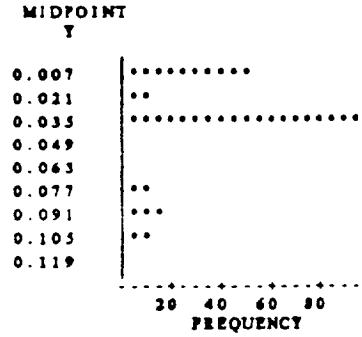
Fig. 15. Frequency bar chart of miss ratio for ELI : Context-switch interval = 20K

(a) Number of samples = 33



FREQ	CUM FREQ	PERCENT	CUM PERCENT
9	9	27.27	27.27
1	10	3.03	30.30
16	26	48.48	78.79
0	26	0.00	78.79
1	27	3.03	81.82
1	28	3.03	84.85
4	32	12.12	96.97
1	33	3.03	100.00
0	33	0.00	100.00

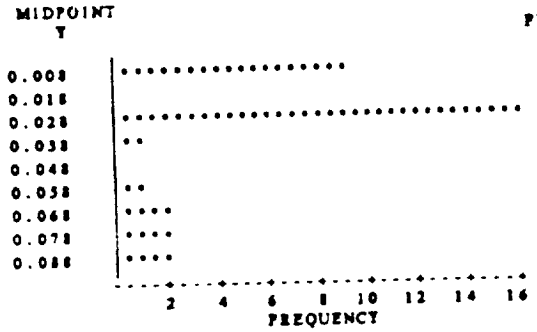
(b) Continuous Trace



FREQ	CUM. FREQ	PERCENT	CUM. PERCENT
52	52	26.53	26.53
12	64	6.12	32.65
94	158	47.96	80.61
1	159	0.51	81.12
2	161	1.02	82.14
9	170	4.59	86.73
16	186	8.16	94.90
9	195	4.59	99.49
1	196	0.51	100.00

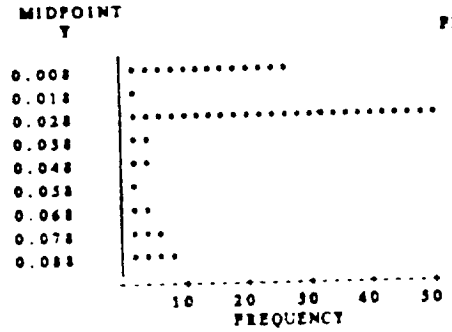
Fig. 16. Frequency bar chart of miss ratio for RRL : Context-switch interval = 10K

(a) Number of samples = 33



FREQ	CUM. FREQ	PERCENT	CUM. PERCENT
9	9	27.27	27.27
0	9	0.00	27.27
16	25	48.48	75.76
1	26	3.03	78.79
0	26	0.00	78.79
1	27	3.03	81.82
2	29	6.06	87.88
2	31	6.06	93.94
2	33	6.06	100.00

(b) Continuous Trace



FREQ	CUM. FREQ	PERCENT	CUM. PERCENT
26	26	26.53	26.53
1	27	1.02	27.55
49	76	50.00	77.55
3	79	3.06	80.61
3	82	3.06	83.67
1	83	1.02	84.69
3	86	3.06	87.76
5	91	5.10	92.86
7	98	7.14	100.00

Fig. 17. Frequency bar chart of miss ratio for RRL : Context-switch interval = 20K

N94-76420

PERFORMANCE OF A PARALLEL ALGORITHM FOR STANDARD CELL PLACEMENT ON THE INTEL HYPERCUBE

Mark Jones and Prithviraj Banerjee

Computer Systems Group
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign

ABSTRACT

In this paper, we present a parallel simulated annealing algorithm for standard cell placement that is targeted to run on the Intel Hypercube. We present a novel tree broadcasting strategy that is used extensively in our algorithm for updating cell locations in the parallel environment. Studies on the performance of our algorithm on example industrial circuits show that it is faster and gives better final placement results than the uniprocessor simulated annealing algorithms.

1. INTRODUCTION

Given a set of standard cells of constant height and variable width, and a net list which describes the interconnections among the cells, our objective is to place the cells in a VLSI layout so as to minimize the total length of wires interconnecting the cells. The simulated annealing technique has been proposed and applied to the placement problem in a program called TimberWolf which, by applying cell displacements and exchanges randomly, avoids getting stuck at local minima and thereby achieves near-optimal placement [1,2]. A major limitation of TimberWolf is that it is extremely slow.

Recently, some researchers have started to investigate speeding up simulated annealing algorithms by running them on parallel processor systems. Aarts et al have proposed schemes for parallelizing simulated annealing algorithms for several general classes of problems and have discussed theoretical convergence characteristics [3]. A parallel algorithm for the Traveling Salesman Problem based on simulated annealing has been reported for the Hypercube [4]. Parallel algorithms for partitioning and routing [5], macro-cell placement [6], and topological optimization of multiple level array logic [7] have been proposed by several researchers.

Two multiprocessor-based simulated annealing algorithms, called Move Decomposition and Parallel Moves, for the standard cell problem have been reported by Rutenbar and Kravitz [8,9]. Last year, Banerjee and Jones had proposed a parallel algorithm that is targeted to run on a Hypercube computer [10]. At the same time, Rose et al proposed two algorithms, called Heuristic Spanning and Section Annealing, for standard cell placement on a shared memory multiprocessor [11].

There are a number of basic differences in the previous approaches. The first approach is a shared memory algorithm that is basically simulating a serial simulated annealing environment but evaluating each individual move faster. The second algorithm is also based on a shared memory environment but evaluates multiple moves in parallel but accepts only one move. Hence, its convergence characteristics are identical to the uniprocessor algorithm. In the third case proposed by Banerjee and Jones, the algorithm is based on a local memory message-passing architecture; the moves are evaluated in parallel and accepted/rejected in parallel on the basis of changes in the cost function for each move assuming that the other moves are not made. The theoretical considerations of whether the annealing properties are still preserved when the cost calculations are based on slightly outdated information and when only a restricted set of moves are allowed, may be a subject of future research. Experimentally, it has been verified that the algorithm works. The scheme of Rose et al uses a combination of heuristic methods and simulated annealing to gain performance, and hence cannot be directly compared with the other approaches that are purely based on annealing.

A hypercube topology consists of 2^d processors that are interconnected through the topology of a cube in d dimensions. Several prototypes of such machines have been built [12,13]; two of them are now available commercially from Intel [14], and Ametek [15].

In this paper, we present an algorithm using simulated annealing on the hypercube that improves upon our earlier work presented in [10]. This enhanced algorithm reduces the communication overhead, can handle more features of the placement problem, and is more machine-specific (it is targeted to run on the Intel Hypercube). The basic idea of allowing parallel exchange and displace moves in different dimensions of the hypercube, and accepting/rejecting moves on the basis of changes in cost functions ignoring the effects of other moves remains the same. However, several new concepts were utilized that reduced the communication overhead substantially for the targeted machine.

In Section 2, we will outline the basic algorithm, describe the data structures that are necessary to support various parallel move evaluations, and discuss how the subtasks for evaluating the acceptability of parallel moves are assigned. We also present a novel tree broadcasting strategy for the hypercube that is used extensively in our algorithm for updating cell locations in the parallel environment. Section 3 describes our implementation of the algorithm on an Intel hypercube simulator. We report on the performance of our algorithm for several actual standard circuits used in industry. We show that the parallel algorithm gives about 10-20% better final placements than conventional uniprocessor simulated annealing algorithms. Finally we present some accurate estimates of the execution time for the algorithm.

2. PARALLEL ALGORITHM FOR CELL PLACEMENT

2.1. Overview of parallel algorithm

We now describe an algorithm for performing the standard cell placement using a variation of the TimberWolf [2] algorithm on a hypercube of $log(P)$ dimensions connecting P processors. Let us suppose that we are given the problem of placing N standard cells where $N >> P$

Acknowledgment: This research was supported in part by the National Aeronautics and Space Administration under Contract NASA NAG 1-613.

ORIGINAL PAGE IS
OF POOR QUALITY

STEP 1. Perform initial cell assignments in P processors.
STEP 2. Determine initial temperature.
STEP 3. While "Stopping criteria" : temperature < 0.1 not reached
STEP 4. Generate new temperature
STEP 5. For inner_loop_count = 1 to NA
STEP 6. For each dimension i=0 to log(P)-1 do
STEP 7. Randomly select P/2 moves (exchange or displace) in parallel among pairs of PEs connected in dimension i.
STEP 8. Check "range-limiter" function in dimension i.
STEP 9. Evaluate change in cost for each move between pairs of PEs independently.
STEP 10. Accept/reject moves using exponential function independently.
STEP 11. Broadcast new cell locations to all other processors.
STEP 12. ENDFOR; ENDFOR; ENDWHILE

In the following subsections, we describe each of the steps in more detail.

2.2. Cell Assignment to Processors

The technique for mapping a $log(P)$ dimensional hypercube onto a two-dimensional area is identical to the one described in [10]. We will briefly mention the scheme here for completeness. In a 64-processor hypercube a processor having a binary address $p_5p_4 \ldots p_i \ldots p_0$ is connected to processor $p_5p_4 \ldots \overline{p_i} \ldots p_0$ via a link in dimension i. We propose that each processor be assigned an approximately equal area portion of the total chip area which can be viewed as a virtual 8×8 square grid. Each virtual grid corresponds to a horizontal portion of a number of rows. The cells are initially assigned randomly to different processors such that each processor has an approximately equal number of cells assigned to it. The cells within each processor are also randomly placed with no regard to area overlaps. Since all cells have constant height, each processor therefore is assigned a rectangular portion of the chip area. The correspondence between processor addresses and virtual grid regions on the physical chip area is shown in Fig. 1. By choosing such a map, we guarantee that the processors that are adjacent in a pre-determined set of four dimensions of the hypercube allow for all nearest North-South-East-West neighbor displace/exchanges. The other two dimensions of the hypercube are used for displace/exchanges across larger distances in the area map. This is illustrated in Fig. 1.

2.3. Distributed Data Structure

We assume that each processor contains the following information to enable the computation of the cost function in parallel among the processors in the hypercube:
(1) A list of cells currently assigned to this processor along with the following information for each cell:
(2) The width of the cell;
(3) The (x,y) coordinate location at which the center of the cell is currently placed;
(4) A list of nets to which this cell is connected;
(5) For each net listed in (4), a list of other cells, to which the net is connected, along with the (x,y) pin location(s) within these cells;
(6) A list of (x,y) locations and widths of all cells that are assigned to processors that are adjacent in the two dimensions of the hypercube corresponding to the East-West nearest neighbors in the physical area map is also maintained in each processor.

The state of any particular cell is composed of the information in (2) through (5) and is packed within a continuous block of memory to allow for easy packet transfer of information between nodes. Fig. 2 shows an example of the blocked memory data structure for typical cells.

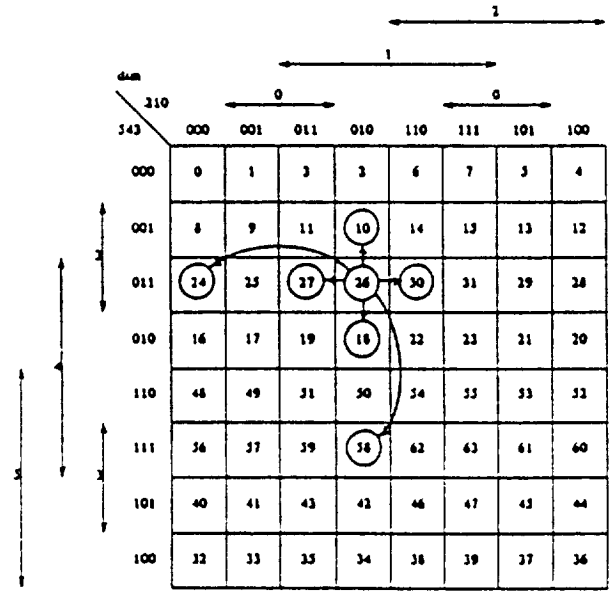


Fig. 1. Area map of 64-processor Hypercube.

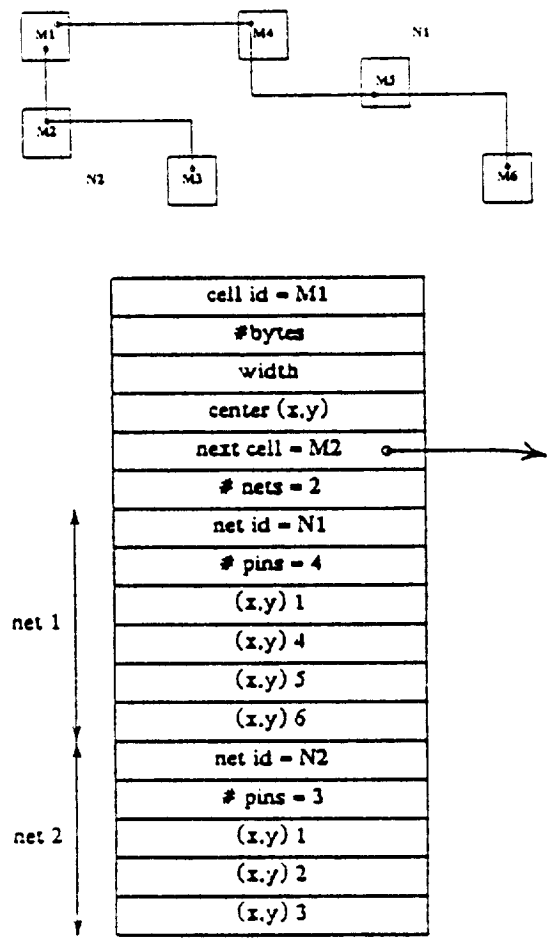


Fig. 2. Example net and corresponding memory structure.

ORIGINAL PAGE IS
OF POOR QUALITY

. Cost function

The cost function for the standard cell placement problem isists of three parts:

Estimated wire-length using half the perimeter of the bound- box rule.

Overshoot or undershoot of each row length over the desired x length:

Area overlap between cells in the same row.

. Moves

After the cells have been evenly distributed by area iongst the processors of the hypercube, each processor repeat- ıy interacts with its neighboring processors in each of the d nensions of the hypercube. The set of steps involved in a rallel set of moves is outlined below. At each time step, $P/2$ irs of processors participate in the evaluating $P/2$ mov's.

ROCEDURE PARALLEL MOVES:

TEP 1. For each pair of processors (p,q) connected in dimension if the inner_loop_count is even and if p < q, then p is chosen be the Master, q to be the Slave, otherwise vice versa.

TEP 2. Master randomly decides the type of the move.

TEP 3.1. If MOVE = INTER-PROCESSOR EXCHANGE, proces- r p (Master) randomly selects a cell CELL(p) and sends its data ructure to processor q. Meanwhile, processor q (Slave) also ndomly selects a cell CELL(q) and sends its data structure to ocessor p.

TEP 4.1. Compute $\Delta_{exchange}(CELL(p),CELL(q)) =$
$\Delta_1(WL,CELL(p),POS(q),p) + \Delta_2(WL,CELL(q),POS(p),q) +$
$\Delta_3(AO,CELL(p),POS(p),p) + \Delta_4(AO,CELL(p),POS(q),q) +$
$\Delta_5(AO,CELL(q),POS(q),q) + \Delta_6(AO,CELL(q),POS(p),p) +$
$\Delta_7(EO,CELL(p),POS(p),p) + \Delta_8(EO,CELL(p),POS(q),q) +$
$\Delta_9(EO,CELL(q),POS(q),q) + \Delta_{10}(EO,CELL(q),POS(p),p)$

TEP 5.1. Processor q sends the portion of the cost function it mputed to processor p.

TEP 6.1. Go to STEP 7

TEP 3.2. If MOVE = INTRA-PROCESSOR EXCHANGE, proces- r p (Master) randomly selects two cells, $CELL_1(p)$ and $CELL_2(p)$, both within its allocated area map.

TEP 4.2. Compute $\Delta_{exchange}(CELL_1(p),CELL_2(p)) = \Delta_1(WL,p)$ $\Delta_2(AO,p) + \Delta_3(EO,p)$

TEP 5.2. Go to STEP 7

TEP 3.3. If MOVE = INTER-PROCESSOR DISPLACEMENT, rocessor p (Master) selects a cell CELL(p) with position POS(p) nd sends the data structure for CELL(p) along with the portion the cost function it has computed to processor q (Slave). Pro- ssor q selects a random position POS(q) within its area map and mputes the remainder of the cost function.

TEP 4.3. Compute $\Delta_{displace}(CELL(p),POS(q)) =$
$_1(WL,CELL(p),POS(q),q) + \Delta_2(AO,CELL(p),POS(p),p) +$
$_3(AO,CELL(p),POS(q),q) + \Delta_4(EO,CELL(p),POS(p),p) +$
$_5(EO,CELL(p),POS(q),q)$

TEP 5.3. Go to STEP 7.

STEP 3.4. If MOVE = INTRA-PROCESSOR DISPLACEMENT, processor p randomly selects a cell, $CELL(p)$, and a position, $POS(p)$, within its allocated area map.

STEP 4.4. Compute $\Delta_{displace}(CELL(p),POS(p)) = \Delta_1(WL,p) +$ $\Delta_2(AO,p) + \Delta_3(EO,p)$

STEP 7. Master accepts/rejects move using exponential function ACCEPT(Δ , T)

END PROCEDURE:

2.5.1. Discussion of moves

Mastership selection

For each pair of processors (p,q) connected in dimension i, one of them is chosen to be the Master and the other to be the Slave using the criteria listed in STEP 1 to ensure that the mastership of the pair alternates between processors in alternate iterations. The choice is not random as in [10] because it would then involve an extra message between the processors, and we wish to reduce the communication overhead as much as possible. We alternate mastership between iterations because otherwise in a fixed scheme, we would bias the displacements of cells from the Master to the Slave processor resulting in the Master processor having no cells after several iterations.

Selection of move

The ratio of cell displacements to cell exchanges has a profound effect on the quality of the final placement. The best results were observed to occur when the random selection favors displacements in a ratio of approximately 5 to 1 similar to the result reported in [2]. In addition, the Master decides if the exchange or displacement move will be an intra-processor (completely within the Master) or inter-processor (between the Master and the Slave). The best results were observed to occur when the number of intra-processor moves is equal to the number of inter-processor moves.

Cost calculation

We now discuss the cost function calculation for an inter-processor exchange, i.e. STEP 4.1., which is the most complicated and of all the move types. (The other move calculations are similar). We break up the task of calculating the cost of an inter-processor exchange move into 10 sub-tasks that are distributed equally among the Master and Slave processors. The first term, $\Delta_1(WL,CELL(p),POS(q),p)$ deals with the change in the wire length due to the movement of CELL(p) from POS(p) to POS(q). This is calculated by estimating the change in half the perimeter of the bounding box of each net. This term can be calculated by processor p alone since it keeps information about all the nets to which CELL(p) is connected, along with all the (x,y) locations of cells that are on the same nets, and can read POS(q) (which is the new (x,y) location for CELL(p)) from the message sent by processor q. The term $\Delta_2(WL,CELL(q),POS(p),q)$ relates to the change in wire length due to the movement of CELL(q) from POS(q) to POS(p), and is computed in an identical manner by processor q. The term $\Delta_3(AO,CELL(p),POS(p),p)$ deals with the change in the area overlap due to the movement of CELL(p) out of POS(p) and is calculated by processor p since it has information about all the cells that are near a given (x,y) location within processor p's area map. When CELL(p) is moved out of from location POS(p), it might remove some of the area overlaps. The term $\Delta_4(AO,CELL(p),POS(q),q)$ deals with the change in the area overlap due to the movement of CELL(p) into POS(q) and is calculated by processor q since it has information about all

ORIGINAL PAGE IS
OF POOR QUALITY

the cells that are near a given (x,y) location within processor p's area map. When CELL(p) is moved into location POS(p), it might create some of the area overlaps. The terms Δ_5 and Δ_6 are similar calculations for CELL(q). The term $\Delta_7(EO,CELL(p),POS(p),p)$ deals with the change in actual row length compared to desired row length (edge overshoot or undershoot) when CELL(p) is moved out of POS(p), and is calculated by processor p. The term $\Delta_8(EO,CELL(p),POS(q),q)$ deals with the change in edge overshoot/undershoot when CELL(p) is moved into POS(q), and is calculated by processor q. The terms Δ_9 and Δ_{10} are similar calculations for CELL(q).

2.6. Annealing schedule

In any simulated annealing algorithm, two important criteria are the choice of the initial temperature and the rate of decrease of the temperature. For the choice of the initial temperature, we adopted the heuristic that at the initial temperatures, we should accept 95% of all moves for which there is an increase in the cost function. Hence, prior to starting the actual annealing algorithm, we calculate the change in cost functions for $10 \times N$ (N = number of standard cells in circuit) single moves within the hypercube. The average change, Δ, is calculated for those moves in which the change in cost is positive. This average cost is then used to find a proper initial temperature:

$$T_{and} = -\frac{\Delta}{ln(0.95)}$$

The temperature of the system is then reduced after each stage of the algorithm according to the cooling schedule given by

$$T_{i+1} = \alpha(i)\,T_i$$

where α varies from 0.8 to 0.95 and decreases to 0.1 during the final stages of the algorithm. This variation is table-driven.

In order to enhance convergence during the later stages of the algorithm, a range limiting mechanism is incorporated similar to [2]. At high temperatures during the simulated annealing process, we do not restrict the distance over which exchanges and displacements of cells can occur. Gradually, as the temperature is decreased, for each processor, the range limit is also decreased accordingly until eventually certain dimensions of the hypercube are "frozen", i.e. changes between pairs of processors connected via those dimensions are effectively inhibited.

At each new temperature, the system is allowed to stabilize. This is accomplished by collectively attempting to generate a user specified number of new states per cell at each stage/temperature of the system. The final stopping criterion is satisfied when the temperature reaches a minimum value of 0.1.

2.7. Broadcasting New Cell Locations

Once the cells have been moved to new locations, these updated locations have to be sent to all processors so that they can update all net and pin information effected by the move. A very simple scheme was proposed in our earlier paper [10] which used the property of the existence of Hamiltonian circuits in the hypercube topology [16]. Each processor which had an updated cell location would inform its Hamiltonian circuit successor of the updated value of the cell location. This processor would then inform its Hamiltonian circuit successor which would do the same. It can be easily seen that if all P processors contained updated cell locations, it will take P time steps for all the updated cell locations to be available at all the processors. Since each message transfer is extremely expensive (it will be shown in Section 3.2 that while a move computation takes approximately 20-30 milliseconds, a message transfer between two adjacent nodes takes about 3-4 milliseconds) we decided to abandon this simple scheme and adapt a more complicated but extremely efficient one.

In the new scheme, each processor having a set of new cell locations broadcasts this information to all its log(P) neighbors in the first time step along its links in log(P) dimensions. In the next time step, the processors that have just received these messages from the first time step, forward the messages to their own neighbors connected via links in the higher most log(P)-i-1 dimensions where i equals the dimension of the link along which a message was received during the first time step. In the jth time step, all processors receiving messages from the $j-1^{st}$ time step forwards the messages to their neighbors in the higher most log(P)-i-1 dimensions where i again equals the dimension of the link along which a message was received during the $j-1^{st}$ time step. In the case of multiple initial processors wanting to broadcast modified cell locations, the messages are combined where needed at intermediate nodes before forwarding. This scheme guarantees that the broadcasting is completed in log(P) time steps without conflicts for links. Fig. 3a shows a 3-dimensional hypercube with labels on processing nodes and links. Fig. 3b shows the steps involved in broadcasting updated cell locations from processors 1, 2, and 7 which are labeled to as M1, M2, and M7 in Fig. 3b.

The entries in Fig. 3b are of the form Mi(j,k) which represents a message which originated from processor P_i during the first time step and going from processor P_j to P_k during the current time step. For example, in time step 2, message M7(6,4) which has originated from P_7 is transmitted from processor P_6 to P_4 along a dimension 1 link. It can be verified that all messages reach all processors within 3 time steps. In case of conflicts for using a particular link at a particular time step, messages are combined. For example, in time step 2, link 1,9 has two messages M1(0,4) and M2(0,4) which represent messages originating from processors P_1 and P_2 but going from P_0 to P_4 during time step 2.

A unique feature of our algorithm is that once messages are combined for transmission over a particular link, they need not be split up at intermediate nodes for transmission over separate links. The process of updating cell locations will take part at all nodes by extracting information from the received messages and using this information to modify local cell structures.

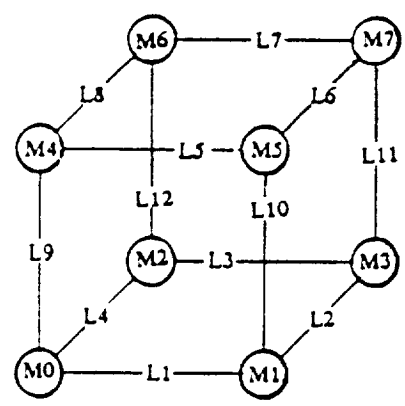


Fig. 3a. A 3-dimensional hypercube.

ORIGINAL PAGE IS
OF POOR QUALITY

link number	STEP 1	STEP 2	STEP 3
1.1	M1(1,0)		
1.2	M1(1,3)	M2(3,1)	
1.3	M2(2,3)		
1.4	M2(2,0)	M1(0,2)	
1.5			
1.6	M7(7,5)		
1.7	M7(7,6)		
1.8		M7(6,4)	
1.9		M1(0,4) M2(0,4)	M7(4,0)
1.10	M1(1,5)	M7(5,1)	M2(1,5)
1.11	M7(7,3)	M1(3,7) M2(3,7)	
1.12	M2(2,6)	M7(6,2)	M1(2,6)

Fig. 3b. Broadcast steps for a 3-dimensional hypercube on a message from nodes 1, 2, and 7.

ALGORITHM IMPLEMENTATION AND PERFORMANCE

The advantage of our algorithm over TimberWolf is that it much faster. We have implemented the algorithm in about 00 lines of C code. Due to the unavailability of an actual Intel percube at the present time at the University of Illinois, initial ting of this algorithm has been completed using the Intel iPSC nulator running on a SUN 3/50 workstation system under IX 4.2 [17]. Initial algorithm testing has only been attempted a small scale due to excessive simulator execution times.

. Placement results

It should be noted that in the parallel annealing scheme, ce we have deviated from the serial acceptance of moves, we not assume the convergence properties of the annealing algo- hms to be valid. The theoretical convergence properties are ll a subject of future research. However, we have experi- ented with a wide variety of standard cell circuits, some of ich were randomly generated, others were obtained from dustry and universities.

In Table 1, we show the results of our parallel placement gorithm on a 4-dimensional hypercube for four standard cell cuits. We have also implemented a uniprocessor version of the nulated annealing algorithm which is slightly simpler than mberWolf in that the only moves that are allowed are changes and displacements and only standard cells are handled macro-cells or pads). At each temperature of the annealing ocess, approximately 100 new states were attempted per cell. ir parallel algorithm gives a final placement cost that is 10-20% tter.

We studied the effect of the parallel simulated annealing at ch temperature. We validated empirically that even though we e performing the accepts/rejects on the basis of outdated infor- ation, our algorithm has the same general convergence property the uniprocessor algorithm. From our studies, we observed at in the initial stages of the algorithm (higher temperatures), a rge percentage of both types of moves are accepted. As the mperature is decreased, the percentage of acceptances both types of moves decreases. However, at extremely low temperatures, the percentage of acceptances of displacements increases with practically no acceptance for exchanges. The increase in the acceptance of displacements is primarily due to only intra-processor displacements being attempted as governed by the implemented range limiter.

Table 1. Final placement wiring length comparison.

number cells	4-dim Hypercube	Uniprocessor (TimberWolf)	Percentage improvement
64	29248	32135	10%
183	63094	76498	21%
286	96778	115359	19%
469	159759	195066	22%

3.2. Timing Estimates

Since we did not have access to an Intel Hypercube at the University of Illinois to evaluate the speedup of our algorithm, we present here an estimate of the expected speedup. The Intel Simulator does not give any timing information for message communication so timing has to be estimated from other sources. The running time of our algorithm depends on two separate components: Computation and Communication. We will present estimates of both in the following sections.

Computation

To evaluate the computation cost per move (exchange and displace ment), we implemented our algorithm on a single processor of the Intel hypercube simulator. We performed 1000 random moves of both the exchange and displacement class and evaluated an average computation time. The CLOCK command in the simulator gives the running time on the machine on which the simulator is running, which was a SUN 3/50 workstation using a Motorola 68020 CPU which is rated to be 2.7MIPs [18]. The Intel Hypercube Nodes consist of Intel 80286 CPUs which have been reported to be 0.78MIPS[19] or 3.5 times slower than the Motorola 68020 for the types of computation performed in our algorithm. Hence, the computation time per move on the Intel Hypercube was estimated to be as shown in Table 2.

Table 2. Computation times (milliseconds) on hypercube node.

number cells	Intra Disp.	Inter Displace M	Inter Displace S	Intra Exchange	Inter Exch. M	Inter Exch. S
64	15	9	12	21	12	9
183	24	12	15	30	24	21
286	30	15	21	33	27	24
469	30	18	24	33	27	24
800	33	18	24	33	30	27
2357	33	21	27	39	30	27

Communication costs

We will use the results of some benchmark studies performed by Reed and Grunwald at the University of Illinois on communication costs on the Intel iPSC [20]. The results are summarized in Fig. 4 which shows the delay in transfer of messages of varying size for simultaneous exchanges and unidirectional message transfers along a link . We therefore need to estimate what the average packet size will be for different types of messages in order to determine communication costs. During the distributed cost calculation phase, the entire data structure for a candidate cell is sent to a neighboring processor over a single link in the hypercube. Table 3 shows the range of message sizes for various size standard cell circuits and corresponding communication times derived from Fig. 4.

Expected Speedup

By combining these timing results and taking into account the parallelism involved in the calculation of the move cost, the time to complete each of the four types of moves was calculated

ORIGINAL PAGE IS
OF POOR QUALITY

Table 3. Estimation of communication costs from size of messages.

number cells	Message length (bytes) min	max	avg	link delay
64	99	688	448	2.8 ms
183	68	792	272	2.5 ms
286	36	844	214	2.4 ms
469	76	724	250	2.5 ms
800	68	1732	473	2.8 ms
2357	36	792	254	2.5 ms

Table 4. Estimate of time to complete the four types of moves in milli-seconds using Intel hypercube

number cells	Intra Displace	Inter Displace	Intra Exchange	Inter Exchange
64	15.0	15.6	21.0	15.6
183	24.0	18.1	30.0	27.1
286	30.0	24.0	33.0	30.0
469	30.0	27.0	33.0	30.0
800	33.0	27.6	33.0	33.6
2357	33.0	30.0	39.0	33.0

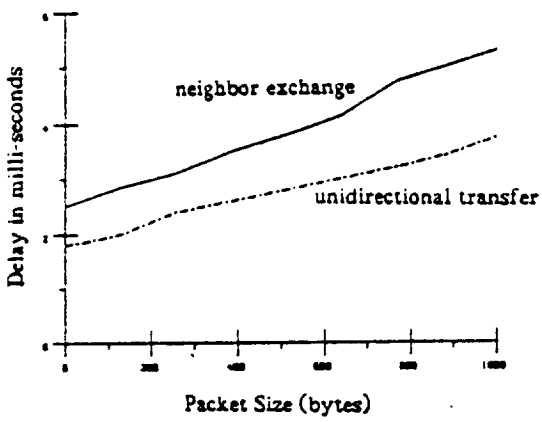


Fig. 4. Link delay for various packet sizes.

as given in Table 4. The time required to broadcast updated cell information has been shown in Section 2.7 to require only log(P) communication steps. A complete broadcast cycle for a 6-dimensional hypercube should therefore require 18.2 milli-seconds. Unfortunately, each node in the Intel Hypercube is not able to actively use all of its log(P) links at the same time due to architectural limitations. Thus the actual number of simultaneous messages that can be transmitted/received will be somewhere between 2 and log(P). In the worst case only a single exchange of data between processing nodes can occur hence a complete broadcast cycle for a 6-dimensional hypercube will require 64.7 milli-seconds.

Using these estimates we can determine the expected speedup of our parallel algorithm over a similar uniprocessor version. If our algorithm were to be run on a 6-dimensional hypercube using the 800-cell standard circuit then at each iteration 32 parallel moves will be attempted. It is to be expected that at least one of these moves will be an intra-processor exchange which will be the bottleneck in terms of timing. The time to complete these 32 moves and update will be between 51.2 ms and 97.7 ms depending on update broadcast timing. For a uniprocessor version of this algorithm the 32 moves will be distributed in a 5 to 1 ratio between displacements and exchanges. Computational time will thus be $25.6 \times 33 + 6.4 \times 33 + 16 = 1072$ ms with the additional 16 ms added for time to complete updating of cell structures. In the hypercube this updating is done while waiting for communications. Using these results the estimated speedup of the Intel hypercube over the uniprocessor version will be somewhere between 11 and 21. Speedup estimates for the other standard circuits are given in Table 5.

4. CONCLUSIONS

In this paper we have presented a parallel version of the simulated annealing technique for solving the standard cell placement problem that is targeted to run in a local memory message passing parallel processing environment, namely the Hypercube computer. We have presented an improved algorithm that reduces the communication overhead, can handle more features of the placement problem, and is specifically targeted to run on the Intel Hypercube. We have presented a novel tree broadcasting strategy for the hypercube that is used extensively in our algorithm for updating cell locations in the parallel environment. We have implemented the algorithm on an Intel hypercube simulator. We reported on the performance of our algorithm on actual standard cells used in industry. We also presented some accurate estimates of the execution time for the algorithm. Our algorithm will not give rise to oscillations because we have a number of cells assigned to each processor, and cells are chosen randomly for possible exchange. Unlike the conventional array algorithms for module placement, our proposed algorithm will thus not get stuck at local minima. The possibility of choosing the same pair of cells for repeated exchange (oscillations) is very low. Cell exchanges can be performed among nearest neighbors through our novel area mapping technique and also between cells that are

Table 5. Time to complete 32 moves in milli-seconds.

number cells	uniprocessor	6-dim. cube min	max	speedup min	max
64	528	39.2	85.7	6.2	13.5
183	817	48.2	94.7	8.6	17.0
286	991	51.2	97.7	10.1	19.4
469	993	51.2	97.7	10.2	19.4
800	1072	51.2	97.7	11.0	20.9
2357	1102	57.2	103.7	10.6	19.3

large distances away. The results show that our parallel algorithm is not only faster but also gives better final placement results than the uniprocessor simulated annealing algorithms.

Future research involves evaluating the algorithm on an actual Intel Hypercube. It would be interesting to explore the impact of different hypercube architectures such as the NCUBE/10, the Ametek S/14 and the Caltech MARK II on our algorithm. We are also investigating improved parallel algorithms that reduce the communication costs. Eventually, we plan to develop an integrated placement and routing package on the Hypercube.

ORIGINAL PAGE IS
OF POOR QUALITY

REFERENCES

C. Sechen and A. S. Vincentelli, "The TimberWolf Placement and Routing Package," *Proc. Custom Integrated Circuits Conf.*, pp. 522-527, May 1984.

C. Sechen and A. S. Vincentelli, "TimberWolf3.2: A New Standard Cell Placement and Global Routing Package," *Proc. 23rd Design Automation Conf.*, pp. 432-439, Jun. 1986.

E. H. L. Aarts, F. M. J. de Bont, E. H. A. Habers, and P. J. M. van Laarhoven, "Parallel Implementations of the Statistical Cooling Algorithm," *Integration, the VLSI Journal*, vol. 4, pp. 209-238, 1986.

E. Felten, S. Karlin, and S. W. Otto, "The Traveling Salesman Problem on a Hypercubic, MIMD Computer," *Proc. 1985 Parallel Processing Conf.*, pp. 6-10, Aug. 1985.

M. J. Chung and K. K. Rao, "Parallel Simulated Annealing for Partitioning and Routing," *Proc. IEEE Int. Conf. on Computer Design (ICCD-86)*, pp. 238-242, Oct. 1986.

A. Casotto, F. Romeo, and A. S. Vincentelli, "A Parallel Simulated Annealing Algorithm for the Placement of Macro-Cells," *Proc. Int. Conf. on Computer-Aided Design*, Nov. 1986.

S. Devadas and A. R. Newton, "Topological Optimization of Multiple Level Array Logic: On Uni- and Multi-processors," *Proc. Int. Conf. Computer-Aided Design (ICCAD-86)*, pp. 38-41, Nov. 1986.

S. A. Kravitz and R. A. Rutenbar, "Multiprocesssor-Based Placement by Simulated Annealing," *Proc. 23rd Design Automation Conf.*, pp. 567-573, Jun. 1986.

R. A. Rutenbar and S. A. Kravitz, "Layout by Annealing in a Parallel Environment," *Proc. IEEE Int. Conf. on Computer Design (ICCD-86)*, pp. 434-437, Oct. 1986.

P. Banerjee and M. Jones, "A Parallel Simulated Annealing for Standard Cell Placement on a Hypercube Computer," *Proc. IEEE Int. Conf. Computer-Aided Design (ICCAD-86)*, Nov. 1986.

[11] J. S. Rose, D. R. Blythe, W. M. Snelgrove, and Z. G. Vranesic, "Fast, High Quality VLSI Placement on a MIMD Multiprocessor," *Proc. Int. Conf. Computer-Aided Design (ICCAD-86)*, pp. 42-45, Nov. 1986.

[12] J. Tuazon, J. Peterson, M. Pniel, and D. Lieberman, "Caltech/JPL Mark II Hypercube Concurrent Processor," *Proc. 1985 Parallel Processing Conference*, pp. 666-673, Aug. 1985.

[13] J. C. Peterson, J. Tuazon, D. Lieberman, and M. Pniel, "The Mark III Hypercube-Ensemble Concurrent Computer," *Proc. 1985 Parallel Processing Conference*, pp. 71-73, Aug. 1985.

[14] Intel Scientific Computers, "iPSC: The First Family of Concurrent Supercomputers," 1985, product announcement.

[15] *Ametek System 14 User's Guide C Edition Version 2.0*, Ametek Computer Research Div., May 1986.

[16] N. Deo, in *Graph Theory with Applications to Engineering and Computer Science*, Englewoods Cliffs, N.J.: Prentice-Hall, Inc., 1974.

[17] Intel Corporation, "Hypercube Simulator Version 2.1," 310175-002, Jun. 1986.

[18] D. MacGregor, D. Mothersole, and B. Moyer, "The Motorola MC68020," *IEEE Micro*, pp. 101-118, Aug. 1984.

[19] Intel Corporation, "Introduction to the iAPX 286," 210308-001, Feb. 1982.

[20] D. A. Reed and D. C. Grunwald, "Benchmarking Hypercube Hardware and Software," *SIAM 2nd Conf. on Hypercube Multiprocessors (to appear)*, 1986.

ORIGINAL PAGE IS
OF POOR QUALITY

Proc. Int. Conf. on Computer Design, New York, NY, pp. 83-86, October, 1987.

AN IMPROVED SIMULATED ANNEALING ALGORITHM FOR STANDARD CELL PLACEMENT

Mark H. Jones and Prithviraj Banerjee

Computer Systems Group
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign

ABSTRACT

The simulated annealing technique has been applied extensively in recent years to find globally minimum cost solutions to very large optimization problems. In VLSI design the primary optimization problem involves the placement of standard logic cells in such a way that the total length of required wire interconnecting the cells is minimized. Several recent papers have parallelized the simulated annealing placement algorithm in an attempt to reduce execution time. It has been shown that these parallel algorithms are not only faster but also converge to a more optimal final placement than a similar uniprocessor version. This paper proposes a uniprocessor version of the simulated annealing placement algorithm which is based on the results obtained from parallelization of the simulated annealing algorithm. A modified generate function is proposed to allow for nonuniformly distributed distances over which exchange and displacement moves can take place; this technique is called *multi-windowing*. Multiple cell movements are considered before updating cell placement data, hence the cost calculations are based on slightly outdated placement data. This technique is referred to as *pseudo-parallel moves*. This new algorithm has been implemented and has been found to achieve better final placement values than conventional simulated annealing algorithms.

I. INTRODUCTION

Simulated annealing, a general purpose method of multivariate optimization, has been applied extensively in recent years to find globally minimum cost solutions to very large optimization problems [1]. In VLSI design, this technique has been applied to the problem of placing standard logic cells in a VLSI chip so that the total interconnection wire length is minimized [2,3,4]. Theoretical studies have shown that a global optimum can be reached with probability one provided a set of conditions on the annealing schedule are satisfied. The real disadvantage of the simulated annealing approach is the massive computing time required to converge to an optimal solution. Various approaches have been proposed to speed up the simulated annealing process.

In the first approach, careful control of the annealing process is used to improve the computing time [5]. Theoretical studies have been directed at deriving optimal cooling schedules on the basis of viewing the annealing process as Markov chains [6,7]. A second approach to reduce the computing time is to employ clever move-sets based on the idea of range limiting or changes in the cost function to reduce the chances of generating a next state that is going to be rejected [3,8,9]. A third approach is to investigate techniques for speeding up simulated annealing algorithms by running them on parallel processor systems [10,11,12,13,14]. Many of these parallel algorithms have been found to be considerably faster, and to converge to a final placement which is better than uniprocessor simulated annealing algorithms that attempt aggressive cooling schedules to reduce computing times. The better performance of the parallel algorithms appears to be caused by the constraints the parallel implementations place on the distances over which moves can occur and the use of slightly outdated cell placement information caused by multiple interacting moves being accepted at each parallel iteration.

In this paper, we present an improved uniprocessor standard cell placement algorithm that takes advantage of the performance enhancements that appear to come from parallelizing the uniprocessor simulated annealing algorithm. Two important differences between this algorithm and the previously reported uniprocessor algorithms are incorporated: (1) a modified generate function is proposed to allow for nonuniformly distributed distances over which exchange and displacement moves can take place; this technique is called *multi-windowing*; (2) multiple cell movements are considered before updating cell placement data, hence the cost calculations are based on slightly outdated placement data; this technique is referred to as *pseudo-parallel moves*.

2. REVIEW OF PARALLEL ALGORITHM ON HYPERCUBE

Before going into the details of the new uniprocessor algorithm, we will review the parallel simulated annealing algorithm on a hypercube multiprocessor to motivate the choice of various heuristics [14]. Each processor in the hypercube is assigned an approximately equal area portion of the total chip area which can be viewed as a virtual square grid. The correspondence between processor addresses and grid points on the physical chip area is chosen such that the processors that are adjacent in a predetermined set of four dimensions of the hypercube allow all nearest North-South-East-West neighbor cell displacements or exchanges. The remaining dimensions of the hypercube are used for displacements or exchanges across larger distances in the area map. After the cells have been evenly distributed among the processors, $P/2$ pairs of processors perform inter-processor or intra-processor cell exchange or displacement moves for each dimension of the hypercube. The rest of the annealing algorithm is very similar to the conventional simulated annealing algorithms.

Observations

Two points differentiate this parallel algorithm from the corresponding serial algorithm. First, the parallel moves are accepted or rejected assuming that there is no interaction between moves. The estimate of the cost of a move is based on outdated information because it is assumed that cells in other processors do not move while a cell is being considered for a move by a processor. However, the information is never allowed to be very outdated because after completion of $P/2$ pairs of moves, the new cell locations are broadcast to all processors in the hypercube.

Second, cell exchanges and displacements are only allowed between processors (and therefore to corresponding area maps)

Acknowledgment: This research was supported in part by the National Aeronautics and Space Administration under Contract NASA NAG 1-613.

PRECEDING PAGE BLANK NOT FILMED

U.S. Government Work. Not protected by U.S. copyright.

that are directly connected by the hypercube topology. In a uniprocessor algorithm, a cell can potentially be moved to any other location in the chip in a single move. Hence, the parallel algorithm has a restricted set of moves.

We have experimentally observed [14] that in spite of the differences between the parallel and serial algorithms, the parallel algorithm gives better final placements *for the same cooling schedule*. We have also observed that the quality of the solution improves as the number of processors of the hypercube increases for the same cooling schedule. It is obvious that given sufficient time, the serial algorithms would converge to the same optimal solution. In practice, one can rarely afford the large computing resources that are required of such serial algorithms. Hence, for aggressive cooling schedules, there is a possibility of these serial algorithms not reaching the global optimum.

3. IMPROVED UNIPROCESSOR ALGORITHM

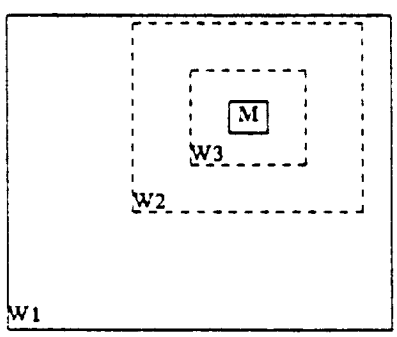
In this section, we propose a new heuristic for a serial simulated annealing algorithm that would model the above characteristics of the parallel algorithm and thereby try to achieve similar improvements in the quality of the final solution. An outline of the improved uniprocessor simulated annealing algorithm is given below.

STEP 1. WHILE "Stopping criteria" not reached
STEP 2. Generate new temperature
STEP 3. FOR inn_loop_cnt = 1 to user_param
STEP 4. Generate new move using multi-windowing technique
STEP 5. Evaluate/accept/reject move
STEP 6. IF number of accepted moves equals LIMIT
THEN update all saved cell positions and zero number of accepted moves counter
ELSE increment accepted moves counter and save cell movements in temporary storage

Two points distinguish it from the conventional simulated annealing algorithms for cell placement: the concepts of multi-windowing and pseudo-parallel moves.

Use of Multiwindowing

In the parallel versions of the simulated annealing placement algorithm, it appears that movement of cells should be biased so that they are restricted more to their local vicinity. We can model this phenomenon in a uniprocessor by complex windowing techniques and distance probability distributions.. For example, in Fig. 1, if cell M was picked to perform a displacement, a simple triple windowing scheme could be used to determine where the cell will be displaced to. The outermost window (W1) is always equal to the physical work space of the circuit. The inner windows, W2 and W3, have sizes proportional to 2/3 and 1/3 of the physical work space and are centered about cell M. In order to favor local movement, the probability of being in the innermost window (W3) is made greater than being in the outer windows. For example, cell M may have a 50% probability that its proposed new position will be within the innermost window W3, a 25% probability of being within window W2 but not window W3, and a 25% probability of being in the physical work space but not within windows W2 or W3.



window	fraction of Max Dimension	example size	probability within window
W1	1	120 X 10	25%
W2	2/3	72 X 60	25%
W3	1/3	36 X 30	50%

Fig. 1. Example use of windowing for determining cell movement for cell M.

Use of Pseudoparallel Moves

A conditional data update statement has been added in the proposed algorithm which allows a multiple number of accepted moves to accumulate before an update of the circuits placement is done. This amounts to allowing all moves after the first successful move determine the cost function on the basis of outdated placement information. For example, in Fig. 2(a), if module M1 is successful in performing a displacement from (x_1, y_1) to (x_2, y_2) during the first iteration of the inner loop, then the circuit should be as shown in Fig. 2(b), but because M1's position is not updated, the remainder of the cells still calculate cost functions which involve M1 as though it were still at position (x_1, y_1). Because of this, if module M2, which is connected to M1 via a net connection, is chosen for an attempted move during iteration two, then the half-perimeter wiring cost associated with the net will be computed using the old position of M1. After each move acceptance, a counter is incremented to keep track of the number of successful moves, since the last cell position update and the new positions of the cells are placed in temporary storage for use later in updating the cell positions. Random cell selection for movement in subsequent iterations is not able to select cells which have made successful moves, but whose positions have not yet been updated. This amounts to freezing the cells' positions

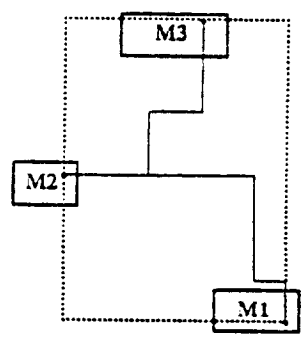


Fig. 2(a). Original net placement.

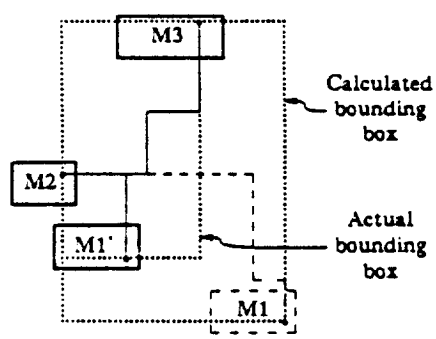


Fig. 2(b). Placement after initial acceptance.

until the required number of moves has been accepted. The effect of using slightly outdated information appears to give a higher probability of getting out of local minima, since this technique will accept a higher percentage of moves with uphill changes in the cost function.

4. RESULTS

The advantage of this algorithm over conventional, uniprocessor simulated annealing algorithms such as TimberWolf is that it converges to a better final placement in a given number of iterations. We have implemented the algorithm in the C programming language on a SUN 3/50 workstation, running under UNIX 4.2.

To ascertain the effects of using multiwindowing and multiple moves before update, initial testing was performed on a 64 cell circuit that was randomly generated such that it had several known clusters of high connectivity. A number of tests were run in order to derive the optimal combination of these two techniques. The first set of tests was concerned with determining the optimal number of moves that should be accepted before cell updating is performed. Table 1 shows the final placement cost associated with waiting for various numbers of multiple moves before cell placement update. For these tests no windowing was attempted and thus, movement of cells was unrestricted and uniformly distributed. The final placement cost was found to decrease as the number of multiple moves is increased up to a certain point after which it increased. The optimal solution occurs when 16 moves have to be accepted before placement update will occur.

A variety of windowing schemes were experimented with as shown in Tables 2 through 4. The number and size of windows for each test vary over a wide range. For example, in the ninth entry of Table 3, a triple windowing scheme is used with the largest window being equal to the physical work space, the second window being equal to 2/3 the size of the physical work space, and the third window equal to 1/3 the size of the physical work space. The two smaller windows are centered about the candidate cell for movement. The probability of a cell moving to within each of these windows but not smaller subwindows is distributed as 8%, 24%, and 68% respectively. The final cost using this windowing scheme is 21643 which is 10.2% less than would be derived by an algorithm which does not use multiple moves or windowing.

Table 1. Cost vs number of multiple moves for 64 cell circuit.

Number of Multiple Moves	Final Placement Cost	Percentage Change
1	24125	+0.0%
2	24138	+0.1%
4	24003	-0.5%
8	23984	-0.6%
	12	23924
16	23821	-1.3%
24	24173	+0.2%
32	24829	+2.9%

Table 2. Comparison of cost vs. distribution for (1 : ¼) double window.

Number Windows	Window Sizes as Fraction of max	Distribution of moves in Windows	Final Cost	Percent Change
1	1	100%	23821	-1.3%
2	1 : ¼	50% : 50%	24017	-0.4%
2	1 : ¼	33% : 67%	23715	-1.7%
2	1 : ¼	25% : 75%	22783	-5.6%
2	1 : ¼	20% : 80%	22893	-5.1%
2	1 : ¼	0% : 100%	23990	-0.6%

Table 3. Comparison of cost vs distribution for (1 : 2/3 : 1/3) triple window.

Number Windows	Window Sizes as Fraction of max	Distribution of moves in Windows	Final Cost	Percent Change
1	1	100%	23821	-1.3%
3	1 : 2/3 : 1/3	17% : 35% : 48%	23654	-2.0%
3	1 : 2/3 : 1/3	12% : 24% : 64%	23104	-4.2%
3	1 : 2/3 : 1/3	10% : 20% : 70%	23249	-3.6%
3	1 : 2/3 : 1/3	9% : 18% : 73%	23187	-3.9%
3	1 : 2/3 : 1/3	12% : 36% : 52%	22731	-5.8%
3	1 : 2/3 : 1/3	10% : 30% : 60%	22149	-8.2%
3	1 : 2/3 : 1/3	8% : 27% : 64%	22314	-7.5%
3	1 : 2/3 : 1/3	8% : 24% : 68%	21643	-10.2%
3	1 : 2/3 : 1/3	10% : 40% : 50%	23007	-4.6%
3	1 : 2/3 : 1/3	8% : 40% : 52%	23721	-1.7%

Table 4. Comparison of cost vs distribution for (1 : ¾ : ½ : ¼) quadruple window.

Number Windows	Window Sizes as Fraction of max	Distribution of moves in Windows	Final Cost	Percent Change
1	1	100%	23821	-1.3%
4	1 : ¾ : ½ : ¼	10% : 20% : 30% : 40%	23764	-1.5%
4	1 : ¾ : ½ : ¼	9% : 18% : 27% : 46%	23521	-2.5%
4	1 : ¾ : ½ : ¼	8% : 16% : 24% : 52%	23902	-0.9%
4	1 : ¾ : ½ : ¼	8% : 16% : 32% : 44%	23782	-1.4%
4	1 : ¾ : ½ : ¼	6% : 12% : 24% : 58%	22813	-5.4%
4	1 : ¾ : ½ : ¼	5% : 10% : 20% : 65%	21032	-12.8%
4	1 : ¾ : ½ : ¼	4% : 8% : 12% : 76%	22314	-7.5%
4	1 : ¾ : ½ : ¼	0% : 12% : 24% : 64%	23431	-2.9%
4	1 : ¾ : ½ : ¼	0% : 10% : 20% : 70%	22946	-4.9%
4	1 : ¾ : ½ : ¼	0% : 6% : 24% : 70%	23007	-4.6%

Several generalized results were observed from our studies, all of which cannot be presented for space limitations. In comparing results of studies with double windowing for different relative sizes of the inner window, the final placements tended to be decidedly inferior where the inner windows are not significantly smaller than the physical work space. This is in agreement with an earlier observation that the movement of cells should be localized to the area immediately surrounding the cell. In all the windowing schemes, better performance is generally obtained as the percentage of localized moves is increased. It appears that a larger number of windows will give the best final placement results if the probability of movement farther away from the initial position decreases at least linearly with distance. The best final placement was obtained using a quadruple windowing technique with 16 multiple moves before update.

A few of the more promising windowing schemes were applied to a larger standard cell circuit obtained from UC Berkeley which has 183 cells and is required to be placed in 13 rows with a desired row length of 1398 units. We ran our program on a SUN 3/50 workstation and compared our results with the TimberWolf3.2 program with the global routing option suppressed and row separation set to zero. The cooling schedule for all the programs was the same. The initial temperature was 4,000,000; the inner-loop count set to 18300; the temperature reductions were identical to those of Timberwolf3.2. The version of our program with a single window and cell updates after every move (first row in Table 5) is similar to TimberWolf3.2. However, as

Table 5. Comparison among various versions and TimberWolf3.2 for an example 183 cell circuit.

Number moves	Number windows	Final Cost	Time (sec.)
1	1	78775	44605
16	1	87851	44496
16	2	77657	51584
16	4	73000	56374
Timb3.2		73566	9669

can be seen from the comparisons of running times between our program and TimberWolf3.2, our implementation is much slower. This is because the whole program was implemented rather crudely by a student in two months. We did not use any clever programming techniques that are used in TimberWolf3.2 to speed up the program. The purpose of our study was to demonstrate that our algorithm gets the the same approximate result as TimberWolf3.2 without multi-windowing and pseudo-parallel moves and then show the effect of these heuristics.

CONCLUSIONS

This paper has proposed an improved uniprocessor simulated annealing placement algorithm which is based on the results obtained from parallelization of the simulated annealing algorithm proposed earlier by us and other researchers. These features involve incorporating two techniques: multi-windowing and pseudo-parallel moves. A variety of windowing schemes and size of windows have been extensively tested, results of some of which have been reported in this paper. It appears that a larger number of windows will give the best final placement results if the probability of movement farther away from the initial position decreases at least linearly with distance. The best final placement was obtained using a quadruple windowing technique with 16 multiple moves before update. Our results show that the algorithm is better (in terms of final placement quality for a fixed number of moves) than conventional algorithms.

REFERENCES

[1] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, pp. 671-680, May 1983.

[2] C. Sechen and A. S. Vincentelli, "The TimberWolf Placement and Routing Package," *Proc. Custom Integrated Circuits Conf.*, pp. 522-527, May 1984.

[3] C. Sechen and A. S. Vincentelli, "TimberWolf3.2: A New Standard Cell Placement and Global Routing Package," *Proc. 23rd Design Automation Conf.*, pp. 432-439, Jun. 1986.

[4] L. K. Grover, "A New Simulated Annealing Algorithm for Standard Cell Placement," *Proc. Int. Conf. on Computer-Aided Design*, pp. 378-380, Nov. 1986.

[5] S. R. White, "Concepts of Scale in Simulated Annealing," *Proc. Int. Conf. Computer Design*, pp. 646-650, Oct. 1984.

[6] D. Mitra, F. Romeo, and A. Sangiovanni-Vincentelli, "Convergence and Finite-Time Behavior of Simulated Annealing," *Proc. 1985 Decision and Control Conf.*, Dec. 1985.

[7] M. D. Huang, F. Romeo, and A. Sangiovanni-Vincentelli, "An Efficient General Cooling Schedule for Simulated Annealing," *Proc. Int. Conf. Computer-Aided Design (ICCAD86)*, pp. 381-384, Nov. 1986.

[8] R. H. J. M. Otten and L. P. P. P. van Ginneken, "Simulated Annealing: The Algorithm," 1984, unpublished manuscript.

[9] J. W. Greene and K. J. Supowit, "Simulated Annealing without Rejected Moves," *Proc. Int. Conf. Computer Design (ICCD84)*, pp. 658-663, Oct. 1984.

[10] S. A. Kravitz and R. A. Rutenbar, "Multiprocesssor-Based Placement by Simulated Annealing," *Proc. 23rd Design Automation Conf.*, pp. 567-573, Jun. 1986.

[11] R. A. Rutenbar and S. A. Kravitz, "Layout by Annealing in a Parallel Environment," *Proc. IEEE Int. Conf. on Computer Design (ICCD-86)*, pp. 434-437, Oct. 1986.

[12] A. Casotto, F. Romeo, and A. S. Vincentelli, "A Parallel Simulated Annealing Algorithm for the Placement of Macro-Cells," *Proc. Int. Conf. on Computer-Aided Design*, Nov. 1986.

[13] P. Banerjee and M. Jones, "A Parallel Simulated Annealing for Standard Cell Placement on a Hypercube Computer," *Proc. IEEE Int. Conf. Computer-Aided Design (ICCAD-86)*, Nov. 1986.

[14] M. Jones and P. Banerjee, "Performance of a Parallel Algorithm for Standard Cell Placement on the Intel Hypercube," *Proc. 24th Design Automation Conf.*, June 1987.

PRECEDING PAGE BLANK NOT FILMED

From: Proceedings of the Second International Symposium on Methodologies for Intelligent Systems.

THE "RID-REDUNDANT" PROCEDURE IN C-PROLOG

N94-70421

*Huo-yan Chen** and *Benjamin W. Wah***

Coordinated Science Laboratory and

Department of Electrical and Computer Engineering

University of Illinois at Urbana-Champaign

1101 W. Springfield Avenue

Urbana, Illinois 61801, U.S.A.

ABSTRACT

C-Prolog can conveniently be used for logical inferences on knowledge bases. However, as similar to many search methods using backward chaining, a large number of redundant computation may be produced in recursive calls. To overcome this problem, we have designed the "rid-redundant" procedure to rid all redundant computations in running multi-recursive procedures. Experimental results obtained for C-Prolog on the Vax 11/780 computer show that there is an order of magnitude improvement in the running time and solvable problem size.

KEYWORDS: Backward chaining, C-Prolog, multi-recursive procedures, redundancy.

1. INTRODUCTION

Previous research indicated that efficiency is one of the critical factors that affect the practicality of logic programming as a software engineering methodology [6,7,8,9,10]. One of the reasons that degrade efficiency is the many redundant computations that are often associated with a backward chaining search method. In this paper we study the elimination of redundant computations in C-Prolog programs.

In the C-Prolog interpreter, the inference engine always selects in each step the leftmost unevaluated subgoal in the current goal and matches clauses in the order of appearance in the program text [1,2,4]. According to this standard control strategy, there are a lot of repeated redundant computations in running recursive procedures.

Example 1. The C-Prolog program for p(N,A), which represents the predicate "N'th Fibonacci number is A," is as follows.

```
C1: p(1,1).
C2: p(2,1).
C3: p(N,A) :- N1 is N-1, N2 is N-2, p(N1,A1), p(N2,A2), A is A1+A2.
```

For simplicity, we use the following C3' instead of C3, although it is invalid in C-Prolog.

```
C3': p(N,A) :- p(N-1,A1), p(N-2,A2), A is A1+A2.
```

In evaluating p(5,A) using the C-Prolog interpreter, the following computations will be produced.

```
?- p(5,A).
```

Invoking C3' under the unifier {5/N}, we get

---

* Visiting research scholar from Department of Computer Science, Jinan University, Guangzhou, P.R.China

** Research supported by National Aeronautics and Space Administration Grant NAG 1-613.

International Symposium on Methodologies for Intelligent Systems, Charlotte, North Carolina, U.S.A., October 1987.

? p(4,A1), p(3,A2), A is A1+A2.

Invoking C3' under the unifier {4/N, A1/A}, we get

? p(3,A11), p(2,A12), A1 is A11+A12, p(3,A2), A is A1+A2.

Here, p(3,X) is evaluated twice: the result of first time is in A11, and the result of second time is in A2. The number of repeated computations will increase greatly when N increases. This increases the running time and reduces the solvable problem size.

There are two methods to eliminate these redundant computations. First, the interpreter [11] can be revised, so it checks every subgoal before evaluating it to determine whether it has been evaluated before. The subgoal will be evaluated and results stored in a knowledge base if it has not been evaluated before. If results of the subgoal are already available in the knowledge base due to a previous evaluation, they will be accessed. The run-time overhead of this method is high, as every subgoal evaluated has to be checked. A second method is to determine by a pre-interpreter all possible redundant predicates before the program is executed. The determination of results already evaluated during execution is similar to that of the first method. Naughton has recently proved that detecting and removing redundant predicates is undecidable in general; that is, there is no algorithm that will find and remove all redundant predicates in an arbitrary C-Prolog program in effective time and space [12]. For a subclass of single-recursive procedures or linear-recursive procedures, which are procedures that only call themselves once in their definitions, he presented a linear-time algorithm to detect and remove redundant predicates [13]. In this paper we discuss the case of multi-recursive procedures. These are procedures that invoke themselves more than once in their definitions. As we are studying the simple case of procedures that call themselves recursively, the identification of possible redundant predicates is trivial, and the second method degenerates into the first method. The technique that we will use is similar to that of dominance in dynamic programming algorithms [5].

2. THE "RID-REDUNDANT" PROCEDURE

The "rid-redundant" procedure is written in C-Prolog and is called with the following form.

rid-redundant([Q,M,X],M0).

where Q, M, and M0 are its input parameters, and X is its output parameter. Q represents the given multi-recursive procedure, which must be defined in C-Prolog. M indicates the variable of recursion in the given multi-recursive procedure. M0 is the maximum of the variable M in the all facts of the definition of the given multi-recursive procedure. M0 is said to be cardinal of the given multi-recursive procedure. X is the output of the given multi-recursive procedure, which can be in any valid data type. For instance, Q, M, M0, and X in Example 1 are p, N, 2, and A, respectively.

The methodology to eliminate redundant computations is as follows.

(1) Design all clauses in the target program using C-Prolog.

(2) If one subgoal calls a clause with head p(N,A) that is multi-recursive and has cardinal N0, that is, p(N,A) invokes itself in its definition, then "p(N,A)" is replaced at the calling point by "rid-redundant([p,N,A],N0)."

(3) If the initial query invokes subgoal q(N,A), which is a multi-recursive procedure and has cardinal N0, then "q(N,A)" is replaced by "rid-redundant([q,N,A],N0)."

3. IMPLEMENTATION

The "rid-redundant" procedure, rid-redundant([Q,M,X],M0), is appended to the original C-Prolog program as a built-in predicate. It is written in C-Prolog itself. Before it is used, we need consult it. The basic algorithm is as follows.

Algorithm RID:

(1) Read the values of Q, M, M0. Suppose that the value of Q is q.
(2) If M ≤ M0, then call q(M,X) and exit.
(3) K := M0+1.
(4) If K ≤ M, then
  (a) Call q(K,Xk).
  (b) Insert q(K,Xk) to the knowledge base as the current first clause for subgoal q.
  (c) K := K+1.
  (d) Go to Step (4).
(5) Call q(M,X).

(6) K := M.
(7) If K > M0, then
  (a) Erase q(K,Xk) from the knowledge base.
  (b) K := K-1.
  (c) Go to (7).
(8) Exit.

The C-Prolog program for the "rid-redundant" procedure is as follows.

```
rid-redundant([Q,M,X],M0) :- M=<M0, F=..[Q,M,X], call(F).
rid-redundant([Q,M,X],M0) :- M>M0, K is M0+1, add([Q,K,Xk],M),
      F=..[Q,M,X], call(F), delete([Q,M,Xm],M0).
add([Q,K,Xk],M) :- K=<M, Y=..[Q,K,Xk], call(Y), asserta(Y), K1 is K+1, !, add([Q,K1,Xk1],M).
add([Q,K,Xk],M) :- K>M.
delete([Q,K,Xk],M0) :- K>M0, Z=..[Q,K,Xk], retract(Z), K0 is K-1, !, delete([Q,K0,Xk0],M0).
delete([Q,K,Xk],M0) :- K=<M0.
```

4. EXPERIMENTAL RESULTS

We have implemented and tested the above "rid-redundant" procedure in C-Prolog running on a VAX 11/780 computer. In our experiments, the following statements were added to print the CPU time used in the computation of p(N,A).

```
?- T1 is cputime, p(N,A), T2 is cputime.
```

where T1 is the CPU time used in seconds before running p(N,A) since C-Prolog was started, and T2 is the CPU time used after running p(N,A). The difference between T2 and T1 is the CPU time of running p(N,A).

In evaluating p(17,A) in Example 1, 6.1 seconds were needed, while 0.3333 seconds were needed to run rid-redundant([p,17,A],2). The speed is, therefore, increased by 18.5 times. On the other hand, running p(18,A) results in "out of local stack," but rid-redundant([p,184,A],2) can be evaluated successfully. Hence, the solvable problem size is increased by 184/17=10.8 times.

The following examples were also tried.

Example 2.

```
q(1,1).
q(2,1).
q(3,1).
q(N,B) :- N1 is N-1, N2 is N-2, N3 is N-3, q(N1,B1), q(N2,B2), q(N3,B3), B is B1+B2+B3.
```

The CPU time of running q(15,B) is 6.2666 seconds, but the CPU time of running rid-redundant([q,15,B],3) is only 0.3 seconds. Hence the running speed is increased by 20.9 times. On the other hand, running q(16,B) results in out of local stack, but rid-redundant([q,146,B],3) can be successfully run. So the solvable problem size is increased by 146/15=9.7 times.

Example 3.

```
s(1,1).
s(2,10).
s(N,D) :- N1 is N-1, N2 is N-2, s(N1,D1), s(N2,D2), D is (D1+D2)/2.
```

The CPU time of running s(17,C) is 6.8666 seconds, while the CPU time of running rid-redundant([s,17,C],2) is only 0.35 seconds. Hence, the running speed is increased by 19.6 times. On the other hand, running s(18,C) results in out of local stack, but rid-redundant([s,1000,C],2) can be run, so the solvable problem size is increased by 1000/17=58.8 times.

Example 4.

```
r(N,A) :- p(N,B), q(N,C), A is B+C.
r1(N,A) :- rid-redundant([p,N,B],2), rid-redundant([q,N,C],3), A is B+C.
```

where p, q are the same as in Examples 1 and 2, respectively. In our experiment, the CPU time of running r(14,D) is 4.88 seconds, but the CPU time of running r1(14,D) is only 0.533 seconds. The running speed is increased by 9.15 times. On the other hand, running r(15,D) results in out of local stack, but r1(146,D) can run well, so the solvable problem size is increased by 146/14=10.4 times.

5. DISCUSSION

Sometimes, the calling of non-multi-recursive procedures can also result in repeated computations. An example is shown below.

Example 5.

```
f(c).
f(a).
g(b).
s(a,b).
p(X,Y) :- q(X,Y), t(X,Y).
q(X,Y) :- f(X), g(Y).
t(X,Y) :- f(X), s(X,Y).
```

In invoking p(X,Y), the current goal will become

```
? f(X), g(Y), f(X), s(X,Y).
```

which results in f(X) evaluated twice.

Sometime, redundant computations produced by non-multi-recursive procedures in C-Prolog may be useful, and they should not be eliminated. The following example illustrates this point.

Example 6.

```
g(a).
t(a).
t(b).
s(c).
p(X,Y) :- q(X), r(Y), g(X).
q(X) :- g(X), t(X).
r(Y) :- asserta(g(b)), retract(g(a)), s(Y).
```

If we query the initial goal

```
!?- p(X,Y).
```

then at some moment in running this program, the current goal will become

```
? g(X),t(X),r(Y),g(X).
```

At this moment, the Prolog interpreter will have to solve g(X) twice, but we can not rid any one of them.

Suppose that G1 and G2 are two subgoals of goal G; suppose further that G1 and G2 are the same predicate, if the built-in predicate "asserts" ("asserta," "assertz"), or "retract" appears between G1 and G2 in G, or another subgoal Gi between G1 and G2 calls "assert" ("asserta," "assertz") or "retract" directly or indirectly, then, in general, we can not eliminate either G1 or G2, although they are repeated.

6. CONCLUSIONS

The above experimental results show that the "rid-redundant" procedure can increase the running speed by more than ten times, and can increase the solvable problem size by about ten times for multi-recursive procedures in C-Prolog we have tested. For non-multi-recursive procedure in C-Prolog, the "rid-redundant" procedure does not have any side effects.

REFERENCES

[1] C. J. Hogger, *Introduction to Logic Programming*, Academic Press, London, 1984.

[2] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, New York, NY, 1981.

[3] Fernando Pereira, *C-Prolog User's Manual*, Version 1.3, EdCAAD, 1983.

[4] H. Gallaire, "Logic Programming: Further Developments", *IEEE Symp. on Logic Programming*, 1985.

[5] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1983.

[6] R. E. Davis, "Logic Programming and Prolog: A Tutorial," *IEEE Software*, vol. 2, no. 5, Sept. 1985.

[7] M. R. Genesereth and M. L. Ginsberg, "Logic Programming," *CACM*, vol. 28, no. 9, Sept. 1985.

[8] R. Kowalski, "Predicate Logic as Programming Language," *Proc. IFIP*, 1974.

[9] R. Kowalski, "Algorithm=Logic+Control," *CACM*, vol. 22, no. 7, July 1979.

[10] T. Murata and D. Zhang, "A High-Level Petri Net Model for Parallel Interpretation of Logic Programs," *Proc. Int'l Conf. on Computer Languages*, 1986.

[11] M. H. van Emden, "An Interpreting Algorithm for Prolog Programs," *Proc. First Int'l Conf. on Logic Programming*, 1982.

[12] J. F. Naughton, "Redundancy in Function-Free Recursive Rules," *IEEE Symp. on Logic Programming*, 1986.

[13] J. F. Naughton, *Optimizing Function-Free Recursive Inference Rules*, Tech. Rep. STAN-CS-86-1114, Department of Computer Science, Stanford University, 1986.

N94-70422

EFFICIENT REORDERING OF PROLOG PROGRAMS

Markian M. Gooley and Benjamin W. Wah

Computer Systems Group
Coordinated Science Laboratory
University of Illinois
1101 W. Springfield Ave.
Urbana, IL 61801

(217) 244-7175

Abstract- Redundant search often makes Prolog programs inefficient. Prolog execution corresponds to a depth-first traversal of an AND/OR graph; often a large subgraph will be searched successfully, with failure resulting from a subsequent small one. In this paper we show how reordering of Prolog clauses and especially goals can prevent unnecessary search. We characterize the restrictions on reordering and show how they may be detected. We design a new system of calling modes for Prolog, geared to reordering, and a system for inferring them automatically. We present an improved method for determining a good goal order for Prolog clauses, and use it as the basis for a reordering system, showing how it can be guided by information about modes and restrictions to generate reordered Prolog that behaves correctly.

Index Terms- Logic programming, Prolog, program transformation, goal order, clause order, mode inference, data dependencies, Markov chains, query optimization.

This research was supported in part by the National Aeronautics and Space Administration under grant NCC 2-481.

1. Introduction

1.1. Motivation for our research

Prolog programs tend to be inefficient. In executing a program, Prolog performs a depth-first traversal of an AND/OR graph [10]. Prolog will often traverse several large subgraphs, each corresponding to a goal, only to fail on a later goal. On failure, Prolog backtracks, seeking another way to satisfy a conjunction of goals, and the subgraphs must be searched again. Experienced Prolog programmers, when they can, arrange Prolog clauses so that inexpensive goals that are likely to fail (*e.g.*, tests) are near the beginning of a clause [15]: if a test fails, the rest of a clause need not be evaluated. Similarly, they put clauses last whose heads are unlikely to match a query to their predicate.

Even for small programs, programmers are not always able to determine the best goal or clause order. If some clauses of a predicate or goals of a clause *seem* to have similar costs and likelihoods of success, a programmer will write them in the order he finds most comprehensible, which may may be a very poor order indeed. As programs grow larger and more complex, it will often be difficult to choose between orders: readability will win if it does not appear to slow a program down.

1.2. Defining the problem

We want to reorder Prolog clauses and goals to minimize, or at least reduce, the expected cost of evaluating a query. (We measures this as number of predicate calls; CPU time is too imprecise, but number of unifications would do.) The process should require little intervention from the user. The reordered program it generates need not be formally equivalent to the original, but must still be correct in some sense: for instance, if a query has multiple answers, it might be acceptable for the altered predicates to infer them in a different order, but not to generate a different set of answers.

1.3. Assumptions and definitions

We assume that the reader has such knowledge of Prolog as could be acquired by reading the first few chapters of Clocksin and Mellish [3] or Sterling and Shapiro [15]. All examples will be in the

de facto standard DEC-10 Prolog [14] notation.

Our analysis and reordering programs are written in Prolog itself. Intermediate results from analysis programs are expressed as Prolog facts, as are any user directives to our system.

A predicate with name *name* and number of arguments *arity* will often be referred to as *name/arity*. A variable not bound to an atom or functor is *uninstantiated* or *free;* otherwise it is *instantiated* or *bound*. A tuple of instantiations corresponding to the arguments of a goal is called a *mode;* we will say that a goal calls a predicate in a particular mode and returns from it in (usually) another. If a predicate will not function with certain arguments uninstantiated or in certain modes (*i.e.*, it produces a run-time error or goes into an infinite recursion), we say that it *demands* a particular instantiation or mode.

A predicate that performs an action that is unrelated to the logic of a program and that cannot be undone by backtracking has a *side-effect;* sometimes the predicate is called *extra-logical*. Built-in predicates that do I/O or modify the program have side-effects. A program is *pure* if it uses no built-in predicates, approximating theoretical logic programming, though under Prolog's execution mechanism [15]. Our techniques are not limited to pure Prolog, though we do assume that the program we are restructuring is free of errors. In general, we perform no transformation that might generate an incorrect program, and our system will ask the user to provide information it cannot derive.

We will treat certain built-in predicates cursorily; *viz.*, those that take conjunctions of goals as arguments: *not/1* and the set-predicates *bagof/3* and *setof/3*. In general, their arguments, if compound, may be reordered *in situ*. We assume that no variable goals, whether written explicitly or using *call/1*, are present. We do not consider asserted and retracted clauses. All these problems are beyond the scope of this paper.

1.4. An example of improvement

An example should help clarify what we want to do. Suppose we have a family tree expressed in Prolog. Three predicates determine relationships: *wife/2*, *mother/2*, and *female/1*. The fact *wife(john,jane).* means that John and Jane are married, *mother(john,joan).* means that Joan is John's

mother, and *female(jan).* means that Jan is female. *female/1* is used only if a female is not a wife or a mother: if we assume no illegitimate births, *female(Woman).- wife(_,Woman).* will account for married women (the underscore is the *anonymous variable*). Thus someone is female if she is stated to be female, or if she is a wife.

We have a large collection of the above facts, and the following predicates:

grandmother(GC,GM):- grandparent(GC,GM), female(GM).

grandparent(GC,GP):- parent(P,GP), parent(GC,P).

parent(C,P):- mother(C,P).
parent(C,P):- mother(C,M), wife(P,M).

The query *:- grandmother(X,Y).* will find all the grandmother-grandchild pairs; however, in each case it will first find a grandparent-grandchild pair, suitably instantiating the variables GC and GM, and about half the time promptly reject it. Changing the first clause to

grandmother(GC,GM):- female(GM), grandparent(GC,GM).

will probably reduce the cost: *female/1* will be solved in at most two calls, whereas *grandparent/2* can take far more. Note also that *female/1* instantiates GM and thus the mode of *grandparent/2:* the goal is less costly because we need consider fewer possibilities. Unless only a tiny fraction of the females in the database are grandmothers, the reordering makes sense.

1.5. Previous work

Warren [18] presented a method for reordering goals of conjunctive queries. The queries were translations by a natural language parser of questions in English: a user typed in a question about geography, and the parser generated a query whose goals followed the order of the words in the question. Unfortunately, this order was often inefficient for querying the database. Warren assigned each mode of each predicate a number: the factor by which the goal multiplies the number of alternatives the system must consider. (A goal *country(C)*, *C* uninstantiated, multiplies the possibilities by the number of countries in the database – about 150. With *C* instantiated, the goal serves as a test: it reduces the number of possibilities and therefore receives a value less than 1.) Warren used a simple

function to estimate this number: divide the number of tuples of (answers to) a predicate by the product of the sizes of the domains of each instantiated position in the calling mode. (If *borders/2* -- one country borders another -- has 900 tuples, with each argument having a domain of 150 countries, the function gives 900 for an uninstantiated call, 6 for a partly-instantiated call, and 0.04 for a fully-instantiated call.) In some cases this yielded a speedup of several hundred times over the original version.

Warren reordered only top-level conjunctive queries. We have experimented with reordering all the predicates of simple, nearly pure Prolog programs (*i.e.*, a few built-in predicates, but none with side-effects) such as family trees. We tailor a version of the predicate to each mode, renaming both the new version and the goals that call it. This replaces a predicate with a set of new predicates, one for each mode: 2^{arity} if all of its modes are used. (The arity is usually two for a family tree.) Speed-ups are typically 5 to 40, and rarely under 1 (slowdowns): somewhat better than Warren's, considering that our database of facts is about an order of magnitude smaller than his.

Both Warren's method and our extension have drawbacks. Finding a good heuristic measure to guide reordering is not easy: it should be inexpensive to evaluate, yet take both probability and cost of goals into account. Though it seems to be effective, Warren's function takes account only of how many solutions the predicate has, not how much it costs to find them. Further, domain size for an argument is problematic even for database programs [18]. Worse, evaluating the function is expensive for the extended method: we have to call each predicate, forcing repeated backtracking, and count the number of solution-tuples. Such a method is hardly practical even for "toy" problems like family trees, and is useless for predicates that require unpredictable instantiated arguments or handle data structures.

A promising method has been presented by Li and Wah [10]. Prolog clauses are modelled as Markov chains, so that the success probability and expected cost of a clause can be computed if they are known for each of its goals. The goals can be reordered to minimize cost. Cost and probability data are generated as a by-product of reordering, so that we need only provide them for ground clauses and let the information propagate upwards. Because this method forms the basis of ours, we will describe it in detail later.

1.6. Approach and goals of this paper

We describe the problem of inefficient search more precisely, and consider the two types of reordering: clause and goal. We show how certain "impure" features of Prolog restrict movement of both goals and clauses, and how to collect and use information on restrictions. We look at modes and the restrictions they cause, comparing the "traditional" mode system for Prolog compilers with a new system of *legal modes* suited to reordering. We review ways to infer modes automatically, and adapt one method to finding legal modes. We extend the idea from the earlier paper [10] of modeling Prolog clauses or predicates as Markov chains. We add impure features and mode restrictions to the model, and look at the problems and details of implementing a real reordering system. We present current results, summarize, and suggest both extensions to our methods and ideas for future work.

2. Reordering of clauses and goals

We want to reduce the expected cost of evaluating a Prolog query by reordering the elements of the predicates it calls. Clauses of a predicate and goals of a clause may be reordered, though with certain restrictions; the two types of reordering are distinct, so that they do not compete but rather have a synergistic effect.

2.1. Reordering of clauses:

In standard Prolog, when a predicate is called its clauses will be tried in the order they are written. If the calling goal fails to unify with the head of a clause, the system tries the next clause. If a head unifies but the clause fails despite backtracking amongst its goals, the system, again, tries the next clause. Clause order thereby provides one of the few control mechanisms the programmer has. In fact, clause order determines the order in which solutions to a predicate will be found [15], so that reordering clauses amounts to reordering solutions. If the order of solutions to a predicate is at all important, its clauses cannot be reordered, and the techniques we discuss cannot be used. Therefore, in the rest of this paper we will keep clause reordering in mind, but concentrate on goal reordering.

We want to put a clause first if it is inexpensive and likely to succeed. What we hope to do to the search tree is illustrated in Figure 2.1; the triangles labeled with probabilities represent subtrees,

with size proportional to cost. Before restructuring, there is no particular order to the subtrees that corresponde to the clauses: we search some large trees likely to fail (*i.e.*, try costly clauses) before searching smaller ones likely to succeed (inexpensive clauses). Note that the search tree does not shrink; if we are looking for all solutions to the predicate, we must search the entire tree anyway and have gained nothing.

Another use of clause reordering is in avoiding futile attempts at unification. For instance, many list-processing predicates handle a list recursively, repeatedly decapitating it; a clause with the empty list in its head ends the recursion. For example,

```
length([_|List],Count,Length):- Count1 is Count+1, length(List,Count1,Length).
length([],Length,Length).
```

is in the proper order: though the second clause is far less costly to evaluate than the first, it will match only the empty list. Were it first, the system would attempt unification on every call. *Clause indexing* [17] can have the same effect: at run-time, the system checks the type of one or more arguments of a call (usually just the first) and tries only the clauses whose corresponding arguments might unify with the call's. However, unless the indexing system always indexes on the germane arguments, reordering will still help to prevent useless unifications.

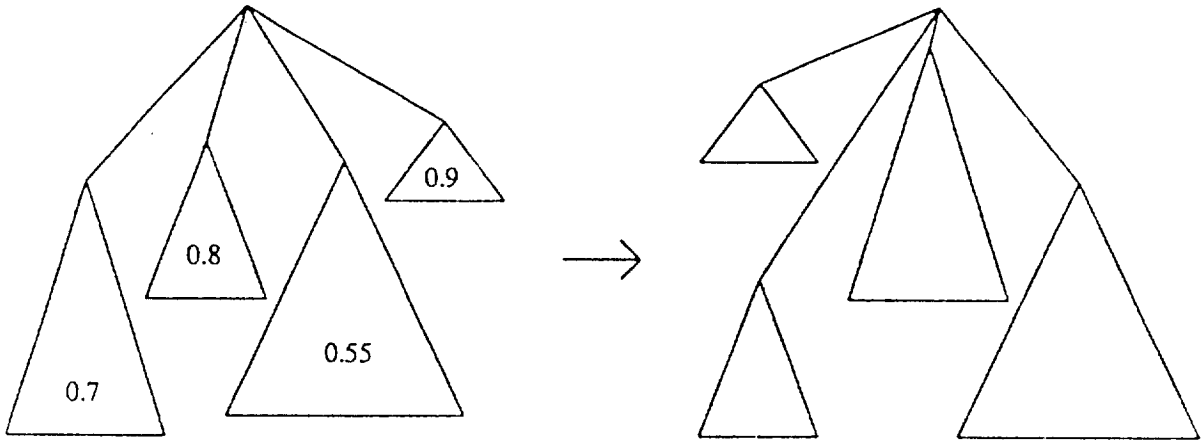


Figure 2.1: Reordering a predicate.

2.2. Reordering of goals:

Standard Prolog evaluates the goals of a clause in order from left to right. On failure, it successively backtracks, first to alternative clauses of the predicate called by the failing goal, and then to those of earlier goals. (*Intelligent backtracking* [1] allows Prolog to backtrack to the first predecessor goal that might provide fresh instantiations, avoiding more-recent goals that cannot. But it is only a palliative for our problem: it does not prevent the initial evaluation of expensive goals. Further, a complete implementation requires NP-complete computations [19].) Thus if a clause fails, it is best to have the culprit goal occur as soon in the clause as possible: not only will later goals never be tried, but less backtracking will be attempted. Figure 2.2 represents a clause as an AND/OR tree. The original version is inefficient: note that if the final goal always fails for a particular call, several large subtrees will have been searched (and probably backtracked through) to no avail. The transformed version puts the inexpensive goal first: if it fails, the larger subtrees will never be tried; if it succeeds, we have narrowed the possibilities for the later, costlier goals.

As with the extended Warren's method we should provide a different version of each predicate for each of its modes. In the worst case, this will again replace a predicate with a collection of 2^{arity} predicates of the same size. In practice, as we will show, this tends not to occur: predicates of high arity can rarely function in more than two or three modes. Furthermore, high arity is considered poor programming style [4], and such predicates can generally be replaced by several new predicates of

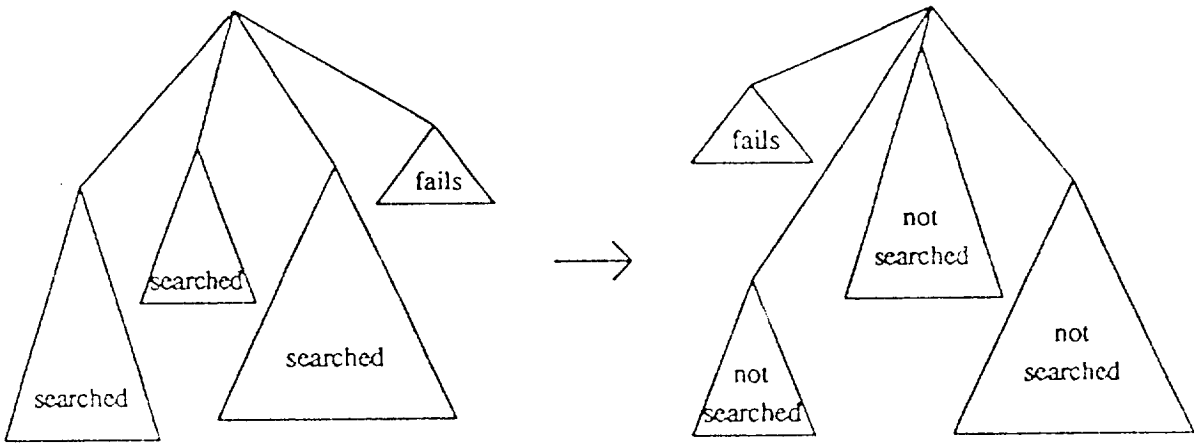


Figure 2.2: Reordering a clause.

lower arity. We trade increased program size for decreased execution time.

2.3. Necessities

In order to go beyond "toy" problems and reorder real Prolog code, we must be able to find restrictions on reordering and apply them to a reordering method. Restrictions are due to recursive and built-in predicates, and fall into two classes: restrictions on movement and restrictions due to modes.

Section III is about restrictions on movement. We show how even a single side-effect can make many reorderings impossible, and describe the exceptions. Control predicates and recursions turn out to have milder effects. Section IV is about modes: specifically, the restrictions they cause and a special type of mode intended for reordering: the *legal mode*. We also describe a method for inferring legal modes. Section V presents the basic reordering method, first for a subset of pure Prolog, then for real programs: the information gained by the methods of Sections III and IV prevents incorrect reorderings. The techniques described in these sections work together; Figure 2.3 shows them as modules between which various pieces of information flow: program code, programmer declarations, and information inferred automatically about program behavior. Note that the system needs to inform the programmer when it cannot infer certain properties of the program, or when declarations are inconsistent.

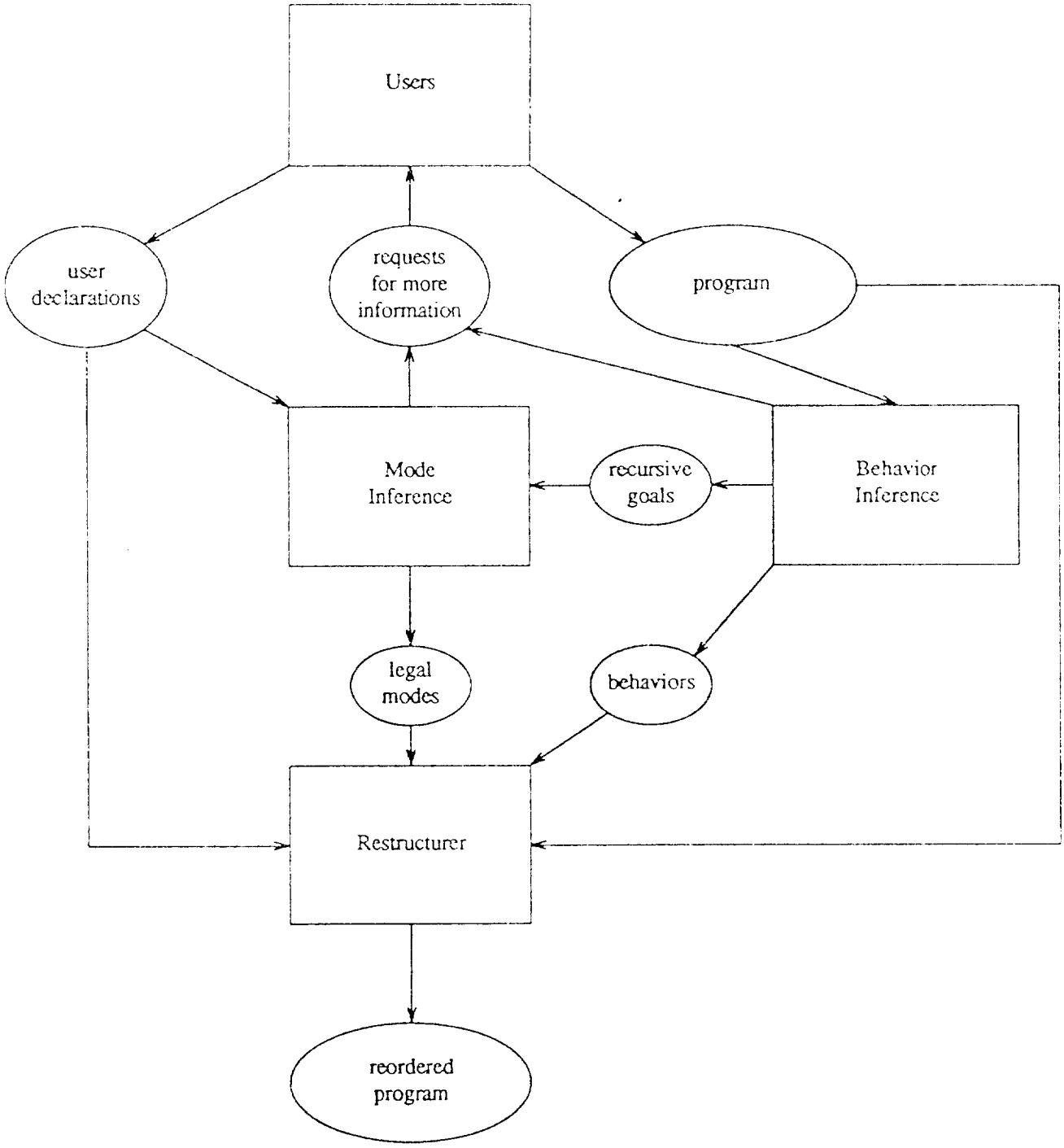


Figure 2.3: The reordering system

3. Restrictions on movement

3.1. Introduction

We cannot reorder carelessly. In all but the simplest programs written in pure Prolog, unconstrained reordering is risky: the reordered predicates may give a set of results different from that of the original, fail when the original succeeds (or vice versa), or go into an infinite recursion. Certain built-in predicates have side-effects that cannot be undone by backtracking; others are designed to provide control. Neither can be treated lightly.

3.2. Predicates with side-effects

Many predicates built into standard dialects of Prolog have *side-effects:* in particular, those that modify the program or do I/O. I/O predicates include *read/1*, *write/1*, *get/1*, *put/1*, and most of the file-handling predicates; predicates that modify a program include *asserta/1*, *retract/2*, and *abolish/2*. It happens that goals calling these predicates are *immobile* within a clause; they therefore act as barriers to permutation of other goals. If we have, say, the clause *a(X,Y):- b(X), c(Y), nl, d(X,Z), e(Z).*, we may exchange *b/1* and *c/1*, or *d/2* and e/1 (perhaps both), other circumstances permitting, but we can do no more. Why not?

Imagine three goals *a*, *b*, and *c* within the body of a clause. *b* has a side-effect. The variables of the goals are instantiated enough that any permutation will work. Suppose we exchange *a* and *b*. Now during an execution of the clause, *a* fails. But *b* has already executed. Its side-effect cannot be undone by backtracking, and in the original program the failure of *a* would have stopped *b* from being reached. Similarly, if we exchange *b* and *c*, failure of *c* will prevent *b* from executing and having its side-effect, which it would have done in the original program. Unless we know that *a* or *c* is certain to succeed in its mode for this situation, we cannot move *b*. We may justly call *b* and predicates like it *fixed* predicates, and other predicates *mobile*.

Clauses of a predicate are also fixed by fixed goals, though for a different reason. Suppose that a_1, a_2, and a_3 are clauses of the predicate *a*, but only a_2 contains a goal that writes something. For now, assume that the three clauses have identical heads. Suppose we exchange a_1 and a_2. The

predicate is called in such a way that any of the three clauses, were it selected, would succeed. Thus a_2 succeeds and writes. Had a_1 been first, it would have succeeded without writing. Now a goal calling a might be followed -- perhaps not immediately or even in the same clause -- by a goal that writes. If so, swapping a_1 and a_2 will change the output of the program: a will now write before the other goal, when in the original it would not have. Thus we cannot move a clause containing a fixed goal.

But the restriction is actually far worse than these examples show. Any predicate that has a fixed predicate as a descendant is itself fixed. If *write/1* within a clause of the predicate *w* prints something, a predicate *x* that calls *w* prints as well. A predicate *y* that calls *w* must also print, and so forth. Thus predicates are responsible for the actions of their descendants: in this way, a single fixed goal can contaminate most of a program.

Before attempting any reordering, we must find out which predicates are fixed. We only need start at a predicate which is not called by any other predicates of the program (an *entry* or *top-level predicate*) and scan its descendants top-down, mimicking program execution. We maintain a list of ancestor predicates so that when we reach a goal that calls a fixed predicate, we can declare its ancestors fixed as well. The information that a predicate is fixed can be asserted, or, better yet, stored in a partly-instantiated binary tree (dictionary) [15]; at the end of the process, the collected information is written to a file as facts.

3.3. Exceptions to fixing goals

Are fixed goals really immobile? There are a few unimportant exceptions: they depend on the neighboring goals and on the nature of the predicate the fixed goal calls. If both the fixed goal and a neighbor will always succeed in their respective calling modes, they can be exchanged -- but nothing will be gained thereby, since both are bound to succeed.

If the fixed goal calls a built-in predicate that has a side-effect, nothing can be done; however, if it calls a user-written predicate there may be hope. Certain predicates have clauses that will not succeed in particular modes. If the call is to such a predicate in one of these modes, the goal that makes the predicate fixed is in one of these clauses, and if we can be certain that the clause will fail

before the culprit goal is reached, then the predicate is not actually fixed for that mode. The situation seems too rare and too complicated to be worth considering.

3.4. Exceptions to fixing clauses

If clauses are *mutually exclusive* in a particular mode (*i.e.*, a call that unifies with the head of one will not unify with the head of the other), the clauses may be swapped in the version of the predicate for that mode. However, the clause order still holds over clauses that are not mutually exclusive; we are only filling in for the deficiencies of the clause-indexing system.

3.5. Control predicates

Prolog includes control predicates which allow programmers to alter its default behavior. These predicates can force or prevent backtracking, provide an *if-then-else* construct, or simply provide shorthand notation.

3.5.1. The cut

The *cut*, written *!*, is notorious for drastically altering program behavior. When a cut is encountered as a goal during forward execution, it succeeds immediately. When it is reached during backtracking, it acts as a barrier: backtracking must stop at the cut, and no other clauses of the predicate may be tried. Backtracking to a cut generally means that the goal has failed.

A cut-bearing clause is usually fixed within its predicate, especially if the cut is *red*, i.e., it changes the meeaning of the program. Consider this example:

```
greater(X,Y,X):- X > Y, !.
greater(X,Y,Y).
```

If we call *greater/3* with its first two arguments instantiated to numbers, it returns the greater number in its third argument. If $X>Y$, it sets the third argument to X: the second clause will not be tried, due to the cut. Otherwise, the second clause sets the third argument to Y. Such programming is common, but it is cryptic, as well as incorrect for such queries as *?- greater(5,0,0)*. If we remove the cut or swap clauses, *greater/3* becomes nonsense.

Sometimes a predicate will consist of several clauses, each with a red cut, followed by a final clause. The first clause that matches the calling goal and then executes at least up to its cut will be the last clause (if not the only clause) tried. Usually clauses in such predicates are arranged in order of their heads' increasing generality: if two heads are the same except that in one position one has a constant and the other has a variable, the one with the variable will come later because it matches a superset of the calls the othe matches. We can exchange any two of the clauses having cuts if their heads do not both unify with some call of the mode in question. If both heads did unify, we would execute the wrong clause, possibly the more-general one when we should have used the other for a special case.

In the body of a clause a cut acts almost like a fixed goal: it is not mobile, but it does not fix the ancestor predicates of the clause. Though cuts will alter our reordering heuristic for goals, they do not restrict goal reordering as greatly as do fixed goals.

3.5.2. Disjunction

A predicate is a disjunction of its clauses. Prolog, however, allows explicit disjunctions within clauses. If two clauses have identical heads, we can write them as one disjunctive clause. For example,

citizen(X):- native_born(X).
citizen(X):- naturalized(X).

can become

citizen(X):- native_born(X) ; naturalized(X).,

where the semicolon represents disjunction. This is not always mere shorthand: if unifying a head with a clause head is particularly expensive, we can use the explicit disjunction to avoid doing it twice. Also, if two clauses share several initial goals, it pays to make the clauses into a single disjunctive clause. That way, there is no need to evaluate the initial goals twice, whether failure occurs in an initial goal or in the first half of the disjunction.

In the process of reordering, we may find it useful to alter a disjunction. Disjunction provides a "semi-permeable" barrier to reordering: we cannot simply move goals between the halves of the dis-

junction, or between either side of a disjunction and the outside. We can move a goal from the outside of a disjunction to its inside by putting a separate copy of it in each half, if it is not fixed or a cut. Once the copy is inside, it can be permuted with the other goals of its half, with the usual restrictions. Taking this process as far as possible creates the equivalent of two separate clauses. Similarly, if duplicate mobile goals in the halves of the disjunction can both be moved to the front or back of their halves, they can be replaced with a single goal outside the disjunction.

What of immobile goals? Consider a disjunctive predicate of the form *a:- b, (c ; d), e, f*. Moving *b* into the disjunction gives *a:- (b, c ; b, d), e, f*. Suppose that *b* is fixed; we execute this clause and *b* succeeds but *c* fails. *b* will be executed twice. If *b* is a cut and *c* fails, the entire predicate *a* will fail. Thus we cannot move immobile predicates into the front of a disjunction; what about the rear? Treating *e* analogously yields *a:- b, (c, e ; d, e), f*. In this case, an immobile *e* won't matter (unfortunately, this transformation isn't useful: we gain nothing by moving a fixed goal into the disjunction).

3.5.3. Implications

Prolog's analog to the *if-then* statements of algorithmic languages is the implication. If a set of goals *p* can be satisfied, then try to satisfy a set of goals *q*: written *p -> q*. This can be made the first half of a disjunction, forming an *if-then-else*. Implication behaves as if defined by the clause

P -> Q :- P, !, Q.

The *then* and *else* parts of an *if-then-else* can be treated much like any other disjunction: a mobile goal common to both can be moved before the premise or after the disjunction. (They are also mutually exclusive -- only one will be executed when the *if-then-else* is invoked -- which will affect our reordering heuristic.) The goals of the premise must stay in the premise, however.

3.5.4. Failure-driven loops

In pure Prolog, a failing clause can accomplish nothing, because any instantiations it makes to variables are undone on failure. In real implementations, goals can call predicates with side-effects, which failure cannot undo. For instance, backtracking can allow a failing clause to execute costly

queries repeatedly, printing the results. Such code is often written for Prolog systems that do not have garbage collection, because heap space is reclaimed on failure.

Several kinds of failure-driven loops are common in Prolog programs. We can force failure by using the built-in predicate *fail* ; here is an example of a common paradigm:

```
show_all:- table(X,Y,Z), write((X,Y,Z)), nl, fail.
show_all.
```

This prints out all the tuples of *table/3* by finding one, printing it and a newline, and then failing, which forces backtracking to find another; when all are found, the second clause prevents the predicate from failing. Another common construct uses the built-in predicate *repeat*. This behaves as if defined by the clauses

```
repeat:- repeat.
repeat.
```

Here is a typical use:

```
main:-
    repeat,
    read(Input),
    process(Input,Output),
    write(Output),
    nl,
    Input == end_of_file.
```

The predicate reads clauses from a file, creates an output for each, prints it, and then fails unless the end of the input file has been reached. Failure forces backtracking to the *repeat*, the next clause is read, and so forth. In general, goals inside a failure-driven loop must remain there; they may be reordered subject to the limitations of fixed goals. For predicates like the first example, the final clause must remain last.

3.5.5. Negation

The built-in predicate *not/1*, also written \+, takes a conjunction of goals as its argument. It behaves as if defined by

```
not(X):- X, !, fail.
```

not(X).

When it is called as a goal, it succeeds only if its argument fails. *not* never instantiates the variables in its argument, but its argument may demand that certain variables be instantiated. (This will require special treatment during mode inference.) In our present work, we treat a negation as immobile (no propagation to ancestors), but allow reordering of multiple goals within its argument.

3.5.6. Set predicates The built-in predicates *bagof/3* and *setof/3* generate lists of terms that satify certain properties specified by a conjunction of goals. We allow reordering of the internal goals, as with *not*, but treat calls to these predicates as immobile, pending further study.

3.5.7. Recursion

Reordering the goals of a recursive predicate is often unsafe. Consider this example [15] written in pure Prolog:

```
select(X,[X|Xs],Xs).
select(X,[Y|Xs],[Y|Ys]):- select(X,Xs,Ys).

permutation(Xs,[X|Ys]):- select(X,Xs,Zs), permutation(Zs,Ys).
permutation([],[]).
```

permutation/1, given a list as its first argument, produces (on backtracking), all of its permutations. Given a variable instead, it will go into an infinite loop. If we swap the two goals of the first clause of *permutation/1*, we will get an infinite loop if the *second* argument is a variable. Thus, reversing two goals changes a safe mode for an unsafe one.

Better implementations of Prolog include some form of *tail-recursion optimization* [17]: stack space is reclaimed on each tail-recursive call, saving memory and speeding executing somewhat. It is tempting to reorder a recursion so that the recursive call is final; Debray [7] has a method that works for some cases.

Reordering recursive clauses is beyond the scope of this paper; for now, we assume that the programmer declares a predicate recursive and provides necessary information to the reorderer. We can easily detect recursion, using a method similar to the one for finding fixed predicates: traverse the program top-down, keeping a list of predicates being scanned, and check if each new goal is a member of

the list.

4. Legal modes

4.1. Introduction

Some Prolog compilers, notably the classic one for the DEC-10 [14], allow users to specify the modes in which a predicate will be called. The compiler then needs to generate object code only for those modes, code that will run faster and require less dynamic storage.

Prolog predicates can often function in more than one mode. When we reorder goals, we usually change their calling modes; therefore, the more modes we can show to be *legal*, the more reorderings are possible. The usual mode declarations do not tell us what modes are legal, merely which will be used in the original program.

What makes a mode legal? A legal mode for a predicate satisfies the demands of the goals in its clauses for particular instantiations, without which they would generate run-time errors or go into infinite recursions. We assume that the programmer knows what he is doing, and does not deliberately call any predicate in an illegal mode. We could have the programmer declare certain modes legal, but this will be time-consuming and error-prone: a last resort.

4.2. Restrictions due to modes

Fixed predicates sharply reduce the number of ways that the goals of a clause or the clauses of a predicate may be permuted. Modes place further restrictions upon goal order, but may serve to allow more clause orders.

4.2.1. Built-in predicates

Many built-in predicates refuse to function in some modes. They *demand* that certain of their arguments be instantiated, or demand certain modes. For example, *functor/3* is a built-in predicate. If its first argument is a term, it can instantiate its other arguments to the term's name and arity; similarly, given a name and number as its last two arguments, it can construct a new term as its first. Give *functor* only a name or arity, and it will not work: in C-Prolog [13], for example, it will fail without a

name and give a run-time error without an arity.

4.2.2. Recursive predicates

Recursive predicates are used chiefly for building or modifying recursive structures such as lists or trees. Given the wrong modes, they enter unproductive infinite recursions, or produce infinite sets of solutions on backtracking. Consider the ubiquitous *delete/3*:

```
delete(X,[X|Y],Y).
delete(U,[X|Y],[X|V]):- delete(U,Y,V).
```

With an atom as its first argument and a list of atoms as its second, *delete* can set its third to the list with one instance of the atom deleted (C-Prolog interpreter):

```
?- delete(b,[c,b,a,b,c],X).

X = [c,a,b,c] ;

X = [c,b,a,c] ;

no
```

(The semicolon, used interactively, forces backtracking.) It functions in several other modes, as well. However, if only its first argument is instantiated, it produces an infinite set of solutions:

```
?- delete(b,X,Y).

X = [b|_1]
Y = _1 ;

X = [_7,b|_9]
Y = [_7|_9] ;

X = [_7,_14,b|_16]
Y = [_7,_14|_16] ;

and so forth...
```

The atom *b* appears in successive positions of the list X *ad infinitum*. (Interestingly, calling *delete* in this way with its clauses reversed puts it into an infinite recursion that returns no results.)

If a new structure is being built based on an existing one, the new one is unlikely to have infinite size. We can require that at least one of the structures that is modified on each recursive call be instan-

tiated. For instance, for *delete*, this would be either the second or third argument; it happens that this is correct. But consider this predicate:

```
perm(X,[U|V]):- delete(U,X,Z), perm(Z,V).
perm([],[]).
```

The first argument needs to be instantiated but the second does not. The call to *delete* will go into an infinite recursion if *X* is not instantiated. The above "heuristic", applied blindly, tells us that only the second argument needs to be instantiated. Obviously the rule does not hold in general.

Unfortunately, we cannot determine with certainty what modes will yield infinite recursions. A solution sufficient for most cases will probably involve showing that for a particular mode, a structure remains essentially unchanged (though growing larger), and nothing in the clauses of the predicate can alter it [9]. We propose that a goal calling any recursive predicate be considered fixed, but without immobility propagating to its ancestors -- like a cut. The programmer must declare legal modes (and, we will see, estimates of execution cost) for a recursive predicate; these may allow a goal to be moved.

4.3. Mode systems

4.3.1. Traditional

The DEC-10 Prolog compiler [14] and many recent commercial compilers allow the programmer to specify the modes in which a predicate will be called. A compiler directive of the form

:- mode <predicate>(<modes>).

declares *<predicate>* to have the modes represented by the tuple *<modes>*. *<modes>* has one element, or *mode item*, for each argument of the predicate; an element is "+" if the argument will always be instantiated, "-" if it will be uninstantiated, or "?" if it might be either (a declaration whose elements are all "?" is therefore superfluous [14]). The compiler need generate code only for the mode or modes declared. Compilation is faster, and the compiled code is more efficient and compact.

4.3.2. Our system: legal modes

4.3.2.1. Traditional versus legal modes

The DEC-10 mode system does not suffice for reordering. We are concerned not with the modes that arise in the original program, but with the modes in which a predicate can be called and still deliver a useful result. DEC-10 mode declarations need only describe a superset of the modes that are actually called; ours must describe a (preferably improper) subset of the legal modes. An illegal mode will make a program incorrect. For example, the *delete/3* predicate avoids infinite recursion if its second or third argument is instantiated. This might be expressed as

:- mode delete(?,?,+), delete(?,+,?).

in a mock DEC-10 notation. A traditional mode declaration might be anything from

:- mode delete(-,-,+).

to the superfluous

:- mode delete(?,?,?).

4.3.2.2. Description

Our system must be suitable both for declaring modes explicitly (for built-in and recursive predicates) and inferring them automatically. Automated mode-inference is desirable because legal modes require verbose but accurate declarations. We do not want to burden the programmer with these.

To avoid confusion with DEC-10 modes, we change notation. "-" becomes *u* (*u*ninstantiated), "+" becomes *i* (*i*nstantiated), and "?" becomes *a* (*a*ny mode). In our full system, *i* will be reserved for terms with no uninstantiated elements, and *a* will be both a "don't care" (like "?") and a "don't know." We introduce other mode items in later sections.

Each predicate has a set of modes in which it can be called. These are its legal *input modes*. Each input mode has a corresponding *output mode*, which is at least as instantiated as its input mode. (Output modes are not necessary in the traditional system, but they are necessary, at least for internal use, in mode inference.) For instance, *delete/3* has (in our notation) the input/output pairs

[(a,i,a),(i,i,i)], *[(u,u,i),(u,i,i)]*, and *[(i,a,i),(i,i,i)]*.

A three-symbol system is not expressive enough. Two problems arise: *coupled variables* and *partially-instantiated structures*.

Coupled (or *aliased*) variables have different names but have been unified together, losing their distinctness; instantiating one instantiates the other. Suppose that a goal couples two variables, but the coupling is overlooked. If one variable is instantiated at the time, the other will wrongly be thought uninstantiated; if neither is, any later instantiation of either will not be passed to the other.

During reordering, we update the instantiations of variables so that we can reorder goals safely. Missing couplings will make instantiated variables appear uninstantiated, constraining reordering but not creating illegal calls. We assume that the programmer is always right, and allow the original order of calls even if some appear illegal. Thus the reordering is sound, if only, as a last resort, by not being a reordering at all. In practice, we do try to detect coupling and propagate its effects. We show two variables coupled by marking their positions in the output mode with the same symbol: c_i, say, for the *i*th coupled pair.

A list or other recursive structure may contain uninstantiated variables. If we treat it as fully-instantiated and then feed it to a predicate that demands an instantiated argument, the predicate might, after a few recursive calls, expose a variable and go into an infinite recursion.

Consider this common predicate:

```
append([X/Y],Z,[X/W]):- append(Y,Z,W).
append([],X,X).
```

If we call it in *(i,u,u)*, it appends the variable of the second argument to the list in the first, yielding a list whose tail is the variable: a form of a useful structure called a *difference list* [15].

Now suppose we have this clause:

```
build(L1,L2,L3,L4):-
    transform(L2,L2a),
    transform(L3,L3a),
    append(L1,L2a,L2b),
    append(L2b,L3a,L4).
```

We call *build/4* in mode (i,i,i,u); *transform/2* works in modes *(i,a)*, and *append/3* in modes *(i,a,a)* and *(a,a,i)*. Now we try to reorder the clause. If we say that *append(i,u,u)* returns *(i,u,u)* (the third argument has a variable in its tail), the following good order will be rejected:

```
build(L1,L2,L3,L4):-
    append(L1,L2a,L2b),
    transform(L2,L2a),
    append(L2b,L3a,L4),
    transform(L3,L3a).
```

On the other hand, if we say that *append(i,u,u)* returns *(i,u,i)*, the following illegal order will be generated:

```
build(L1,L2,L3,L4):-
    append(L1,L2a,L2b),
    append(L2b,L3a,L4),
    transform(L2,L2a),
    transform(L3,L3a).
```

Clearly we must forego the first rather than risk the second, but can we do better?

Making the mode system more complex can make it more expressive. We give it the flavor of a typing system [12], adding notations for structures and lists that are incomplete (or not known to be complete). Our solution is tentative at best; we hope to improve it.

A list known to be fully instantiated has mode *i*; otherwise, it has a mode item of the form

l(<element 1>,<element 2>,<element>,...,<element n>,<tail>).*

There is one mode item for each known element of the list and one for the tail. If the length of the list is known exactly, each element of the body gets a single item. If not, any known leading or trailing elements each get an item, and the other elements of the body are lumped together and given a single mode, followed by an asterisk indicating zero or more items. The tail is *i* if the list has determinate length, *u* otherwise; recall that because the list-constructor function "." has two arguments, the list *[a,b,c,d]* is actually *.(a,.(b,.(c,.(d,[]))))*. The empty list is the last element. Thus we might describe one mode of *append/3* as *[(l(a*,i),u,u),(l(a*,i),u,l(a*,u))]*: a list of determinate length but unknown types of elements generates a similar list of indeterminate length.

Functors other than lists, unless known to be fully instantiated, are treated similarly, with mode items of the form *f(<arg 1>,<arg 2>,...)*. If the arity of the functor is known, each *<arg i>* will be a mode item; otherwise we treat them as with lists. Thus we might describe one mode of the built-in predicate =.. (or *univ*) [3], which converts between terms and lists, the functor corresponding to the head of the list and the arguments to the tail, as *[(f(a*)),(l(i,a*,i))]*. A term of unknown arity and contents becomes a list of unknown size and contents with a constant head and a determinate length.

We have the following mode items:

u: uninstantiated argument
a: argument of uncertain instantiation
i: instantiated argument
c_i: member of the *i*th coupled set
l: partially instantiated list
f: partially instantiated term (not a list).

To these we add *p*, found only in output modes: it *preserves* the mode of the argument; in other words, the argument is unaltered by the predicate. This is essentially shorthand. Instantiated arguments are not altered by predicates; their presence in output modes is as placeholders.

4.3.2.3. Uses

These modes are intended primarily for reordering-systems, but other applications are possible. They might be useful for running Prolog on multiprocessors: for instance, goals that appear to depend on predecessors might actually be able to run in parallel with them. Other researchers have recognized the usefulness of more-specific mode declarations for parallel processing [2].

4.4. Mode inference

4.4.1. Previous work

Not much has been published on automatic mode-inference. Mellish [11] had success with a mode system intermediate in complexity between the DEC-10 system and ours. Tung and Moldovan [16] presented a method for finding modes as part of their work on AND-parallelism. Debray [5] developed a useful method, with accompanying theory; because it is the basis of our method, we will

present it in some detail.

4.4.2. Debray's method

Debray uses three modes: c (closed: contains no variables; corresponds to "+" and our i), f (free: an uninstantiated variable; corresponds to "-" and u), and d (don't know: not definitely closed or free; corresponds to "?" and a). He defines two orders over these modes: the *instantiation order* $\leq_{ins}$, for which $f \leq_{ins} d \leq_{ins} c$ (c is at least as instantiated as d, which is at least as instantiated as f), and the *information order* $\leq_{inf}$, for which $d \leq_{inf} f$ and $d \leq_{inf} c$ (d provides no more information than f or c).

The idea behind Debray's method is simple. We have a program and a query to one of its predicates in a given mode. We combine the query's mode with the head of a clause; initially all of the variables in the clause are f, but some become c or d. We now know a calling mode for the predicate of the first goal. We check a table whether we already know the corresponding output mode; If so, we update the clause variables and go on to the next goal; if not, we "call" the predicate of the goal to infer the output mode, just as we did with the query, performing the process recursively and storing results as they are derived. (Eventually, facts and built-in predicates provide modes at the bottom levels.) The output mode for a given input mode is taken as the greatest lower bound, with respect to $\leq_{inf}$ ($\cap_{inf}$), of the output modes arising from the different clauses; in other words, an argument position is d unless all clauses yield c or f for it. When the process is finished, we take $\cap_{inf}$ of the calling modes as the mode for the predicate.

Debray speeds up the process by transforming the program into a *mode interpreter*, a program that mimics the behavior of the real program in producing output modes from input modes and propagating the results. Each original predicate maps to a *table manager;* a call to the predicate becomes a call to its manager. On being called with a mode as one argument, the manager checks whether the table contains an output mode for that calling mode. If so, it returns the mode. If not, it calls new clauses based on the clauses of the original predicate. These clauses call other table managers (also standing in for predicates) and, when necessary, predicates that propagate instantiations of variables. The manager takes $\cap_{inf}$ of the results as the output mode, stores it in its table, and returns it to the

caller. Thus we execute only the parts of the original program that are pertinent to finding its modes, and avoid duplicated effort. Because the mode interpreter executes deterministically, and requires simple unifications, it should execute in far less time than the original program.

4.4.3. Current work

We follow Debray in making a mode interpreter of the program and using a table of results to save effort. Our complicated mode system requires more information, including separate records for input modes with distinct output modes. We find sets of legal modes, not a single DEC-10-style mode, and we consider built-in predicates.

We begin with a program, possibly with explicit mode-declarations and directives not to try inferring modes already declared. All built-in predicates have modes declared. We scan throgh the program, reading the declarations and finding name, arity, and number of clauses for every user predicate. We load all this into our table, implemented as a dictionary.

Now we can generate transformed predicates corresponding to each user predicate, analogous to Debray's table managers. We call them *root predicates*, and they look like this:

```
<predicate>(<arity>,Dictionary):-
    lookup(<predicate>,<arity>,ModePairs,Dictionary),
    nonvar(ModePairs),
    !.


<predicate>(<arity>,ModePairs,Dictionary):-
    <predicate>(<arity>,0,ModePairs0,Exceptions0,Dictionary),
    <predicate>(<arity>,1,ModePairs1,Exceptions1,Dictionary),
    ...,
    <predicate>(<arity>,N,ModePairsN,ExceptionsN,Dictionary),
    combine([ModePairs0,ModePairs1,...,ModePairsN],
      [Exceptions0,Exceptions1,...,ExceptionsN],ModePairs),
    lookup(<predicate>,<arity>,ModePairs,Dictionary).
```

(Replace words in angle-brackets with appropriate constants.) *lookup/4* gets the set of input/output mode pairs that has been inferred for the predicate; if there is no set, it calls the transformed clauses of the predicate, each providing a set of mode pairs as from an original clause. *combine/3* combines the results into a single set of pairs, stored in the dictionary (so that the predicate needs only two

arguments); the lists *Exceptions*, tell if cuts or other built-in predicates (such as *var/1*) will allow or forbid certain input modes. If all of the exception lists are empty, we disallow the input modes that do not appear in the sets of all clauses. If an input mode generates more than one output mode, we take the $\bigcap_{ins}$ of the outputs. The $\leq_{ins}$ for our mode system is

$$u \leq_{ins} a \leq_{ins} f \leq_{ins} i,$$

with l treated as f and p replaced by the corresponding input mode.

For the rest of the transformation, we scan through the program again, making of each clause two predicates resembling these:

```
<predicate>(<arity>,<clause number>,ModePairs,Exceptions,Dictionary):-
    <goal 0>(<arity 0>,Dictionary),
    <goal 1>(<arity 1>,Dictionary),
    ...,
    <goal M>(<arity M>,Dictionary),
    Exceptions = [Exception0,Exception1, ... ],
    setof( (InMode,OutMode),
      InMode^OutMode^<predicate>(<arity>,<clause number>,InMode,OutMode),ModePairs).


<predicate>(<arity>,<clause number>,InMode,OutMode):-
    lookup_member(<goal 0>,<arity 0>,InMode0,OutMode0,Dictionary),
    update(<state 0>,<head 0>,InMode0,OutMode0,<state 1>,<head 1>),
    lookup_member(<goal 1>,<arity 1>,InMode1,OutMode1,Dictionary),
    update(<state 1>,<head 1>,InMode1,OutMode1,<state 2>,<head 2>),
    ...,
    lookup_member(<goal M>,<arity M>,InModeM,OutModeM,Dictionary),
    update(<state M>,<head M>,InModeM,OutModeM,<final state>,InMode),
    extract(<final state>,OutMode).
```

First we call the root predicates of the goals, to ensure that the necessary sets of mode pairs are in the dictionary. (We also keep the inefficient *setof/3* predicate out of deep recursions.) If the same predicate is called by several goals of the clause, only one root-predicate call need be retained. We also build a list of *exceptions* contained in the clause (cuts and so forth) that might affect combiuation of clause results. *setof/3* then calls the second predicate repeatedly to find the set of mode pairs for this clause.

The second predicate computes a single input mode / output mode pair for the original clause. During execution, it maintains two tuples of variables. One represents the demands made on the head of the original clause, and the other represents the state of all the original variables. *lookup_member* returns one of the legal mode pairs for a goal. *update* looks at the demands of the input mode. It checks whether the clause variables can satisfy them; if they cannot, it tries to satisfy the demands by passing them on to the head tuple. If the head tuple can satisfy them, *update* updates the head tuple with the demands, and the clause variables with the output mode; if it cannot, *update* fails, and we backtrack to *lookup_member* for another mode pair. When the second predicate succeeds, each of its goals has been assigned a legal mode, the demands made by the goals are satisfied by the head (creating a legal input mode), and the instantiations made by the goals have been recorded (and transformed by *extract* into an output mode): it has found a mode pair that is legal for the original predicate.

5. Markov-chain method for reordering

We want to find the expected cost and success probability for a predicate called in a legal mode, given that we know the cost and probability for the predicates called by its clauses. We use an idea introduced in the paper by Li and Wah [10]: modeling execution of a Prolog clause as an absorbing Markov chain [8]. The cost and probability of the clause follow directly from properties of its chain; combining results for clauses gives us values for the predicate.

5.1. Pure Prolog

First we describe the method assuming a subset of pure Prolog; *viz.*, no recursions, and only variable arguments to a clause unless the clause is a fact. We extend it to full pure Prolog, leaving real-life Prolog for a later subsection.

5.1.1. Overview

Consider the clause

k:- a, b, c, d.

(We omit variables for brevity.) Suppose that we know the success probability p_i and the expected

cost c_i for each predicate i. To find the expected probability and cost of finding a single solution to the clause, we model it as the Markov chain of Figure 5.1. Each goal receives a state; the arcs are labeled with success and failure probabilities. The two additional absorbing states S and F represent success and failure. (In order for the clause to be selected, the head must already have unified, so that a separate state k, unification probability p_k, and arcs to b and F are unnecessary.)

The state transitions of the chain mimic Prolog execution. We start in the state labeled with the first goal b, proceeding to state c if b succeeds, or ending up in state F if it does not. In every goal state i we either move forward with probability p_i, or backtrack with probability $1-p_i$. Eventually the process stops in an absorbing state: success or failure.

Note that if we want the cost of finding all solutions to a conjunction of goals, as in a *bagof*, a *setof*, or a failure-driven loop, we add an arc of probability 1 from S back to the last goal, as in Figure 5.2. Whenever we find a solution, then, we backtrack, looking for another, until we fail.

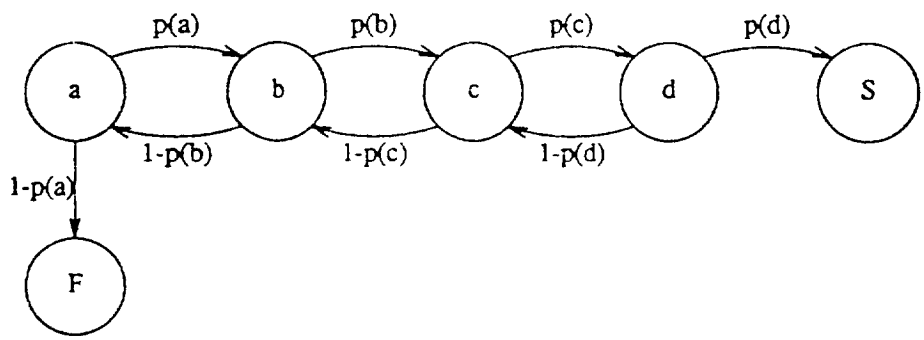


Figure 5.1: A clause body as a Markov chain.

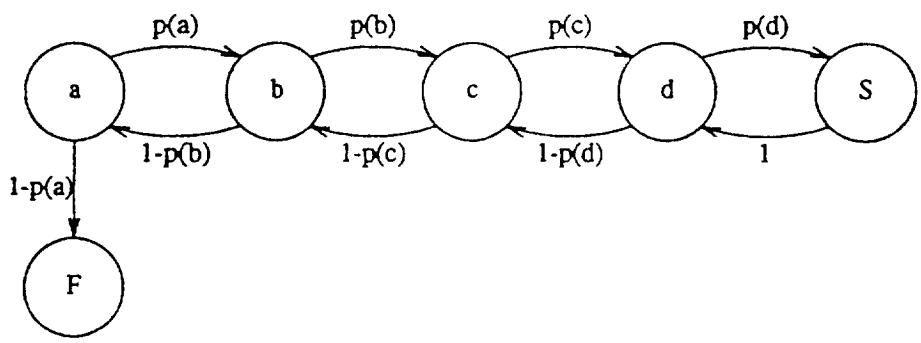


Figure 5.2: Markov chain for all solutions to a clause.

5.1.2. Deriving probability and cost for a clause

We take the success probability p_{clause} of the clause as the probability that the Markov process will be absorbed in state S. We take $\sum_{i \in goals} c_i v_i$ as the expected cost of a solution, where c_i is the expected cost of goal i and v_i is the mean number of visits by the process to state i. Therefore we need to calculate p_{clause} and the v_i: textbook [8] mathematics.

We begin with the transition matrix P. An element p_{ij} contains the probability of a transition from state i to state j; S gets index 1 and F gets index 2. The chain has r states, s of which are transient. We consider four submatrices of the $r \times r$ matrix P:

$$P = \begin{bmatrix} I_{r-s} & 0 \\ R & Q \end{bmatrix}$$

I_{r-s}, the identity matrix, gives transitions between absorbing states; R, $s \times (r-s)$, gives transitions into absorbing states; Q, $s \times s$, gives transitions between transient states; and the zero matrix, $(r-s) \times s$, shows that nothing leaves an absorbing state. Here is P for our example:

$$P_k = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1-p_a & 0 & p_a & 0 & 0 \\ 0 & 0 & 1-p_b & 0 & p_b & 0 \\ 0 & 0 & 0 & 1-p_c & 0 & p_c \\ p_d & 0 & 0 & 0 & 1-p_d & 0 \end{bmatrix}$$

The matrix $N = (I - Q)^{-1}$ provides our answers. The first row of N contains the numbers of visits v_i for the transient states, because we start in the first state. The product NR gives the success probability p_{clause}: it is the first element of the column of state S (column 1), so that we need only take the dot product of that column of R and the first row of N. For our example,

$$Q_k = \begin{bmatrix} 0 & p_a & 0 & 0 \\ 1-p_b & 0 & p_b & 0 \\ 0 & 1-p_c & 0 & p_c \\ 0 & 0 & 1-p_d & 0 \end{bmatrix},$$

and

$$N_k = det(I-Q)\begin{bmatrix} p_c p_d+(1-p_b)(1-p_c) & p_a[1-p_c(1-p_d)] & p_a p_b & p_a p_b p_c \\ (1-p_b)[p_c(1-p_d)-1] & 1-p_c(1-p_d) & p_b & p_b p_c \\ 1-p_b(1-p_c) & p_c-1 & 1-p_a(1-p_b) & p_c[1-p_a(1-p_b)] \\ (1-p_b)(1-p_c)(1-p_d) & (1-p_c)(1-p_d) & (1-p_d)[p_a(1-p_b)-1] & p_b p_c+(1-p_b)(1-p_a) \end{bmatrix},$$

$$det(I-Q) = \frac{1}{(1-p_b)[(1-p_a)(1-p_c)-p_a p_c p_d]},$$

and

$$p_k = N_k R_k[1,1] = \frac{p_a p_b p_c p_d}{(1-p_b)[(1-p_a)(1-p_c)-p_a p_c p_d]}.$$

In practice we will do everything numerically; if we have N goals, this requires inversion of an $N \times N$ matrix, $2N$ extra multiplications for finding the probability and cost, and sundry additions. If our reorderer is written in Prolog, it is best to call a routine written in an algorithmic language, such as C, to generate the matrix, invert it, and return the results. (C-Prolog [13] does not have a built-in interface to C, but many commercial Prologs do.)

5.1.3. Applying the results

We can use the Markov-chain method to find, for each calling mode, the least expensive reordered version of a predicate. Different modes may require different orders, and the cost of a goal will vary with its mode. For the subset of pure Prolog we have assumed for now, we can reorder the goals as we please, computing the cost for each permutation and choosing the cheapest. An N-goal clause has $N!$ permutations; for $N>4$ we can save effort by culling some of these as useless (*e.g.*, an expensive goal with a high probability of success should not be first). With probability and cost values found for each clause, we combine the probabilities to find the conditional probabilities for the clauses in order, multiply these by the costs, and obtain the expected probability and cost for the entire predicate.

5.1.4. Initial probabilities and costs

We derive the cost and probability of a clause from those of its goals; in our subset of Prolog, this information must be based on costs and probabilities of facts. We allow facts to have non-variable arguments, so that their heads are prone to failure.

Evaluating a fact costs one call; the success probability of a fact is problematical unless the call is uninstantiated, when it is unity. We can, like Warren [18], find *domains* for each argument; a domain might be the set of constants in that position of the predicate, or perhaps the set of all constants in the program. (We have found the latter satisfactory for our family tree programs). The probability is taken as 1, divided by $|domain_i|$ for every position i that contains a constant in both fact and call.

Suppose we allow full pure Prolog, with recursion, and with constants and data structures allowed in any predicate. Now it is hard to determine the probability that a call will unify with a clause head: if any pair of corresponding arguments has both members non-variable, the probability will no longer be 1. For constants, we can use domains, as with facts; for structures, the predicates are usually recursive, with a special case that matches empty structures. The size of a structure will affect both the chance that the empty case will match, and the cost of a call. Because size is usually unknown before run-time, we need an estimate, supplied either by the system or the user. For some predicates, the user may have to provide explicit unification probabilities for each argument of each clause head.

5.2. Restricted (real-world)

5.2.1. Restrictions

The restrictions we discussed in earlier sections limit the number of possible reorderings. Many attractive orders are thus forbidden, but the number of permutations that we have to examine for each clause body is sharply reduced.

Fixed goals cannot be moved, but goals on either side of one are not restricted. For example, if the third goal of a five-goal clause is fixed, the number of permutations plummets from 5!=120 to 2!2!=4. A cut gives rise to a Markov chain like that of Figure 5.3: the goal after the cut does not backtrack to it, but fails; of course, the cut also restricts goal movement like a fixed goal. A disjunction generates a split chain, as in Figure 5.4; reordering is under the restrictions given earlier. An *if-then-else* implication generates a chain that combines features of the two previous chains (Figure 5.5); this emphasizes that the implication is essentially a cut with effects local to its clause.

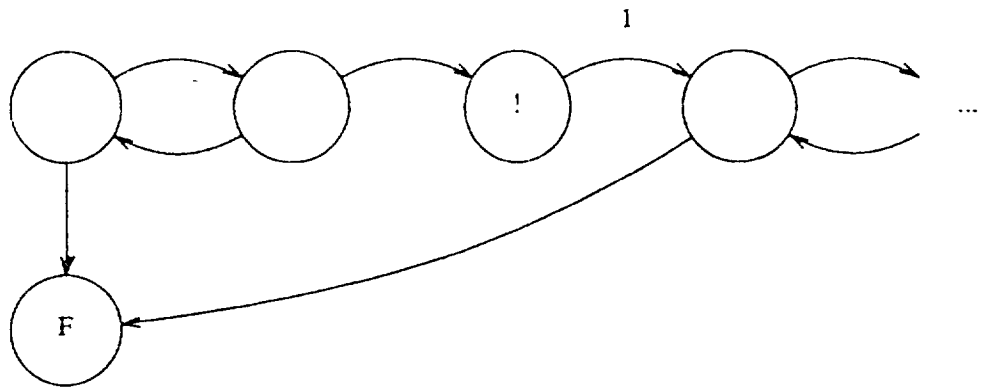


Figure 5.3: A clause with a cut.

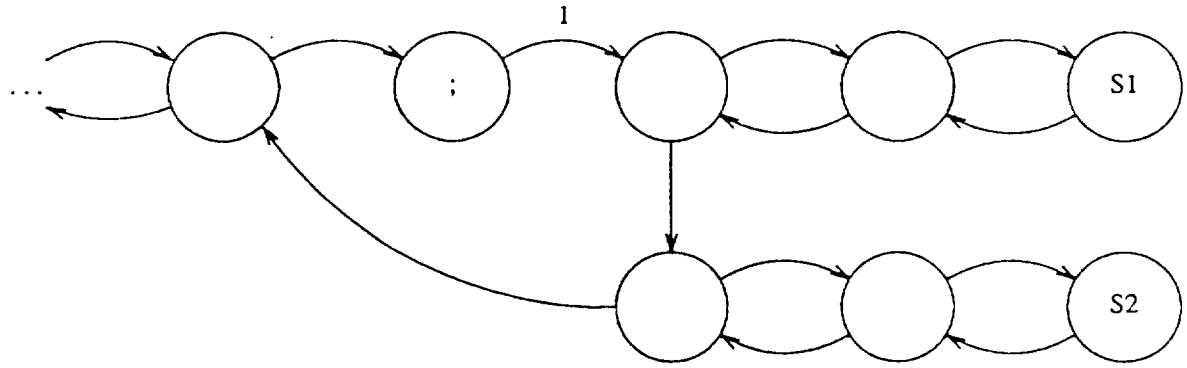


Figure 5.4: A disjunctive clause.

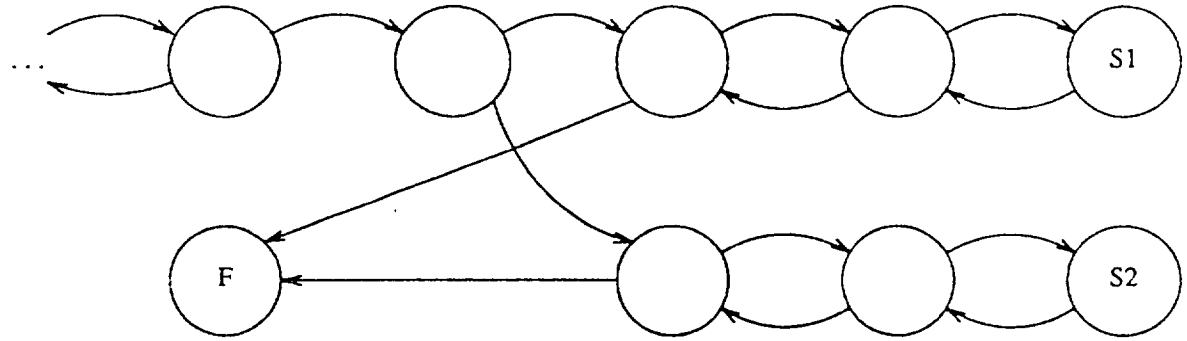


Figure 5.5: An *if-then-else* implication

Every goal must make a legal call to its predicate. A reordering that prevents this, not providing the goal with an instantiation it needs, is rejected. We can generate a potential order for a clause in a particular mode, goal by goal, keeping track of which variables a goal demands and which it instantiates. As soon as an illegal mode arises, we backtrack to generate another order, so that only legal

orders are tested for the lowest cost. The process is reminiscent of our mode-inference technique.

5.2.2. Implementation

5.2.2.1. Necessities

Reordering goals requires a great deal of information about a program, provided by the programmer or, preferably, inferred automatically. This includes:

Legal modes for each built-in or user predicate, declared or inferred
Unification probabilities for certain clause heads in certain modes
Probabilities and costs for built-in predicates
A list of entry points
A list of the fixed predicates
A list of the recursive predicates
Probabilities and costs for recursive predicates.

These are in the source file as declarations, in files generated by inference programs, or in a file (hand-written for a particular Prolog implementation) containing information about the built-in predicates. They will be in the form of facts. In most cases we can read them into the clause space of a Prolog interpreter, or compile them and load their object code; however, probabilities and costs for predicates are perhaps best loaded into a dictionary, where they will be joined by others being inferred for the transformed program. In this way, all such information can be accessed uniformly without asserting new facts, which is clumsy and costly.

5.2.2.2. Execution

The reordering program begins by loading the facts and the program to be transformed. Starting at an entry point, it traverses the program depth-first until it reaches a clause whose goals all have probability and cost values in the dictionary. It checks for control predicates, and looks for further information about the goals, such as legal modes and mobility. Choosing one of the predicate's legal modes, it begins to generate permutations, discarding those that break the rules. When a legal permutation is generated, the reorderer calls a routine written in C, giving it the probabilities and costs in order, and whether or not a special transition matrix must be generated. The routine returns the expected probability and cost for the permutation. The reorderer generates other permutations and

chooses the least costly. It prints the altered clause under a new name and stores the probability and cost in the dictionary. It repeats this for the other legal modes.

The reorderer repeats this for the other clauses of the predicate; when a predicate is complete, it combines the stored probability and cost values, along with any necessary estimates of unification probability, to obtain the probability and cost for the predicate, which it stores. Working upwards, the reorderer repeats the process for every user predicate in the program, changing goal names to correspond to the new predicate names.

5.2.2.3. Example of a transformation

We have several predicates from the family tree program:

grandfather(X,Y):- parent(X,Z), father(Z,Y).

parent(X,Y):- mother(X,Y).
parent(X,Y):- father(X,Y).

father(X,Y):- mother(X,Z), wife(Y,Z).

(*mother/2* and *wife/2* are collections of facts. We want a better-ordered *grandfather/2* for mode *(u,u)*. Suppose that reversing the goal orders of *father* and *grandfather* is helpful in this mode, according to the reorderer. (The clauses of *parent/2* have one goal each: no room for improvement.) The reorderer might generate this:

grandfather_uu(X,Y):- father_uu(Z,Y), parent(X,Z).

father_uu(X,Y):- wife(Y,Z), mother(X,Z).

If we want to call the new *grandfather/2* as a query, we might add

grandfather(X,Y):- var(X), var(Y), !, grandfather_uu(X,Y).

and similar clauses appropriate to the other modes.

6. Performance

Results so far have been promising. We have found that if restrictions on reordering are not too great, fairly large speedups are possible. Also, fixed predicates do not seem to have as great an effect

as we had feared, despite their habit of propagating immobility to their ancestors.

6.1. Reordered programs

Table 6.1 shows some results of reordering a family-tree program written in nearly pure Prolog.

Table 6.1: Results of reordering a family tree program.

Predicate	Mode	Number of calls: original	Markov	optimal
aunt	(u,u)	1452	985	
	(u,i)	61872	1409	
	(i,u)	2035	2035	
	(i,i)	93354	67238	
brother	(u,u)	792	792	792
	(u,i)	4555	539	530
	(i,u)	1004	1004	1004
	(i,i)	16003	21449	16003
cousins	(u,u)	62987	1477	
	(u,i)	106606	2034	
	(i,u)	63570	2203	
	(i,i)	138088	151062	
daughter	(u,u)	382	229	229
	(u,i)	700	494	494
	(i,u)	4251	670	670
	(i,i)	21423	19803	19803
grandmother	(u,u)	230	200	200
	(u,i)	11254	367	367
	(i,u)	548	548	548
	(i,i)	28426	18705	18705
grandson	(u,u)	709	663	663
	(u,i)	1080	699	690
	(i,u)	24930	1034	1034
	(i,i)	44964	42586	36156
mother_in_law	(u,u)	98	98	98
	(u,i)	4338	314	314
	(i,u)	363	363	363
	(i,i)	18648	16055	16055
nephew	(u,u)	10379	1400	
	(u,i)	102122	1523	
	(i,u)	10750	2592	
	(i,i)	122156	81072	
siblings	(u,u)	416	416	416
	(u,i)	4126	575	575
	(i,u)	575	575	575
	(i,i)	12712	18158	12712
sister	(u,u)	723	383	275
	(u,i)	4486	648	540
	(i,u)	935	935	935
	(i,i)	15934	21380	15934

There are 55 constants in the program, representing people; we called each predicate in each mode, with one call for each possible assignment of constants to instantiated positions. Therefore, a test of mode *(u,u)* required one call, modes *(u,i)* and *(i,u)* required 55 apiece, and modes*(i,i)* required 3025. All results were generated by an instrumented version of C-Prolog 1.5 [13] running on a Sun-3/50M workstation. We give figures for the original program, the reordered version, and for the least-expensive reordering as determined by exhaustive enumeration (when combinatorial explosion did not forbid).

Here are the pertinent predicates of the original program. Not shown are the ten facts for *girl/1*, nineteen for *wife/2*, and 34 for *mother/2*.

```
female(X):- girl(X).
female(X):- wife(_,X).

male(X):-not(female(X)).

husband(X,Y):-wife(Y,X).

father(X,Y):-mother(X,Z),wife(Y,Z).

parent(X,Y):-mother(X,Y).
parent(X,Y):-father(X,Y).

married(X,Y):-wife(X,Y).
married(X,Y):-wife(Y,X).

daughter(X,Y):-parent(Y,X),female(Y).

siblings(X,Y):-mother(X,Z),mother(Y,Z),unequal(X,Y).

unequal(X,Y):- X==Y.

sister(X,Y):-siblings(X,Y),female(Y).

brother(X,Y):-siblings(X,Y),male(Y).

grandparent(X,Y):-parent(X,Z),parent(Z,Y).

grandmother(X,Y):-parent(X,Z),mother(Z,Y).

grandson(X,Y):-grandparent(Y,X),male(Y).

aunt(X,Y):-parent(X,Z),sister(Z,Y).
aunt(X,Y):-parent(X,Z),brother(Z,W),wife(W,Y).

nephew(X,Y):-siblings(X,Z),son(Z,Y).
nephew(X,Y):-siblings(X,W),married(W,Z),son(Z,Y).
```

```
cousins(X,Y):-parent(X,Z),parent(Y,W),siblings(W,Z).
cousins(X,Y):-parent(X,Z),parent(Y,W),siblings(W,V),married(V,Z).

mother_in_law(X,Y):-married(X,Z),mother(Z,Y).
```

The gains are most impressive for the half-instantiated modes. Apparently, the partial instantiation is exploited by a good goal order, so that large search trees are pruned early. This does not happen so readily for mode *(u,u)*: the first goal must be called uninstantiated, and if all goals of a clause are expensive, reordering is not as useful. Similarly, for mode *(i,i)*, enough variables are instantiated that goal order is not crucial; sometimes the reordered clauses are more expensive in this mode.

6.2. Effects of impure features

Here is an example of how impure features of Prolog need not constrain reordering. Fixed predicates, as we have seen, have a far-reaching effect on reordering because they propagate their behavior to their ancestors. Analysing some large Prolog programs, We count the number of fixed predicates and compare it to the total number of predicates to see how many will be constrained during reordering due to immobile goals.

Table 6.2 shows results for a checkers-playing program and some knowledge-based systems, designed for faculty registration, written by students as course projects. Program size ranged from about 40K to 350K bytes. Even though these programs are heavily interactive and use *assert* freely, they still have a fair number of mobile predicates.

7. Conclusions

In this paper we have presented a method for reordering Prolog for more efficient execution. We have shown what information is necessary to keep a reordering correct, how much of this can be inferred automatically at present (with descriptions of the automatic inference methods), and how much must still be provided by the programmer. We have introduced a new system of Prolog modes which may also be useful for multiprocessor work. We have extended the Markov-chain reordering method to cover full Prolog, and presented some preliminary results of our work.

Table 6.2: Prevalence of fixed goals in example programs

Name	Total predicates	Fixed predicates
checkers	145	34
uva04	173	80
uva12	75	99
uva16	87	64
uva20	273	148
vu08	133	95
vu14	154	93
vu17	141	87
vu19	139	96
vu20	333	178

Much work remains to be done. We need to see whether the mode system is sufficiently expressive, and improve it if necessary. We must improve our techniques for recursive predicates; the system should be able to derive or estimate modes, probabilities, and costs, relying upon the programmer only as a last resort. If analytical methods do not suffice, it may be necessary to implement an expert system to provide information for certain predicates. We must see whether *not*, *bagof*, and *setof* really need to be immobile; it seems that we only need satisfy any demands made by their enclosed goals, combining modes to form a single mode for the entire goal. We can also exploit for reordering other properties that can be inferred for Prolog predicates, such as functionality [6]. Finally, we should integrate our techniques into one system, so that we can provide a program as input and, with slight intervention by the user, receive a reordered, improved program as output.

Acknowledgements:

Thanks are in order to Barry Fagin of the University of California at Berkeley and Hans Tebra of Vrije University, Amsterdam, for the programs discussed in Section 6.2.

References:

[1] M. Bruynooghe and L. M. Pereira, "Deduction Revision by Intelligent Backtracking," in J. A. Campbell, *ed.*, *Implementations of Prolog*, Ellis Horwood, Chichester, England, 1984.

[2] J.-H. Chang, A. M. Despain, and D. DeGroot, "AND-Parallelism of Logic Programs Based on a Static Data Dependency Analysis," *Digest of Papers of COMPCON Spring '85*, pp. 218-225, February 1985.

[3] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, Berlin, 1984.

[4] J. K. Debenham, Style in Logic Programs, Technical Report 85.7, School of Computing Sciences, New South Wales Institute of Technology, Australia.

[5] S. K. Debray, Automatic Mode Inference for Prolog Programs, Technical Report 85/019, June 1985, Department of Computer Science, State University of New York at Stony Brook, Stony Brook, NY.

[6] S. K. Debray, Detection and Optimization of Functional Computations in Prolog, Technical Report 85/020, August 1985, Department of Computer Science, State University of New York at Stony Brook, Stony Brook, NY.

[7] S. K. Debray, Optimizing Almost-Tail-Recursive Prolog Programs, Technical Report 84/089, November 1984, Department of Computer Science, State University of New York at Stony Brook, Stony Brook, NY.

[8] J. G. Kemeny and J. L. Snell, *Finite Markov Chains*, Van Nostrand, New York, 1965.

[9] R. Kowalski, *Logic for Problem Solving*, North-Holland, New York, 1979.

[10] G.-J. Li and B. W. Wah, "How Good are Parallel and Ordered Depth-First Searches?" *Proceedings of the International Conference on Parallel Processing*, pp. 992-999, IEEE, August 1986.

[11] C. S. Mellish, "Some Global Optimizations for a Prolog Compiler," *Journal of Logic Programming*, vol. 1, pp. 43-66, 1985.

[12] A. Mycroft and R. A. O'Keefe, "A Polymorphic Type System for Prolog," *Artificial Intelligence*, vol. 23, pp. 295-307, 1985.

[13] F. C. N. Pereira, C-Prolog User's Manual, version 1.5.

[14] L. M. Pereira, F. C. N. Pereira, and D. H. D. Warren, User's Guide to DECsystem-10 Prolog, September 1978.

[15] L. Sterling and E. Y. Shapiro, *The Art of Prolog*, MIT Press, Cambridge, MA, 1986.

[16] Y.-W. Tung and D. I. Moldovan, "Detection of AND-Parallelism in Logic Programming," *Proceedings of the International Conference on Parallel Processing*, pp. 984-991, IEEE, August 1986.

[17] D. H. D. Warren, Applied Logic -- Its Use and Implementation as a Programming Tool, Technical Note 290, June 1983, SRI International, Menlo Park, CA.

[18] D. H. D. Warren, "Efficient Processing of Interactive Relational Database Queries Expressed in Logic," *Seventh Conference on Very Large Data Bases*, IEEE, 1981.

[19] D. A. Wolfram, "Intractable Unifiability Problems and Backtracking," *Third International Conference on Logic Programming*, pp. 107-121, July 1986.

N94-70423
3
p. 85

RESOURCE SHARING ON CSMA/CD NETWORKS IN THE PRESENCE OF NOISE

BY

DUANE EDWARD DINSCHEL

B.S., University of Illinois, 1985

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, 1987

Urbana, Illinois

PRECEDING PAGE BLANK NOT FILMED

ABSTRACT

Resource sharing on CSMA/CD networks can be accomplished by using window-control algorithms for bus contention. The window-control algorithms are designed to grant permission to transmit to the station with the minimum contention parameter. Proper operation of the window-control algorithm requires that all stations sense the same state of the network in each contention slot.

Noise causes the state of the network to appear as a collision. False collisions can cause the window-control algorithm to terminate without isolating any stations. This thesis develops a two-phase window-control protocol and approximate recurrence equation with noise as a parameter to improve the performance of the window-control algorithms in the presence of noise. The results are compared through simulation, with the approximate recurrence equation yielding the best overall performance.

Noise is even a bigger problem when it is not detected by all stations. In such cases it is possible for the window boundaries of the contending stations to become out of phase. Consequently, it is possible to isolate a station other than the one with the minimum contention parameter. To guarantee proper isolation of the minimum, a broadcast phase must be added after the termination of the algorithm. This thesis discusses the protocol required to correct the window-control algorithm when noise is not detected by all stations.

ACKNOWLEDGEMENTS

The author would like to thank his advisor, Professor Benjamin W. Wah, for his help and direction in conducting this research.

The author also wishes to thank NASA (grant NAG 1-613) and NSF (grant DMC 85-19649) for their support of this work.

Finally, the author would like to thank his family and friends for their support and encouragement.

TABLE OF CONTENTS

PAGE

1. INTRODUCTION 1

1.1. Resource Sharing Issues 1
1.1.1. Scheduling Strategies 1
1.1.2. Interconnection Networks 2
1.2. Objective of Thesis 5
1.3. Organization 5

2. BACKGROUND 6

2.1. CSMA/CD Networks 6
2.2. Window-Control Algorithm 7
2.2.1. General Operation 7
2.2.2. Binary-Divide Window-Control 8
2.2.3. Dynamic Programming Window-Control 10
2.2.4. Greedy Window-Control and Approximations 13
2.2.5. Load Estimation 14

3. EFFECTS OF NOISE ON CSMA/CD BUS CONTENTION 15

3.1. Types of Noise 15
3.2. Effects of Noise on Ethernet Network 16
3.3. Effects of Noise on Window-Control Algorithms 21
3.3.1. Noise Detected by All Stations 22
3.3.1.1. Original Window-Control Algorithm 22
3.3.1.2. Two-Phase Algorithm 24
3.3.1.3. Heuristic Window-Control with Noise 25
3.3.2. Windows Becoming Out-of-Phase 30
3.4. Dependent Noise 32
3.5. Load Estimation 34
3.6. Summary 34

4. SIMULATION AND ANALYSIS OF WINDOW-CONTROL PROTOCOLS 35

4.1. Independent Noise, Synchronized Windows 35
4.1.1. Binary Exponential Backoff Algorithm 36
4.1.2. Binary-Divide Window-Control Algorithm 37
4.1.3. Original Dynamic Programming Window-Control Algorithm 37
4.1.4. Two-Phase Window-Control Algorithm 40
4.1.5. Approximate Recurrence Relation 40
4.1.6. Comparison of Algorithms 43
4.2. Asynchronized Windows 45
4.3. Load Estimation 47
4.4. Implementation of a Window-Control Algorithm on Ethernet in the Presence of Noise 49

5. CONCLUSIONS 53

PAGE

APPENDIX A. SIMULATION PROGRAMS IN FORTRAN 55

APPENDIX B. SIMULATION PROGRAMS IN C 71

REFERENCES 80

1. INTRODUCTION

1.1. Resource Sharing Issues

Physical limitations in the size and speed of current semiconductor technologies, combined with the availability of inexpensive processors and interconnection networks, have spurred great interest in distributed computing. One problem of distributed computing is to get the maximum throughput in the system. Chu, Holloway et al. show that a *saturation effect* occurs as more processors are added to a system [1]. In other words, the throughput of a system increases as more processors are added, but decreases after a certain point due to interprocessor communication overheads. Experience shows that a system's computing power normally increases by only a factor of 0.8 for each additional processor [2].

Although there are several essential issues involving multiprocessor systems [3], one critical consideration is a good resource sharing strategy. Such a resource sharing strategy should decompose work among the processors, in order that the maximum throughput of the system can be realized. There are two major issues to consider in regard to resource sharing: the job scheduling strategy and the type of interconnection network among the resources.

1.1.1. Scheduling Strategies

There are many different ways to categorize the various scheduling techniques. Probably the major distinctions are between static and dynamic techniques [4, 5]. Static techniques make scheduling decisions based on fixed probabilities and queueing theory models [6]. Consequently, they are also known as *probabilistic* techniques. Static scheduling has the advantages of simplicity and low interprocessor communication overheads, but has a disadvantage in that it cannot account for changes in the workload distributions of the system. Ni and Hwang [6], and Chow and Kohler [4] have discussed various static scheduling strategies.

Dynamic techniques make scheduling decisions based on the current state of the system. Consequently, they are also often called *deterministic* techniques. Since dynamic scheduling techniques can account for changes in the workload distributions, they can potentially perform better than static techniques. However, they also introduce additional communication overheads to obtain the resource status. Chow and Kohler [4], Lo and Liu [7], Wah and Hicks [8], and Juang and Wah [9] have discussed various dynamic scheduling strategies.

Scheduling strategies can also be centralized or distributed. In centralized scheduling, one processor is responsible for scheduling all resources. It must receive status information from each processor, process that information according to a scheduling strategy, and send messages to processors to reschedule the tasks. Consequently, a lot of traffic flows in and out of the central scheduler, causing it to become a bottleneck of the system. Furthermore, the system is not fault-tolerant, because the scheduler is a critical component of the system. These drawbacks tend to favor distributing the scheduling intelligence among all the processors. The problems with distributed scheduling strategies are that they are more difficult to implement and require additional overheads for all processors to maintain the current status on the other processors. Nevertheless, the robustness of distributed scheduling strategies combined with their better performance, makes distributed scheduling the preferred choice in many multiprocessor applications [8, 9, 10, 11].

1.1.2. Interconnection Networks

The characteristics of the interconnection network among the resources are important factors in determining the load sharing strategy. There are basically three types of interconnection networks: shared bus, crossbar switch, and multistage dynamic blocking networks. The following examples assume homogeneous resources and processors.

The most basic and least expensive interconnection network involves a single shared bus that connects p processors to r resources (Figure 1.1). Each of the p processors must compete for using the bus before it can send a job to one of the r resources. There are two common methods of resolving bus contentions: token passing and carrier sense multiple access with collision detection, (CSMA/CD) [12]. This thesis focuses on the design of resource sharing strategies on CSMA/CD networks. Wah and Juang have studied resource scheduling issues on CSMA/CD networks for both single shared busses [13] and multiple shared busses [14].

The crossbar switch (Figure 1.2) is at the other end of the spectrum from the single shared bus. It is an expensive network in that connections must be possible from all p processors to all r resources. Such a network involves $p \cdot r$ switching elements for p processors and r resources. The crossbar switch is non-blocking, that is, connections from P_i to P_j do not inhibit connections from other processors to free resources. Therefore, crossbar networks do not impose any limitations on the resource scheduling algorithms. However, they are often used in small systems

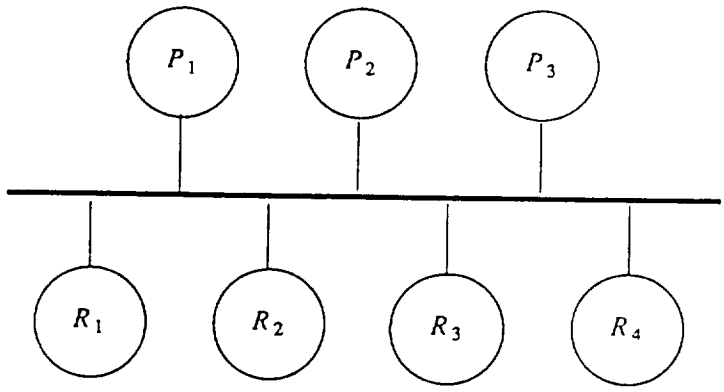


Figure 1.1. Single Shared Bus

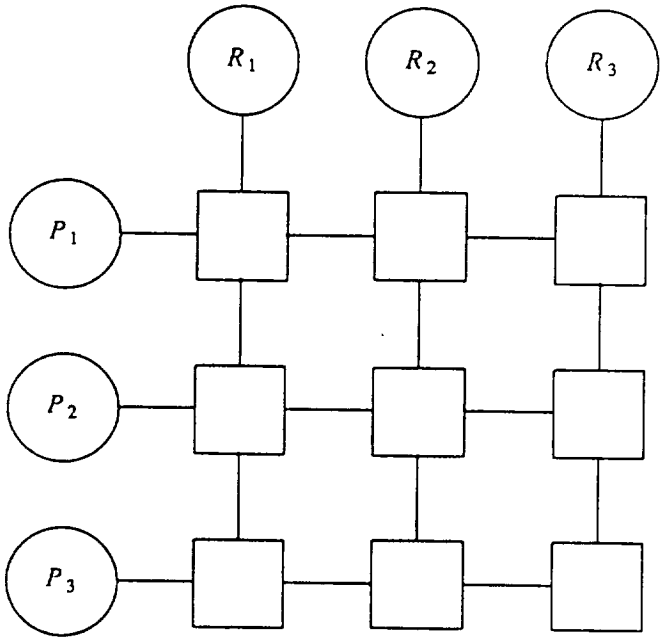


Figure 1.2. Crossbar Switch

due to their high cost of implementation.

Multistage dynamic networks offer a trade-off between cost and flexibility. They are constructed from a series of stages with switch boxes in each stage. The Omega network (Figure 1.3) is an example in this class. Each switch box can perform either a straight or exchange connection. Note that this network is less expensive than a crossbar switch in that there are only $O(n*\log_2 n)$ switch boxes to connect n processors to n resources. However, multistage networks are often blocking. Consider the scenario in Figure 1.3. The bold lines represent connections that have already been established. Suppose R_3 is busy and P_0 has a job ready. P_0 will be blocked from accessing the only free resource, R_1, due to the connection from P_2 to R_0.

Owing to the blocking effects in multistage networks, the design of the resource scheduling strategies is often more difficult. Wah has presented a distributed algorithm for multistage dynamic networks [8]. The algorithm consists of two phases: a resource phase and a request phase. In the resource phase, information concerning the number of free resources is passed from the resource side to the processor side. Then, in the request phase, the network propagates requests from the processors to the resources; each switch box makes an independent routing decision based on the results of the resource phase. If a request is blocked, it must be sent back to the exchange box in the previous stage for rerouting. Conflicts in a box due to two signals arriving simultaneously are resolved by a priority scheme. This algorithm is distributed in that exchange boxes operate independently rather than under a centralized control; it

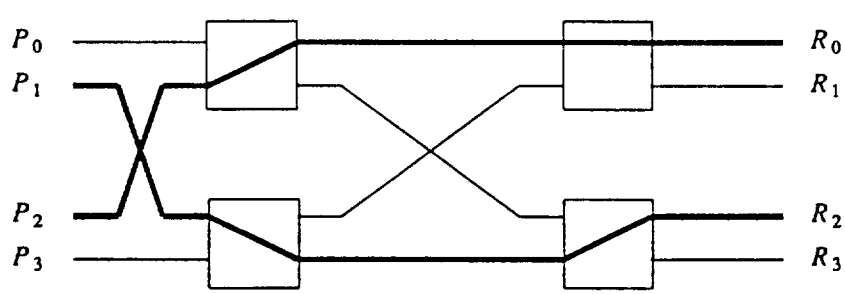


Figure 1.3. A 4-by-4 Omega Network

is deterministic in that requests are routed dynamically.

1.2. Objective of Thesis

This thesis focuses on the study of resource sharing strategies on CSMA/CD networks in the presence of noise. More specifically, it concentrates on window-control methods for bus contention on a particular CSMA/CD network, the Ethernet. The objective of this thesis is to develop a robust window-control algorithm for resource sharing on CSMA/CD networks in the presence of noise.

1.3. Organization

There are four chapters following the introduction. Chapter 2 provides background information on CSMA/CD networks and the various window-control algorithms that can be used to resolve bus contention for load balancing applications.

Chapter 3 deals with the problems of noise in relation to the window-control algorithms. It outlines the various types of noise and discusses how noise can affect the operation of a CSMA/CD network. The effects are divided into two classes, one in which the window boundaries of all stations remain synchronous, and the other in which the window boundaries of the stations may become asynchronized.

Chapter 4 contains the effects of noise on both the exponential backoff method and the window-control methods. It compares the effects of noise with regard to various numbers of stations, various probabilities of noise, and various window-control methods. This chapter deals with simulations for both synchronized and asynchronized window boundaries. Finally, this chapter outlines the implementation of a window-control algorithm in a real system in which noise is a factor.

Chapter 5 contains a summary of results and future research directions.

2. BACKGROUND

This chapter surveys previous work on CSMA/CD networks and window-control algorithms for resolving bus contention. The first section describes the operation of CSMA/CD networks and Ethernet's binary exponential backoff method for resolving bus contentions. The second section discusses the various window-control algorithms and the technique used to estimate the channel load.

2.1. CSMA/CD Networks

Carrier sense multiple access with *collision detection*, CSMA/CD was developed and patented by Xerox as part of its Ethernet local network [15]. The concept of CSMA/CD is fairly simple. Any station wishing to use the bus senses it to determine whether it is busy or idle. If the bus is idle, it may attempt to transmit; otherwise, it must wait until the bus is no longer busy. When two stations both detect that the bus is idle and simultaneously attempt to transmit, their transmitted signals will collide with each other. A collision detection mechanism is necessary to detect this event and abort the transmissions. In the worst case, assuming a signal propagation delay of τ from one end of the network to another, a station at one end of the network may attempt to transmit at time t and another station at the other end of the network at a time slightly less than $t+\tau$. The two transmissions will then collide, and the second station will detect the collision almost immediately. However, the first station will not detect the collision until τ units after the second station has transmitted. The total time from initial transmission to collision detection in the worst case is, therefore, 2τ. This amount of time is called a *contention slot*, and is the amount of time a station must listen to the bus after transmitting to guarantee that no collision will occur.

The fact that there are collisions leads to the issue of determining the time a station should wait before it retransmits. There are several solutions to this problem. The first is to transmit as soon as the station detects that the network is idle. This is known as *persistent* CSMA/CD [12]. However, under heavy traffic, this method works very poorly because stations constantly collide in their attempt to access the bus. Consequently, a *nonpersistent* protocol [12] may be used in which the stations always wait a random amount of time before retransmitting. This method does not work as well under light traffic. One compromise is to use a *p-persistent* protocol [12] in which the stations transmit immediately upon sensing the network idle with probability p and wait a random amount of time

before retransmitting otherwise.

Ethernet uses the *truncated binary exponential backoff algorithm* to determine when a station can retransmit after detecting a collision [16]. According to the data link layer of the Ethernet specifications, a station has 16 attempts to transmit on the bus [16]. If it is unsuccessful, it waits an integral number of contention slots before retransmitting. The average delay before retransmission grows exponentially until n=10, but up to 15 retransmissions may be attempted. The number of slot times to delay before the nth retransmission attempt is chosen as a uniformly distributed random integer, I, in the range $0 \le I \le 2^k$, where $k=\min(n,10)$. If all 16 attempts fail, the station quits transmitting and logs the event as an error.

The binary exponential backoff method has an advantage over the persistent methods for bus contention in that it can more effectively account for the system load [15]. However, its performance degrades as the number of stations increases. Furthermore, it is not possible to establish priority of one station over the others with regard to the use of the bus. Hence, it is not easy to implement a load balancing strategy in which the priority to transmit is based on the system workload. Both of these problems can be resolved by the window-control algorithm to be outlined in the next section.

2.2. Window-Control Algorithm

The purpose of the window-control algorithm is to identify the extremum from a set of physically dispersed random numbers called *contention parameters* [13]. The distribution of these numbers varies according to the load balancing scheme used. For example, the length of the job queue or a random number from a uniform distribution can be used as the contention parameter. Since procedures for isolating the minimum and maximum contention parameter are similar, this thesis discusses only procedures for isolating the minimum. Bus contention is resolved by granting permission to transmit to the station with the minimum contention parameter.

2.2.1. General Operation

B. W. Wah and J. Y. Juang have presented a window-control algorithm for determining priority on CSMA/CD networks [13]. It is a distributed algorithm in which each station maintains a window to decide whether it will retransmit in the future. All stations update their windows synchronously, each contention slot, according to

the state of the network.

The algorithm assumes that all contention parameters lie in the interval (L,U]. The initial window is chosen with a lower bound, a, at L, an upper bound, b, at U, and a temporary upper bound, w, in (a,b]. The value of w is chosen according to the window-control protocol being used, and varies as a function of a and b. In each contention slot, all stations with contention parameters smaller than w transmit. If there is a collision, there must be at least two stations in the window, $(a,w]$; therefore, b is lowered to w in the next contention slot. If the state of the network is idle, the window must be empty; therefore, a is raised to w in the next contention slot. Otherwise, if there is a successful transmission, the algorithm terminates. This process is repeated until the minimum contention parameter has been isolated. The operation of the window-control algorithm for an individual station is outlined in Figure 2.1.

Operation of the window-control algorithm is best illustrated by an example (Figure 2.2) . Initially, each of the five stations generates a random contention parameter in the interval (L,U], and sets its window to $(L,w_1]$. During the first contention slot, stations 1, 2 and 4 transmit. Collision is detected, and the window is reduced to $(L,w_2]$. In the second contention slot no stations transmit. Therefore, the lower bound of the window is increased to w_2 and the upper bound of the window to w_3, resulting in a window $(w_2,w_3]$. Finally, in the third contention slot only station 2 remains-- the transmission is successful and the algorithm terminates.

Wah and Juang discuss the performance of several window-control algorithms in which the channel load and the distribution functions from which the contention parameters are generated are exactly known: binary-divide, dynamic programming, greedy window-control, and approximations to greedy window-control [13, 17]. These are outlined in the next sections and evaluated with respect to the presence of noise in the next chapter. In the following discussion, a is the current lower bound, b is the current upper bound, w is the upper bound of the next window, and n is the number of contending stations at the start of the algorithm.

2.2.2. Binary-Divide Window-Control

The simplest of the window-control methods is known as the binary-divide window-control. The new upper bound of the window, w, is set to the value $(a+b)/2$. If there is a collision, b is set to w; if the line is idle, a is set to w. The algorithm is reiterated until the minimum is isolated. The expected number of contention slots to resolve the collisions is $O(\log_2 n)$ [13].

```
procedure window_protocol_station_i;
/* procedure to find window boundaries for isolating one of the contending stations */
 /* window - function to calculate window size w,
  * random - function to generate local contention parameter,
  * transmit_signal - function to send signal to bus with other stations synchronously,
  * detect - function to determine state of the network,
  * x - local contention parameter,
  * lb - lower bound of interval containing minimum
  * ub - upper bound of interval containing minimum
  * contending - boolean to continue the contention process,
  * state- state of the network: collsion, idle, or success
  */
    [
    lb := L;
    ub := U;
    r := random(L,U);
    w := window(lb,ub);
    contending := true;
    while (contending) do [
            if (r ≥ lb and r ≤ w) then [
                    /* parameter is inside window, contend for bus */
                    transmit_signal();
                    /* test for unique station in the window */
                    state := detect;
                    if (state = collision) then
                            /* update upper bound of interval containing minimum */
                            ub := w;
                    else /* successful isolation of minimum */
                            return(lb,ub);
                    w := window(lb,ub); ]
            else[
                    state := detect();
                    if (state = idle) then
                            /* all parameters are outside window */
                            /* update lower bound of interval containing minimum */
                            lb := w
                    else
                            /* some other parameters are inside window, stop contending */
                            contending := false; ]
            ]
            return(failure);
    ]
```

Figure 2.1. Procedure illustrating the execution of the window-control algorithm by a single station [13]

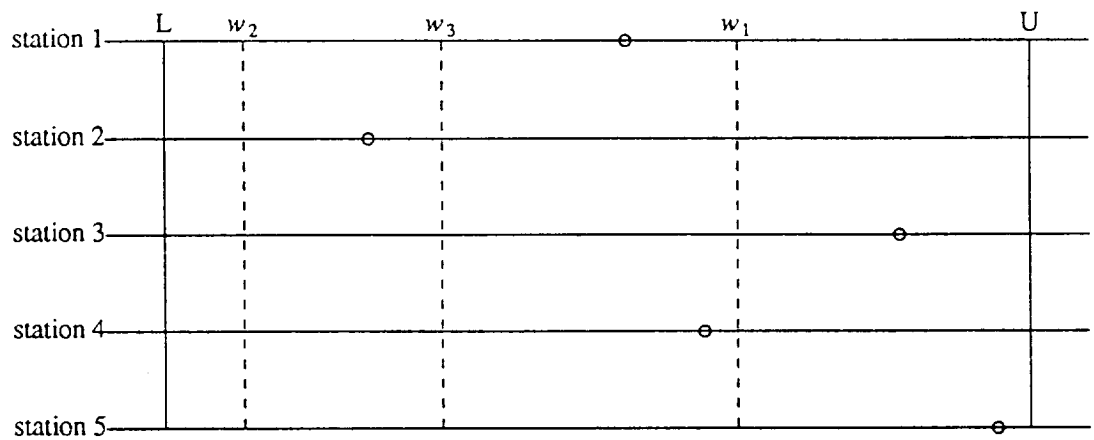


Fig. 2.2. Window-Control Algorithm [13]

2.2.3. Dynamic Programming Window-Control

A significant improvement to the number of contention slots to resolve collisions can be gained by using a dynamic programming window-control algorithm. The algorithm attempts to minimize the expected number of contention slots to resolve collisions, $M(a,b)$, given that there are n contention parameters in $(a,U]$, and that collision occurs in the current window. The following notations are first defined [13]:

$M(a,b)$:
The minimum expected number of iterations to resolve contention given that there are n contention parameters in $(a,U]$ and collision occurs in the current window $(a,b]$.

$g(w,n,a,b)$:
Probability of *success* in the next iteration if a window of $(a,w]$, $a < w < b$, is used.

$l(w,n,a,b)$:
Probability of *collision* in the next iteration if a window of $(a,w]$, $a < w < b$, is used.

$r(w,n,a,b)$:
Probability of *no transmission* in the next iteration if a window of $(a,w]$, $a < w < b$, is used.

It follows directly from the above definitions that

$$l(w,n,a,b) + g(w,n,a,b) + r(w,n,a,b) = 1. \tag{2.1}$$

Functions $l(w,n,a,b)$, $r(w,n,a,b)$, and $g(w,n,a,b)$ are conditional probabilities based on the condition that there was a collision in the last iteration of the protocol. This implies that there must be at least two contention parameters within the boundary $(a,b]$, and that all contention parameters must be greater than a. This condition can

be designated as event A.

Three mutually exclusive events can occur by choosing w, $a<w<b$, which can be identified as follows [13]:

B={exactly one of the x_i's is in $(a,w]$, given that all x_i's are greater than a}

C={no x_i is in $(a,w]$, given that all x_i's are greater than a}

D={more than one x_i is in $(a,w]$, given that all x_i's are greater than a}

From these events, the conditional probabilities can be expressed as [13]:

$$g(w,n,a,b) = Pr(B \mid A) = \frac{Pr(A \cap B)}{Pr(A)} \tag{2.2}$$

$$r(w,n,a,b) = Pr(C \mid A) = \frac{Pr(A \cap C)}{Pr(A)} \tag{2.3}$$

The set $A \cap B$ represents the event that exactly one x_i is in $(a,w]$, one x_i is in $(w,b]$, and all other x_i's are greater then w. The set $A \cap C$ represents the event that at least two x_i's are greater than w.

Let $F_i(x)$ be the distribution function governing the generation of x_i. Wah and Juang presented the formulas for g,r and $Pr(A)$ for the general case of arbitrary independent distribution functions [13]. The functions g and r can be simplified by assuming uniform distribution functions for each station. $F_i(x)$ is then equal to x. These functions can be expressed as:

$$Pr(A) = \frac{(1-a)^n - n(b-a)(1-b)^{n-1} - (1-b)^n}{(1-a)^n} \tag{2.4}$$

$$g(w,n,a,b) = \frac{n(w-a)[(1-w)^{n-1} - (1-b)^{n-1}]}{Pr(A)\cdot(1-a)^n} \tag{2.5}$$

$$r(w,n,a,b) = \frac{(1-w)^n - n(b-w)(1-b)^{n-1} - (1-b)^n}{Pr(A)\cdot(1-a)^n} \tag{2.6}$$

The function, $M(a,b)$, can then be formulated recursively as follows:

$$M(a,b) = \min_{a<w<b}\left\{ 1 + g(w,n,a,b)\cdot 0 + l(w,n,a,b)\cdot M(a,w) + r(w,n,a,b)\cdot M(w,b) \right\} \tag{2.7}$$

A boundary condition δ must be set to terminate the algorithm after a reasonable number of iterations. By assuming that all contention parameters differ by at least δ, contention can always be resolved in one step when the

window size is smaller than δ. Therefore, the following boundary condition is included:

$$M(a,b) = 1 \qquad \text{for } all \ \ (b-a) < \delta \ . \tag{2.8}$$

The dynamic programming algorithm requires an average of only 2.4 contention slots to resolve collisions. What is significant is that this value remains constant as the number of contending stations increases. Figure 2.3 illustrates the performance of the binary exponential backoff method as compared to the dynamic programming method.

There is an extensive computational overhead associated with this method. This overhead can be overcome in one of two ways. The first is to precompute $N(a,b)$ for a truncated dynamic programming algorithm and store the

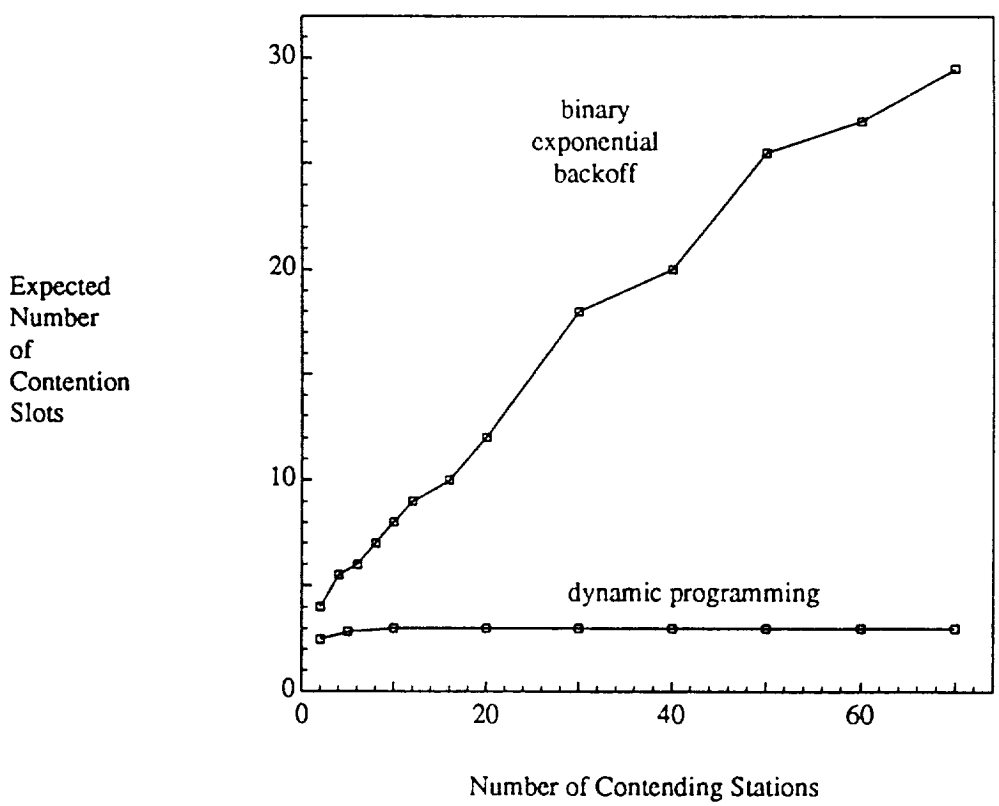


Figure 2.3. Binary Exponential Backoff vs. Dynamic Programming Window-Control

results of optimal w's for each (a,b) pair in a table. When $(b-a)<\delta$, a binary-divide algorithm can be used to complete the iterations. This method was used in the simulation results shown in Figure 2.3. The other method to reduce computation time involves using approximations to the dynamic programming algorithm as discussed in the next section.

2.2.4. Greedy Window-Control and Approximations

The computation time of the dynamic programming algorithm can be greatly reduced by finding a window to maximize the success probability in just the next iteration rather than in all possible future iterations. This technique requires an average of only 2.7 contention slots for bus contention [13]. Similar to the dynamic programming window-control algorithm, the overhead is also independent of the number of contending stations.

Computation time can be further reduced by using an approximation to the greedy window control scheme that calculates w directly from the following formula [13]:

$$w = \frac{-C - \sqrt{C^2 - 4D}}{2} \tag{2.9}$$

where

$$C = \frac{-(n-1)[F(a) + F(b)] + 2}{n} \tag{2.10}$$

$$D = \frac{F(a) + F(b) + (n-2)\cdot F(a)\cdot F(b)}{n} \tag{2.11}$$

This calculation of w has been shown to have a performance comparable to the optimal greedy window-control algorithm [13].

A final simplification is mentioned by Juang in his Ph.D. dissertation [17]. He showed that the window used in the first contention slot is critical to the overall performance. The ideal window for this first slot is $1/n$. After the first slot, a binary-divide scheme may be used to complete the algorithm. Again, the performance is comparable to that of the pure dynamic programming window-control algorithm.

2.2.5. Load Estimation

If the load of the channel is not exactly known, it must be estimated from the distributions of contention parameters and statistics of previous channel activity [13]. Wah and Juang have presented a method for estimating

load, based on the windows used to isolate the successful stations in previous contention slots [13]. The method computes a moving average based on all previous windows, w(1), w(2), ..., w(t). This moving average, $w_{mv}(t)$, is computed according to the following formula [13]:

$$w_{mv}(t) = \frac{w_{mv}(t-1) + w(t)}{2}. \tag{2.12}$$

The estimation of the number of stations, $n_e(t)$, is approximated by the formula [13]:

$$n_e(t) = \frac{-1}{\log(1-w_{mv}(t))}, \qquad 0 \le w_{mv}(t) < 1 \tag{2.13}$$

This estimate of n can be used in the approximate greedy window-control algorithm, yielding a performance very close to that when the channel load is exactly known. However, the estimate also assumes that there is no noise on the network. The effects of noise on load estimation will be discussed in Chapter 3 and illustrated by simulation in Chapter 4.

3. EFFECTS OF NOISE ON CSMA/CD BUS CONTENTION

One major problem in any communication network is noise. Noise on a network may be defined as any unwanted signal present in the transmission medium. This chapter discusses the types of noise, how noise can affect the physical operation of an Ethernet network, and how noise affects the various window-control algorithms for bus contention.

The effects of noise on the window-control algorithm are divided into two cases in which noise is either detected by all stations or by just some of the stations. In the first case, the windows of different stations remain in phase, but in the second case the windows may become out of phase. All window-control algorithms assume that the channel load is exactly known and that noise is independent for each contention slot. The last section discusses the effects of noise when these assumptions are invalid.

3.1. Types of Noise

There are basically four types of noise: thermal noise, intermodulation noise, crosstalk, and impulse noise [15]. Thermal noise is caused by the thermal agitation of electrons in the transmission medium. It varies as a function of temperature, but is otherwise fairly predictable. Thermal noise is often referred to as white noise [15]. Most communication systems take thermal noise into account in their design and use signals with sufficient strengths such that thermal noise has little effect. Although thermal noise limits the effective bandwidth of any communication medium, it normally does not cause signals to be misinterpreted by the receiver. Consequently, thermal noise is not a factor in regard to the window-control algorithm.

Intermodulation noise is due to signals of different frequencies producing a new signal that is the sum or difference of a multiple of those frequencies. Intermodulation noise is important when two or more simultaneous transmissions of different frequencies are attempted on the line. Since all transmission attempts in the window-control algorithms are made at the same frequency, intermodulation noise is also not a factor.

Crosstalk can be a problem depending on the communication medium used. It is rare in coaxial cable and nonexistent in optical fiber, but might occur in twisted-pair lines. *Crosstalk* is electric coupling between lines such that the signal on one line affects the signal on the other. Nevertheless, the magnitude of the noise produced by

crosstalk is relatively small and can usually be taken into account by the system designer [15].

Impulse noise, however, is a major factor in producing false signals. Impulse noise is an unpredictable burst of electrical energy that lasts a random amount of time. It can be caused by lightning, electromagnetic radiation, or any sudden pulse of energy. Such noise can often be found in harsh environments, such as those encountered in the military. Unlike the other types of noise, the randomness in the occurrence, duration, magnitude and nature of impulse noise makes it impossible to predictably filter out. Furthermore, impulse noise is usually of sufficient strength to cause logical 1's to be misinterpreted as logical 0's and logical 0's to be misinterpreted as logical 1's. It can also cause a whole block of transmitted signals to be altered dramatically. As a result, impulse noise has the greatest net effect on transmission on CSMA/CD networks and will be the main type of noise considered in this thesis.

3.2. Effects of Noise on Ethernet Network

To determine the effects of noise on bus contention on Ethernet networks, it is necessary to discuss the encoding of transmission signals on the interpretation of network status by stations, and the effects of noise on this interpretation. According to the physical layer specifications of Ethernets [16], an idle transmission line is set to a value of 0 volts. Data is encoded using *Manchester Code*, in which a logical 0 is represented by a transition from -0.225 V to -1.825 V, and a logical 1 is represented by a transition from -1.825 V to -0.225 V. Bits are transmitted at a frequency of 10 MHz, and the transitions are set to occur in the middle of each bit. Figure 3.1 illustrates the Ethernet specifications for its Manchester encoding scheme.

Any two different transmitting stations are likely to have their clocks skewed, and if simultaneous transmissions are attempted by both, the receiver will likely detect a collision either by a voltage level that is lower than it should be or by the lack of a detected transition in the data at the specified time. The receiver may also detect unexpected transitions less than 50 ns (1/2 bit time) apart. Figure 3.2 illustrates the results of a collision. Note that in the case of a collision, the overall high voltage is reduced from -.225 V to -.45 V and that the low voltage occurs at -3.65 V which is well below -1.825 V. Furthermore, there are two transitions less than 50 ns apart.

When collision is detected by the transmitting stations, the stations do not abort transmission immediately, but continue to transmit for a period of time to guarantee that the collision is heard by all stations [16]. The resultant

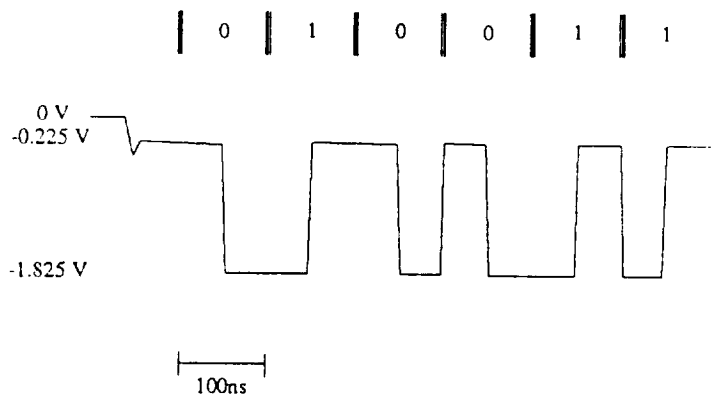


Figure 3.1. Manchester Encoding

transmission is known as a *jamming signal.*

What happens to a normal signal in the presence of noise? Most impulse noises introduced into the system will cause the voltage levels to be incorrect or cause the 10 MHz transition not to be detected. For example, a -1 V surge for 200 ns would cause the voltage levels to drop to -1.225 V and -2.825 V triggering a false collision, as illustrated in Figure 3.3. Figure 3.3 also shows how noise could interfere with the signal in such a way as to mask out the expected transition due to the 10 MHz clock. In this sense, it is relatively easy for noise to generate a false collision on the bus.

A false idle contention slot can occur only if there is a positive voltage surge that cancels out the signals present on the bus. As it is, a +2 volt surge will cause all voltages on the line to be greater than or equal to 0 when only one station is transmitting. Hence, it is necessary for the receiving station to recognize that voltages greater than 0 can be caused only by noise. The current Ethernet specifications [16] do not address this case, although minimal hardware can be added to the transceiver to detect positive voltages on the line. Under these conditions, the only way a false idle contention slot can occur is when the noise is the exact inverse of the signal, resulting in a net voltage of 0. This event is extremely unlikely for any signal, and hence, the probability that either a collided or

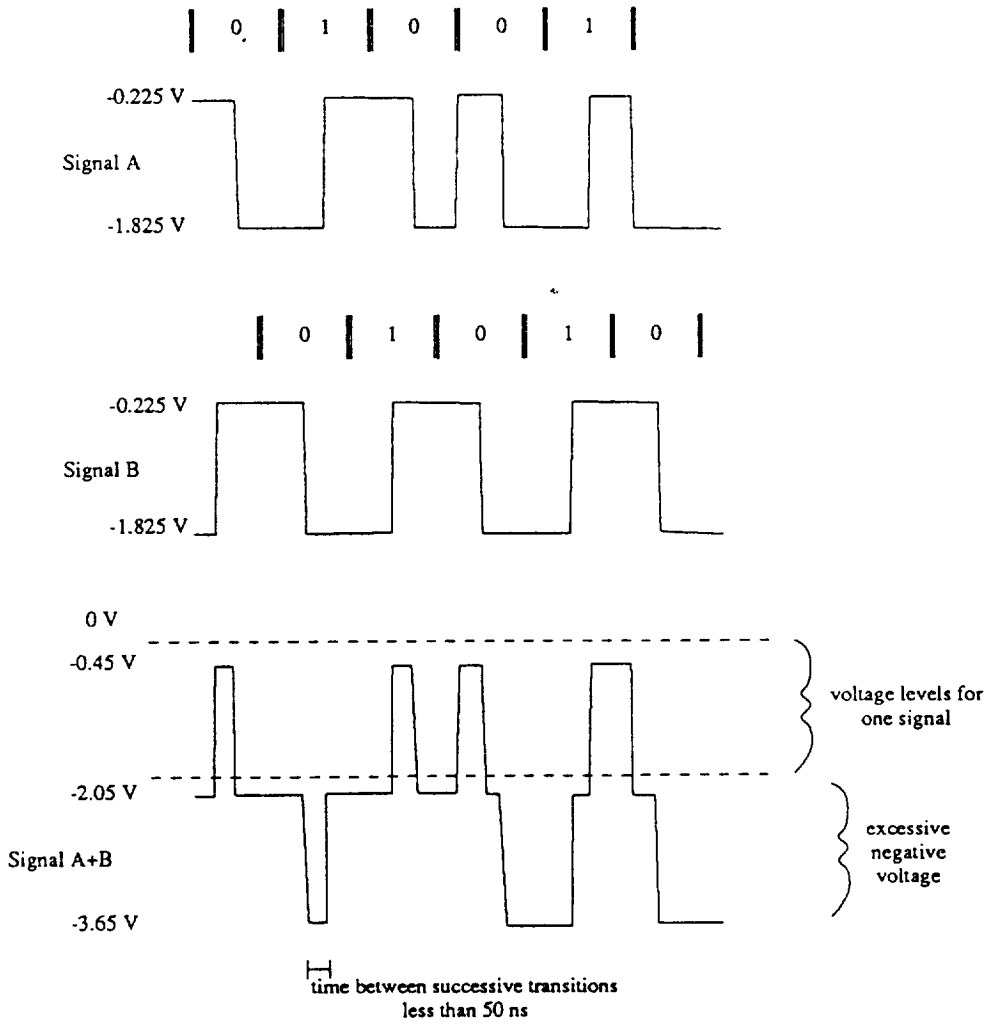


Figure 3.2. Collision of Two Signals

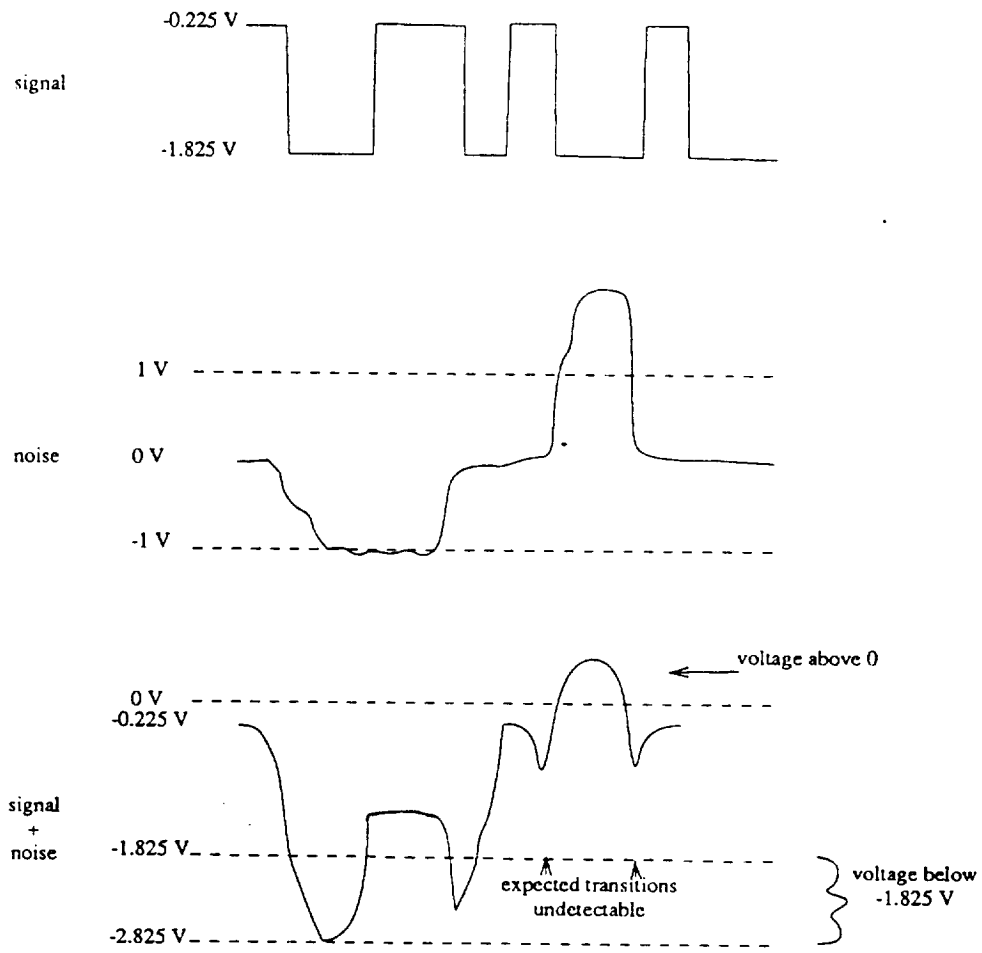


Figure 3.3. Signal with Added Impulse Noise

successful contention slot will falsely be detected as an idle slot is negligable.

It is also unlikely that a successful contention slot is falsely detected. In the case of an idle contention slot, a false success could be triggered by an arbitrary 10 MHz signal that varies between -.225 V and -1.825 V. Likewise, a collision could falsely be detected as a success when noise cancels out the collision in such a way as to produce a 10 MHz signal that varies within the -.225 to -1.825 voltage range. Both of the above are extremely unlikely since first, random noise is likely to have a voltage outside the range between -.225 V and -1.825 V, and second, even if the voltage levels are valid, the noise is likely to have a frequency other than 10 MHz. The case of a collision being falsely detected as a success is even more unlikely, since either transmitting station will detect a signal different from the one it transmitted, and enforce the collision with a jamming signal. In the worst case, if an idle contention slot is falsely treated as a successful transmission, all stations will be expecting the nonexistent successful station to transmit, resulting in an idle channel. This case can be logged as an error, and the stations can restart the algorithm.

All in all, the probabilities of a false idle slot or a false successful slot are extremely small, while the probability of a false collision in the presence of impulse noise is very likely. In modeling the effects of noise, a fourth state, noise, could be added to give additional information regarding the state of the network. However, noise is only distinguishable from a true collision when there are voltages greater than 0 on the transmission line. In all other cases, noise is indistinguishable from collisions. Therefore, for the purposes of evaluations, all noise will be treated as resulting in false collisions.

There is still the issue of whether all stations on the network will detect noise when it occurs. The probability that this is true is high for noise on the global bus since Ethernet is designed to allow collisions to be detected by all stations through use of a jamming signal. It is very unlikely that the jamming signal will be misinterpreted as a false success or no transmission since the transmitting station is also listening to the line and can continue to issue the jamming signal if it detects that the signal is being cancelled out by noise. This leaves the problem, however, of noise occuring while no stations are transmitting. In this case, no station would issue a jamming signal to amplify the false collision. Whether a station detects a collision will then be dependent on the sensitivity of a stations's transmission-line sensing equipment.

Unfortunately, one cannot guarantee that all stations will always detect all noise whenever it occurs. Consider the Ethernet network in Figure 3.4. If noise is more prevalent only on the right side of the repeater, there will be

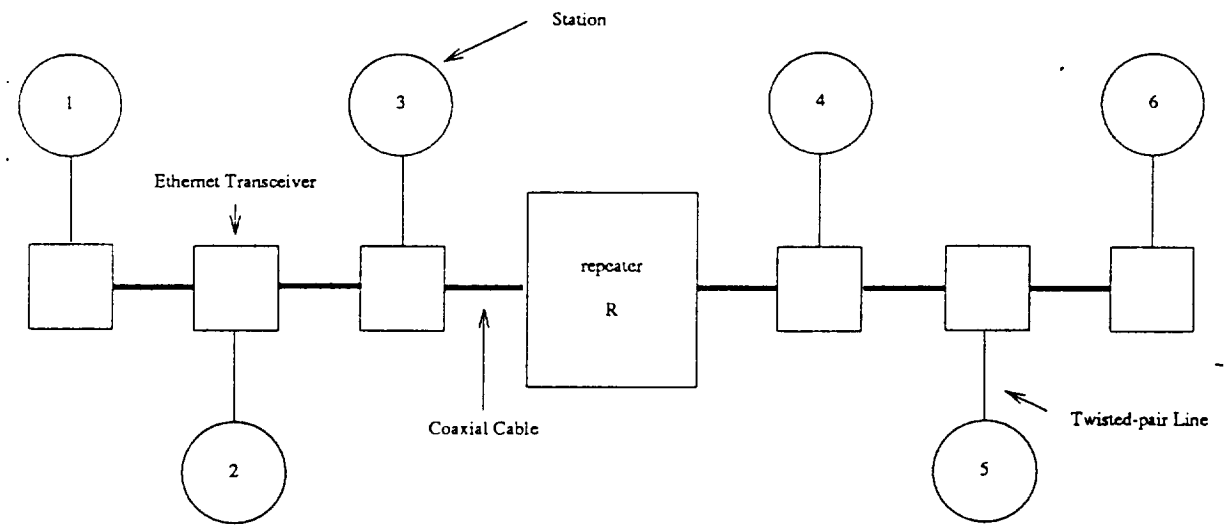


Figure 3.4. An Ethernet Network

occasions when stations 4,5 and 6 detect noise while stations 1,2 and 3 do not. If station 4 has a noisy connection to the network, it might detect noise, while the others do not. There might also be the case where the collision and noise detection mechanisms are more sensitive in some stations than others. Basically, any condition that causes noise to be localized to one or more areas of the network, or any discrepancy in transmission-line sensing equipment of some stations, may result in the stations detecting different states of the network during a given contention slot.

3.3. Effects of Noise on Window-Control Algorithms

The previous section discusses the fact that noise will almost always result in the detection of a collision by the stations on the network. If there is already a collision, noise has no effect. But if there is no collision, some or all of the stations may detect a false collision due to the noise, and inappropriately adjust their windows.

Analysis of the noise problem can be divided into two cases. The first case assumes that noise is always detected by all stations, and that all stations treat noise as a collision. Provided that the stations start with the same windows in the first contention slot, their windows will remain synchronized with respect to each other, since they always detect the same state of the network. The second case is more general and assumes that a subset of the stations detect noise while others do not. As a result, it is possible for stations to have different window boundaries in a

given contention slot. These two cases are discussed separately in the following sections.

3.3.1. Noise Detected by All Stations

The assumption that the windows remain synchronized is necessary in developing a theoretical analysis for the performance of the window control algorithms under the presence of noise. The following sections discuss the effects of noise on the window-control algorithm along with the development of two new window-control algorithms that attempt to overcome the noise problem-- the two-phase algorithm and an approximate recurrence relation.

3.3.1.1. Original Window-Control Algorithm

When noise is apparent, windows in the original window-control algorithm may be updated incorrectly. Consider the scenario in Figure 3.5, where there is a window with no contention parameters in it and noise occurs in that slot. After the first real collision, the upper bound of the window is reduced to w_1, but all contention parameters are greater than w_1. Now suppose there is a burst of noise in the next contention slot that causes all stations to detect a false collision and reduce the upper bounds of their windows to w_2. In successive iterations, idle contention slots are detected and cause the upper bounds of the windows to increase to w_3,w_4,w_5, and so on, but all are less than w_1. In any event, the upper bound of the windows will never become greater than w_1, and the contention protocol will con-

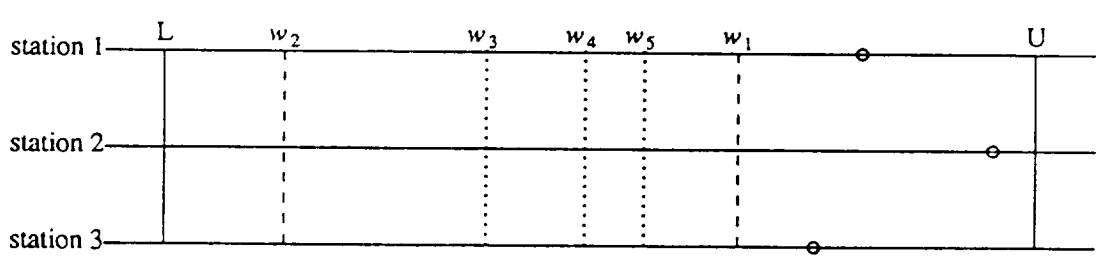


Figure 3.5. Window-Control with Noise

tinue to iterate indefinitely.

To correct this problem, it is necessary to terminate the protocol after the window has been reduced to a certain size. The termination point can be set to the value δ, the resolution of the contention parameters, as defined in Chapter 2. If the window is reduced to δ, the algorithm must be restarted. The number of contention slots to resolve the algorithm will then depend on the probability that collision is resolved before the window is reduced to a size δ and the associated probability for this event. If the probability of noise is high, then it is highly likely that the window will be reduced to a size less than δ without isolating the minimum. If δ is small, then extensive overheads are incurred before false collisions are detected. With respect to the window protocols outlined in Chapter 2, the number of contention slots to resolve collision has to be modified by the probability that collision is resolved before the window is reduced to a size δ. Let M be the number of contention slots to either isolate the minimum or reduce the window size to δ, and S the probability of success before this event occurs. Define T as the total number of contention slots to resolve contention. T is then formulated as follows:

$$T = S \cdot M + S(1-S) \cdot 2M + S(1-S)^2 \cdot 3M + S(1-S)^3 \cdot 4M + \cdots \tag{3.1}$$

$$T = SM \cdot \sum_{i=1}^{\infty} i \cdot (1-S)^{i-1} \tag{3.2}$$

$$T(1-S) = SM \cdot \sum_{i=1}^{\infty} i \cdot (1-S)^{i} \tag{3.3}$$

Subtracting (3.3) from (3.2) yields:

$$TS = SM \cdot \sum_{i=1}^{\infty} (1-S)^{i} \tag{3.4}$$

The summation in (3.4) is a standard geometric distribution that reduces to $1/S$ [18]. Therefore, Equation (3.4) reduces to:

$$T = M/S \tag{3.5}$$

3.3.1.2. Two-Phase Algorithm

The original window protocol suffers from the problem that once the upper bound of the window is reduced to a value smaller than the smallest contention parameter, it is not possible to recover until the window size is reduced to a size less than δ. To alleviate this problem, one can use a two-phase algorithm that adds a check phase each time no transmission is detected. If no transmission is detected in a given contention slot, then the other half of the window is tested to see whether it contains any contention parameters. If there is a collision, the protocol carries on as normal, but if no transmission is detected again, then the window's upper bound is increased to its previous value, while the lower bound is increased to the current upper bound. Referring to Figure 3.5, the two-phase algorithm would modify the situation as follows:

Slot 1) Collision:
window reduced to $(L,w_1]$.
Slot 2) Collision due to noise, although no transmission should be detected:
window reduced to $(L,w_2]$.
Slot 3) No transmission:
window raised to $(w_2,w_1]$.
Note: In the original window protocol, the window used is $(w_2,w_3]$.
Slot 4) Second no transmission:
window raised to $(w_1,w_0]$.
Slot 5) Protocol continues normally, unless further noise develops.

The state of no transmission is used to correct the windows, since whenever no transmission occurs, it is guaranteed, based on the assumptions discussed earlier, that the transmission line is definitely in that state. Therefore, whenever no transmissions occur twice in sequence, it is guaranteed that no contention parameters remain within the lower and upper bounds defined for the window. As a result, it is necessary to retract the lower and upper bounds to a different region of the contention parameters. This process is repeated until a region with a collision is found. Since a collision may be due to noise, the occurrence of collision in a contention slot cannot be used to permanently fix the upper bound of the window. If the probability of noise is low, then most of the collisions are due to simultaneous transmissions in a window. The algorithm is set to terminate when a successful transmission is detected, since it is assumed that there will be no false detections in this case.

Inherently, the two-phase algorithm implies need of additional memory to store previous upper window boundaries. However, these bounds are normally defined in the implementation of the window-control algorithms, as they are precomputed at design time and looked up in real time. In the case in which the window boundaries are

computed in real time, such as the binary-divide algorithm, the maximum number of stored window boundaries is bounded by the difference between the number of collisions and the number of no transmissions.

This can be best illustrated by the binary-divide window control. Assume an initial window size of 1 and a resolution $\delta = 2^{-n}$. Each collision causes the window size to decrease by a factor of two and increases by one the number of upper bounds that must be stored in memory. However, each no transmission retrieves a previous value of the upper bound, and therefore decreases the number of upper bounds to be stored by one. In the worst case there will be n successive collisions, resulting in n upper bounds stored. The next collision will force the algorithm to restart, as the maximum number of collisions is bounded by n. Although each collision decreases the window size by a factor of two, each no transmission increases the window size by a factor of two only after the second no transmission is detected. It takes only one collision to decrease the size of the window, but it takes at least two no transmissions to increase the window size. Unless there is a sequence of n collisions, the window shrinks at a rate that is faster than the net growth of the memory stack and, therefore, the maximum amount of memory needed to store the upper bounds is n. A similar result holds true for the dynamic-programming protocol, except that during collisions the window is reduced by at least a factor of two. Hence, the required memory is likely to be less than n.

There is additional computational overhead to check the other half of the window in the two-phase protocol. Although the probability of success is increased in the two-phase protocol over the original protocol, the number of contention slots to reduce the window size to δ is also increased. Therefore, the net effect on the total number of contention slots to resolve contention, as defined by Equation 3.5, is uncertain. A comparison of the performance of the two-phase and the original protocols is included in the next chapter.

3.3.1.3. Heuristic Window-Control with Noise

The optimal window for each contention slot can be determined by including the probability of noise as a parameter in the dynamic programming window-control algorithm. First, one must consider the effects of noise on state detection and window boundaries. No information concerning the state of the network can be gained during the contention slot in which noise occurs. Windows can be updated with certainty after detection of a successful transmission or idle state on the network, since those states can occur only when there is no noise. Therefore, one can always be certain of the lower bound of the window. Windows can be updated only conditionally after detec-

tion of a collision, because the collision may be due to noise. Therefore, one cannot be certain of the upper bound of the window.

During a collision, if noise does not occur, it is guaranteed that there are at least two contention parameters that are less than the upper bound of the window. However, if noise occurs during a collision, one can guarantee only that there are at least two contention parameters less than the upper bound of some previous window in which noise did not occur. Let a *true collision* be defined as a collision in which no noise occurs. One can guarantee that there are at least two contention parameters smaller than the smallest previous upper bound of the window in which a true collision occurred. Unfortunately, it is not possible to differentiate between true and false collisions, so one must keep track of the upper bounds of the windows from all previous collisions. Since the choice of the window is history dependent, it cannot be optimized by dynamic programming methods because the Principle of Optimality is not satisfied.

Assume that the probability of noise is independent for each contention slot and has a fixed value, p, that is known by all stations. Also, assume that the number of contending stations, n, is known. Finally, assume that each contention parameter, x_i, has a distribution, $F(x)$, and lies in the interval $(L,U]$. The following definitions are used to formulate the problem of choosing the upper bound as a recurrence.

$M(a,q[k])$:
The minimum expected number of contention slots to resolve contention, given k-1 previous collisions with upper bounds of windows stored in the k-element array, $q[k]$.

$g(a,w,q[k])$:
The probability of successful transmission in the interval $(a,w]$, given k-1 previous collisions with upper bounds of windows stored in the k-element array, $q[k]$.

$r(a,w,q[k])$:
The probability of no transmission in the interval $(a,w]$, given k-1 previous collisions with upper bounds of windows stored in the k-element array, $q[k]$.

$l(a,w,q[k])$:
The probability of collision in the interval $(a,w]$, given k-1 previous collisions with upper bounds of windows stored in the k-element array, $q[k]$.

In the array, $q[k]$, $q_1=U$, and q_i is the upper bound of the window during the $(i-1)^{th}$ collision. It follows from the above definitions that:

$$g(a,w,q[k]) + r(a,w,q[k]) + l(a,w,q[k]) = 1. \tag{3.6}$$

Let the elements of $q[k]$ be ordered from smallest to largest; q_1 is now $min(q_1 \cdots q_k)$ and $q_k = U$. Let a *true upper bound* be defined as the upper bound of the window in which a true collision occurred. Since p is independent for each contention slot, each q_i, except q_k, has a probability p of being a true upper bound, and $(1-p)$ of being false. The probability that q_1 is the smallest true upper bound is $(1-p)$. The probability that q_i is the smallest true upper bound is equal to the probability that all smaller q_i's are false and q_i is true. The probability that all q_i's are false is equal to p^{k-1}. Let $Pr(Q_i)$ be the probability that q_i is the smallest true upper bound.

$$Pr(Q_i) = \begin{cases} p^{i-1}\cdot(1-p) & 1 \le i < k \\ p^{k-1} & i = k \end{cases} \tag{3.7}$$

All Q_i's are mutually exclusive, so $\sum_{i=1}^{k} Pr(Q_i) = 1$. If q_i is a true upper bound, then there are at least two contention parameters in the interval $(a,q_i]$ and all x_i's are in the interval $(a,U]$. This condition is designated as event $(A \mid Q_i)$, with the following probability:

$$Pr(A \mid Q_i) = \frac{[1-F(a)]^n - [F(q_i)-F(a)]\cdot[1-F(q_i)]^{n-1} - [1-F(q_i)]^n}{[1-F(a)]^n} \tag{3.8}$$

If q_i is the smallest true upper bound, three mutually exclusive events can occur by choosing $a < w < q_i$, which can be identified as follows:

B = { exactly one x_i is in $(a,w]$, given A $\mid Q_i$ }
C = { no x_i is in $(a,w]$, given A $\mid Q_i$ }
D = { more than one x_i is in $(a,w]$, given A $\mid Q_i$ }

Using Bayes' rule [19], the probabilities of $g(a,w,q[k])$ and $r(a,w,q[k])$ can be expressed for all q_i's as:

$$g(a,w,q[k]) = \sum_{\substack{i=1 \\ w<q_i}}^{k} Pr(Q_i)\cdot Pr(B \mid A \mid Q_i) = \sum_{\substack{i=1 \\ w<q_i}}^{k} Pr(Q_i)\cdot \frac{Pr((A \mid Q_i) \cap B)}{Pr(A \mid Q_i)} \tag{3.9}$$

$$r(a,w,q[k]) = \sum_{\substack{i=1 \\ w<q_i}}^{k} Pr(Q_i)\cdot Pr(C \mid A \mid Q_i) = \sum_{\substack{i=1 \\ w<q_i}}^{k} Pr(Q_i)\cdot \frac{Pr((A \mid Q_i) \cap C)}{Pr(A \mid Q_i)} \tag{3.10}$$

The set $((A \mid Q_i) \cap B)$ represents the event that exactly one x_i is in $(a,w]$, that at least one x_i is in $(w,q_i]$, and that all other x_i's are in $(w,U]$. Appropriate substitution in Equation (3.9) yields:

$$g(a,w,q[k]) = \sum_{\substack{i=1 \\ w<q_i}}^{k} Pr(Q_i) \cdot \frac{[F(w)-F(a)]\cdot[[1-F(w)]^{n-1} - [1-F(q_i)]^{n-1}]}{Pr(A \mid Q_i)\cdot[1-F(a)]^n} \tag{3.11}$$

The set $((A \mid Q_i) \cap C)$ represents the event that at least two x_i's are in $(w,q_i]$, and all x_i's are in $(w,U]$. Appropriate substitution in Equation (3.10) yields:

$$r(a,w,q[k]) = \sum_{\substack{i=1 \\ w<q_i}}^{k} Pr(Q_i) \cdot \frac{[1-F(w)]^n - [F(q_i) - F(w)]\cdot[1-F(q_i)]^{n-1} - [1-F(q_i)]^{n-1}}{Pr(A \mid Q_i)\cdot[1-F(a)]^n} \tag{3.12}$$

The heuristic recurrence relation can then be expressed as:

$$M(a,q[k]) = \min_{a<w<U} \left\{ 1 + g(a,w,q[k])\cdot 0 + l(a,w,q[k])\cdot M(a,q[k+1]) + r(a,w,q[k])\cdot M(w,q[k]) \right\} \tag{3.13}$$

The recurrence terminates upon detection of successful transmission. This is represented by the term $g(a,w,q[k])\cdot 0$ in Equation (3.13). However, since the number of collisions may be infinite, the number of states is unbounded, and Equation (3.13) cannot be solved. Consequently, only an approximation of the exact recurrence equation may be realized.

The following approximation attempts to reduce the number of states to a tractable finite value, by making certain assumptions. It assumes that the number of contending stations, n, and the probability of noise, p, are known at the start of contention. It also assumes p is independent for each contention slot. Furthermore, the approximation assumes that the upper bound of the next window chosen, w, is always less than the upper bound of the current window, and that after a collision, the upper bound of the window is reduced to w. Finally, although the conditional probabilities of success, idle and collision are dependent on all previous upper bounds of the windows, the only upper bounds used are the current upper bound and the absolute upper bound of the contention parameters. The probabilities in Equations (3.6-3.12) are then solved by using $k=2$, $q_1=b$ and $q_2=U$.

If one truncates the recurrence relation at $(b-a) < \delta$, it is possible that contention may not be resolved. Therefore, $M(a,q[k])$ must be redefined as the estimated number of contention slots to resolve contention either successfully or unsuccessfully, given that the current window is $(a,b]$ and that all contention parameters are in $(a,U]$. Since the only variable upper bound of the window is b, the approximate recurrence equation can be renamed in terms of a, w and b. Using the assumptions previously stated, the approximate recurrence equation is:

$$M(a,b) = \min_{a<w<b} \left\{ 1 + g(a,w,q[2])\cdot 0 + l(a,w,q[2])\cdot M(a,w) + r(a,w,q[2])\cdot M(w,b) \right\} \tag{3.14}$$

Since the recurrence relation always terminates when $(b-a) < \delta$, the boundary condition for $M(a,b)$ is:

$$M(a,b) = 1 \quad when \;\; b-a < \delta \tag{3.15}$$

As stated in equation (3.5), the total number of contention slots to resolve contention successfully is M/S. However, S is also dependent on all previous windows, and is as complex to model as M. It can also be approximated as a recurrence equation using the same assumptions that were used to generate $M(a,b)$ in Equation (3.14). Let $S(a,b)$ be the probability of successfully isolating the minimum, given the current window, $(a,b]$, upper bound of next window, w, and all contention parameters in the interval (a,U]. $S(a,b)$ is the probability of success in the current window plus the probability of collision times the probability of success in window $(a,w]$ plus the probability of no transmission times the probability of success in window $(w,b]$. $S(a,b)$ can then be formulated as:

$$S(a,b) = \left\{ g(a,w,q[2]) + l(a,w,q[2])\cdot S(a,w) + r(a,w,q[2])\cdot S(w,b) \right\} \tag{3.16}$$

The boundary condition for $S(a,b)$ when $b-a < \delta$ is the probability that there is exactly one contention parameter in the window $(a,b]$. This is equal to $g(a,b,q[2])$.

The minimum number of contention slots, $T(a,b)$, is then defined as:

$$T(a,b) = \min_{a<w<b} \frac{M(a,b)}{S(a,b)} \tag{3.17}$$

$$T(a,b) = \frac{1}{g(a,b,q[2])} \qquad \text{for } b-a < \delta \tag{3.18}$$

It must be guaranteed that the w chosen for each $M(a,b)$ is consistent with the w chosen for each $S(a,b)$. Since $M(a,b)$ and $S(a,b)$ are computed from smaller subintervals, the w's for these subintervals can be computed and stored in a lookup table to be shared by both $M(a,b)$ and $S(a,b)$. The algorithm for deriving $T(a,b)$ is simulated in the next chapter, and the associated program is given in Appendix B.

The approximate recurrence relation also requires additional memory overhead for implementation. While the original dynamic programming algorithm needs a separate table entry for each n, the recurrence relation requires a separate table entry for each combination of n and p. The number of entries for each (n,p) pair can be reduced by using a truncated binary decision tree as outlined by Juang and Wah [13]. The decision tree chooses one of two

paths depending on whether a station detects collision or an idle line. No path is needed for the detection of a successful transmission, since such detection terminates the algorithm.

Using a four-level decision tree with 16-bit entries, and 16-bit random numbers for the contention parameters, each (n,p) pair would then require 0.03 Kbytes of memory. The standard Ethernet controller chip, MCS 8396, has 8 Kbytes of ROM available, so 266 (n,p) pairs may be stored. The next chapter discusses a possible implementation of the approximate recurrence relation in a real system.

3.3.2. Windows Becoming Out-of-Phase

This section discusses the case of the windows of the contending stations becoming asynchronized due to noise. As was discussed earlier, it is possible that not all stations may sense the same state of the network during a given contention slot. Figure 3.6 illustrates how this can affect the window-control protocol. During the first contention slot, all stations except station 1 transmit, and all stations reduce the upper bounds of their windows to w_2. Suppose in the second contention slot, station 2 detects noise while the others do not. It will then lower its upper bound to $w_3(2)$, while the other stations detect no transmission and raise the upper bounds of their windows to w_3.

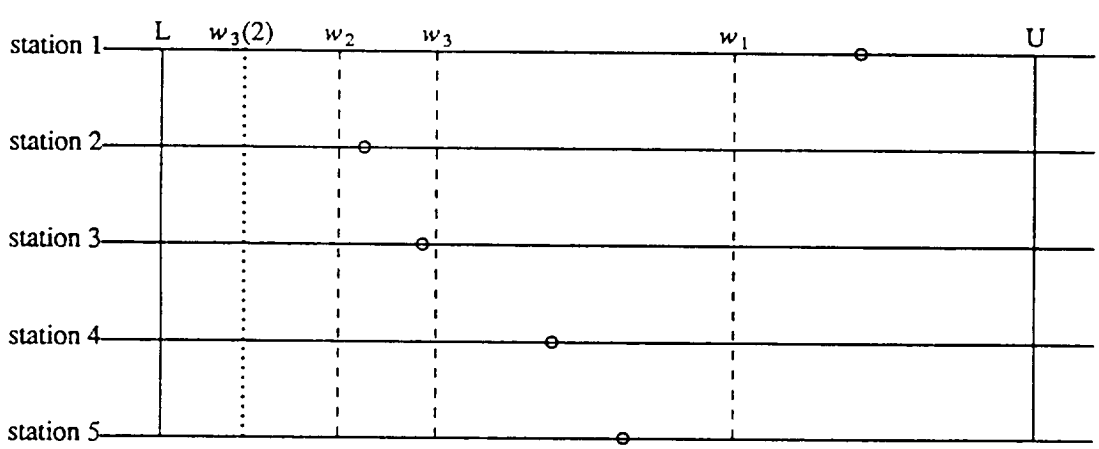


Figure 3.6. Wrong Minimum Identified by Window-Control Algorithm when Windows are Out-of-Phase

Now, during the third contention slot, only station 3 transmits and is successful. Station 3 is, therefore, falsely identified as the minimum.

To correct this problem, one logical solution is to add a broadcast phase after the minimum has been identified. After a station transmits successfully, it broadcasts its parameter, x, to all other stations in the next message. Following the first broadcast, all stations with contention parameters less than x attempt to broadcast their parameters in a second broadcast. If there is a collision, all stations restart the window-control algorithm with the upper bound of the contention parameters set to x. If some station broadcasts its parameter successfully during the second broadcast phase, it captures the bus and begins transmission of its message. Otherwise, if the line remains idle during the second broadcast phase, the original successful station captures the bus and transmits its message.

There is an additional problem in that all stations may not terminate the algorithm at the same time. This problem can be alleviated by the successful station continuing to transmit in the next σ contention slots before broadcasting its parameter, where σ must be long enough to guarantee that all stations have an opportunity to either detect that successful transmission is occurring or else timeout and terminate the algorithm. During the contention slot that a station succesfully transmits, it will be the only station with a contention parameter smaller than the upper bound of its window. Since the other stations' contention parameters are higher than the upper bounds of their windows, they will continue to sense the line but will not transmit. In the following contention slots, one of two events will occur to terminate iterations of new windows. The unsuccessful stations can either detect success and wait for the broadcast, or they can detect a series of false collisions and reduce the upper bounds of their windows until their window sizes become smaller than δ. The number of contention slots to reduce a window from its original size to δ, given a series of collisions, is $O((log_2(1/\delta))$, and is the lower bound on σ. The event of a station reducing its window to a size smaller than δ can be designated as the start of a timeout period. If a station senses a successful transmission within this timeout period, it listens for the broadcasted parameter. If a station senses an idle line within the timeout period, it restarts the algorithm. Otherwise, if a station continues to sense false collisions during the entire timeout period, it aborts its contention attempt.

If a station aborts contention due to a timeout, or detects success in the middle of the broadcast phase, it will not know the value of the broadcasted parameter. Also the value of the broadcast parameter might become corrupted, and therefore, unreadable due to noise. These problems can be remedied by all stations involved in such

events forcing a collision in the second broadcast phase, and restarting their algorithms with an upper bound of U. In this case, the upper bounds of the windows may not be the same for all stations in the restarted algorithm, but this situation will be corrected during the next broadcast phase. Operation of the window-control algorithm with broadcast phase is illustrated in Figure 3.7.

The broadcast phase adds significant overhead to the time required to resolve contention. For applications where it is not necessary to identify the minimum contention parameter, the broadcast phase can be deleted. The successful station automatically captures the bus, and all other stations abort contention upon either detecting success or reaching a timeout. Even when all stations detect noise independently, the performance of the window-control algorithm is still better than Ethernet's exponential backoff algorithm. These results are shown in Chapter 4.

3.4. Dependent Noise

In reality, noise is often longer than one contention slot in length. Therefore, the probability that noise occurs during a given contention slot is dependent on whether noise occurred in previous contention slots. To model dependent noise, it is necessary to retain state information for the entire history of contention. Because of the complexity of modelling dependent noise, this section discusses only its general effects and does not attempt a detailed analysis.

In regards to the window-control algorithms, noise in successive contention slots will result in false collisions that continue to reduce the window. If the noise terminates before the window is reduced to a size less than δ, then the algorithms continue normally. If the noise continues after the window is reduced to a size less than δ, the stations enter a timeout period, as discussed in the last section. If the noise abates before the end of the timeout period, then the stations will restart the window-control algorithm; otherwise, the stations will abort the algorithm. The general effect of longer bursts of noise is that the probability of reducing the window to a size less than δ without being successful is greater. As a result, the algorithm is more likely to need to be restarted, and the total number of contention slots to resolve contention is increased. The behavior of the system under long bursts of noise should be similar to the behavior of the independent noise models for very high probabilities of noise.

On the other hand, short bursts of noise less than one contention slot in length will only affect the current contention slot. Therefore, such noise can be modelled as independent noise. If the burst of noise is too short, it might

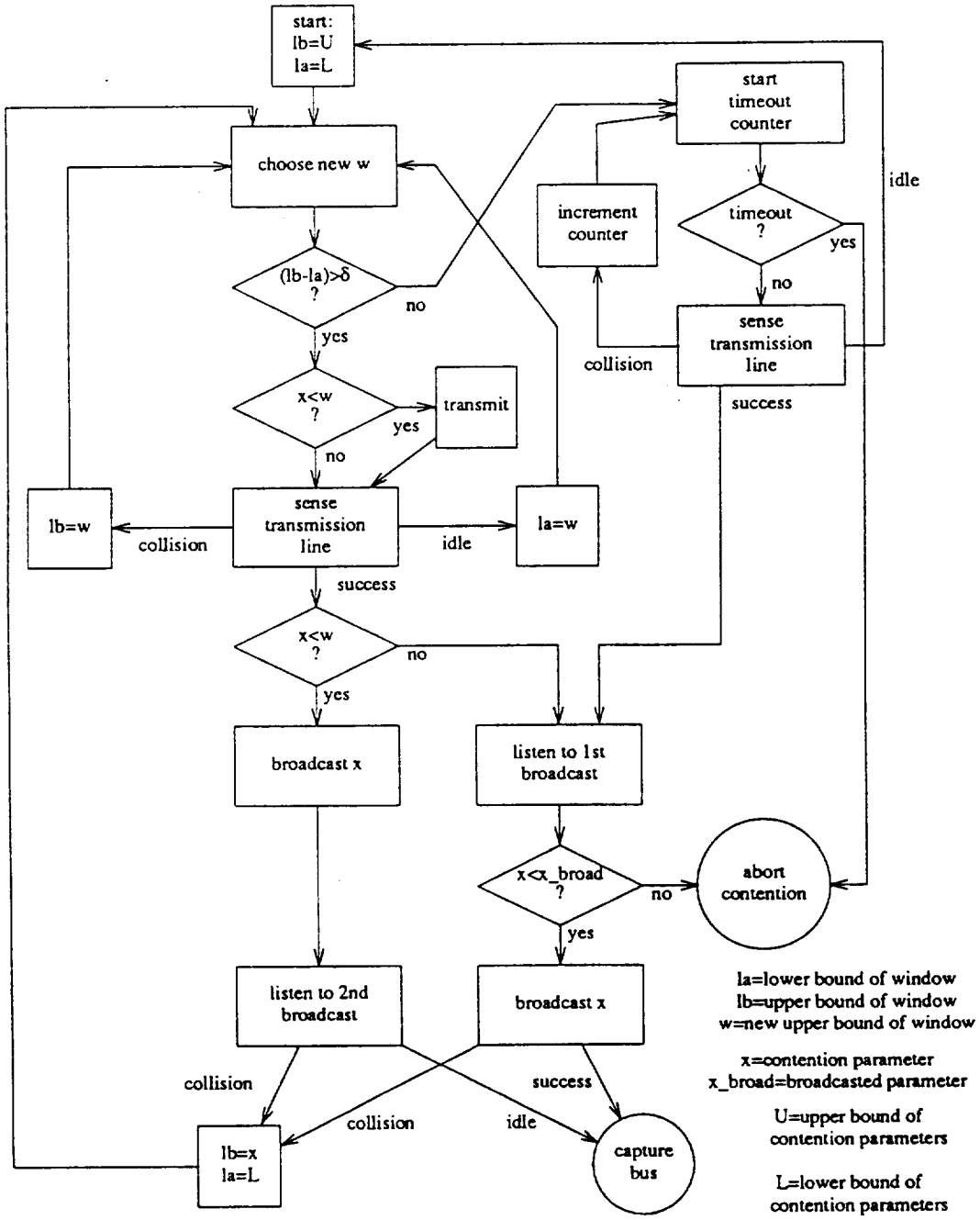


Figure 3.7. Window-Control Algorithm with Broadcast Phase

not be detected by all stations, but this problem was already discussed. Consequently, although the probability of noise is actually history dependent, the general behavior of the system can still be fairly accurately determined assuming a fixed, independent probability of noise for each contention slot.

3.5. Load Estimation

Noise also has an effect on load estimation. As stated in Chapter 2, the channel load is estimated according to a moving average of the windows used to successfully isolate the minimum contention parameter. However, noise causes false collisions such that the window used to isolate the minimum contention parameter may be lower than the window that is actually needed. Consequently, the added collisions due to noise may make the network appear as if the channel load was higher. The next chapter simulates the effects of noise on load estimation for the approximate greedy window-control algorithm.

3.6. Summary

This chapter has shown that noise can affect the detection of the states, collision, idle and successful transmission on the network. Noise may result in the false detection of a collision, but it will almost never result in the false detection of a successful transmission or idle line. Noise can also cause the window boundaries of the contending stations to become out of phase when its detection is localized to a particular station or set of stations.

This chapter outlined the effects of noise on the original window-control algorithm for the case where noise is detected by all stations. It also presented two protocols for countering these effects: the two-phase algorithm and the approximate recurrence relation given by Equation (3.18). In all protocols, it is possible for the window to be reduced to a size less than δ without isolating a station. In such cases, the algorithms must be restarted.

In the case where the windows may become out of phase, it is possible for the window-control algorithm to isolate a station that does not have the minimum contention parameter. Therefore, a broadcast phase must be added to correct for this phenomenon. The next chapter compares the performance of the various window-control algorithms through simulation, and discusses the implementation of a window-control algorithm in a real system in which noise is a factor.

4. SIMULATION AND ANALYSIS OF WINDOW-CONTROL PROTOCOLS

This section outlines simulation models for the binary exponential backoff algorithm and window-control algorithms and uses the simulations to gauge the performance of the models under various conditions. It then uses the results to present an implementation of a window-control algorithm in a real system in which noise is a factor. The first simulations assume that the channel load is exactly known, that noise is independent for each contention slot and that windows remain synchronized. They are used primarily to show the general effect of noise on the various algorithms. The next set of simulations shows what happens when the windows in the window-control algorithms become asynchronized. The last simulations show the results of load estimation on the approximate greedy window-control algorithm. All simulations were run on SUN 3/50 and SUN 3/260 workstations. The simulations involving the original dynamic programming lookup table and the binary exponential backoff algorithm were written in Fortran77, and all other simulations were written in C. The associated programs are included in the appendices.

All the simulations were performed with a 95% confidence interval C, which is ±5% of the mean, computed as follows [19]:

$$C = \pm\, 1.96 \times \sqrt{\frac{\sigma^2}{Z}}$$

In the equation above, Z is the total number of trials performed, and σ^2 is the variance of the sample. In the simulations, C was calculated arbitrarily after every 800 trials. If C was in the interval $\pm[\, 0.05 \times mean \,]$, then the simulation was terminated, otherwise it was run again with another 800 trials. Consequently, Z was some multiple of 800.

All simulations considered noise probabilities ranging from 0 to 0.9. The asymptotic value for the probability of noise is 1.0. When the probability of noise is 1.0, the algorithms can never terminate, and the number of contention slots to resolve contention is infinite.

4.1. Independent Noise, Synchronized Windows

All the following models assume that for each contention slot, all stations sense the same state of the network, and noise is independently generated in each contention slot with a fixed probability. In the simulations, noise was modelled by generating a random number during each contention slot. If that random number was less than the

threshold of noise defined for a given simulation, then the network was treated as if it were in a state of collision, regardless of whether or not there was an actual collision during that contention slot. The contention parameters were chosen as uniformly distributed random numbers in the interval [0,1). Upon restart of the algorithm, the contention parameters were regenerated to avoid infinite loops when the two smallest contention parameters differ by less than δ. In all the protocols, the number of stations contending in a given slot was counted and placed in h. If there was noise or $h > 1$, then there was a collision. If $h = 0$, then the line was idle. Otherwise, if $h = 1$, then the algorithm terminated successfully.

4.1.1. Binary Exponential Backoff Algorithm

Figure 4.1 illustrates the effects of noise on the binary exponential backoff algorithm for 10, 20, 30, 40 and 50 contending stations. Note how the performance degrades fairly evenly for probabilities of noise < 0.8, as the number of stations is increased. This corresponds to the linear degradation in performance of the binary exponential backoff

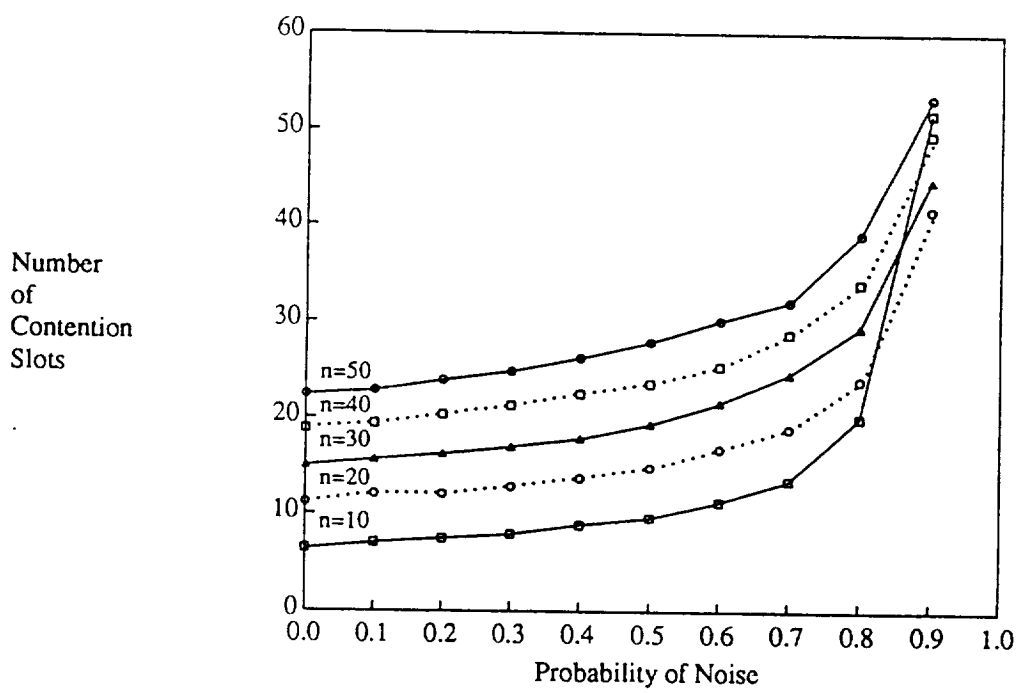


Figure 4.1. Binary Exponential Backoff Algorithm
(n=10,20,30,40,50)

algorithm when noise is not present (Figure 2.2). Under the presence of most levels of noise, the binary exponential backoff algorithm still degrades linearly as the number of stations, n, increases.

There seems to be an anomaly for the number of contention slots at 90% noise and 10 contending stations, but this can be explained from the fact that the exponential backoff algorithm causes the mean waiting time between successive transmissions to become greater for each retransmission. As the number of contention slots becomes greater, there are more empty slots. Early in the algorithm, the probabilities of collision are much higher when n is large than when n is small. However, at some point in the algorithm, the probability of no transmission is greater when n is small than when n is large. Higher probabilities of noise tend to cause the number of contention slots to increase to the point where this is the case. Consequently, at very high probabilities of noise, the performance of the algorithm when n is small may be worse than when n is large. Another consequence of this result is that noise has a greater effect when n is small than when n is large. This can be seen from Figure 4.1 where the performance curve is much steeper for $n=10$ than for $n=50$.

4.1.2. Binary-Divide Window-Control Algorithm

The first window-control simulation shows the effects of noise on the binary-divide window-control algorithm. The simulation was written in C and simulated on a SUN 3/260. The algorithm was restarted if contention was not resolved when $\delta \leq 1/(10n)$.

Figure 4.2 shows the various performance curves for 10, 20, 30, 40 and 50 contending stations. Note that the performance degrades logarithmically as the probability of noise increases. The performance also degrades as the number of stations increases, but nonlinearly, unlike the binary exponential backoff algorithm.

4.1.3. Original Dynamic Programming Window-Control Algorithm

The dynamic window-control simulations were written in Fortran and simulated on a SUN 3/50, using the dynamic programming method with lookup table as outlined in Chapter 2. The dynamic programming algorithm was truncated when the window size, δ_1, became less then $1/(10n)$ as in Wah and Juang's model [13]. If the window size became smaller than δ_1, a binary-divide algorithm was used to complete the iterations. The binary-divide

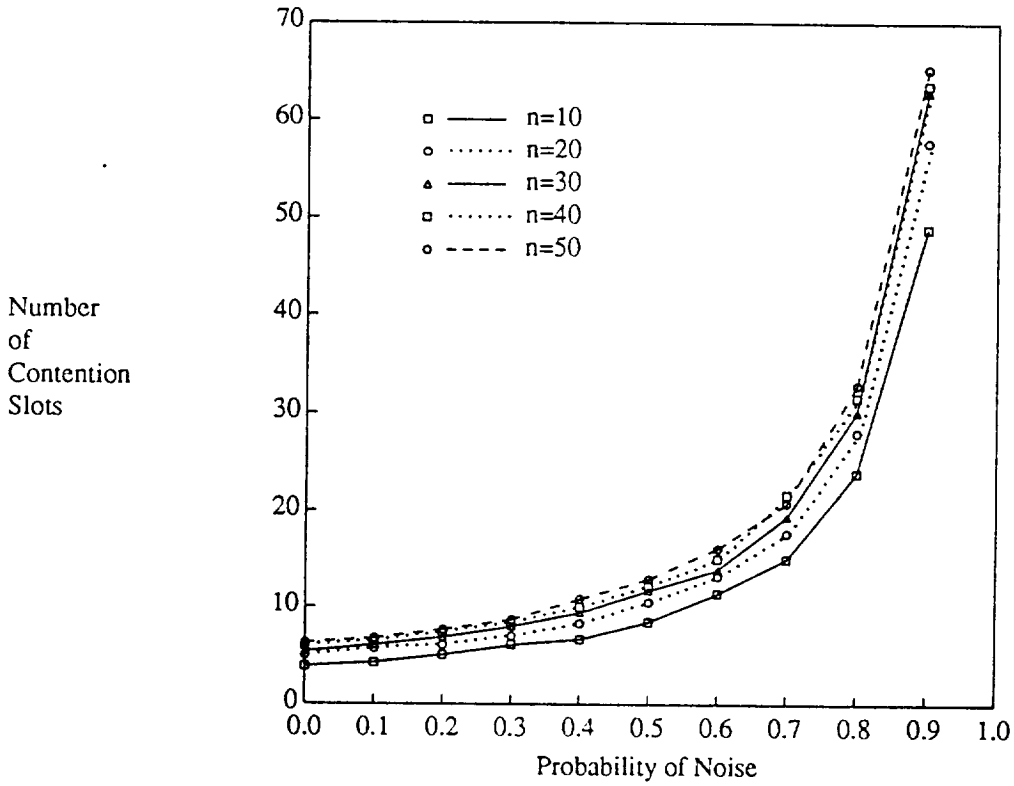


Figure 4.2. Binary-Divide Window-Control
(δ_2=1, n=10,20,30,40,50)

algorithm was terminated at δ_2, at which point the algorithm was restarted if a minimum had not been identified. The total resolution was therefore: $\delta_1 \cdot \delta_2 = \delta_2 / (10n)$.

Figure 4.3 shows the results of varying δ_2 for n=10 stations. (If δ_2=1, then the binary-divide phase is skipped altogether, and the algorithm is restarted when δ_1 is reached.) As δ_2 decreases, more iterations are wasted before the algorithm can be restarted after an error due to noise. For higher probabilities of noise, the probability of failure increases and the number of restarts increases. The net effect is that decreasing δ_2 causes an increase in the number of contention slots that becomes more pronounced as the probability of noise increases.

Figure 4.4 shows the various performance curves of n= 10, 20 and 30 when δ_2=1. Note that the number of contentions is relatively independent of n, unlike the binary exponential backoff and binary-divide window-control algorithms.

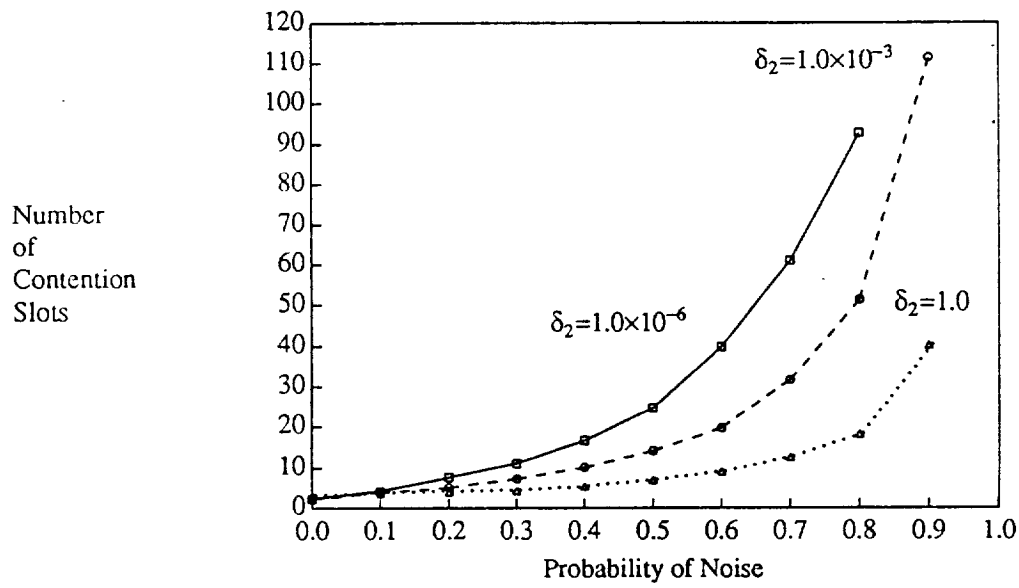


Figure 4.3. Original Dynamic Window Protocol
(n=10)

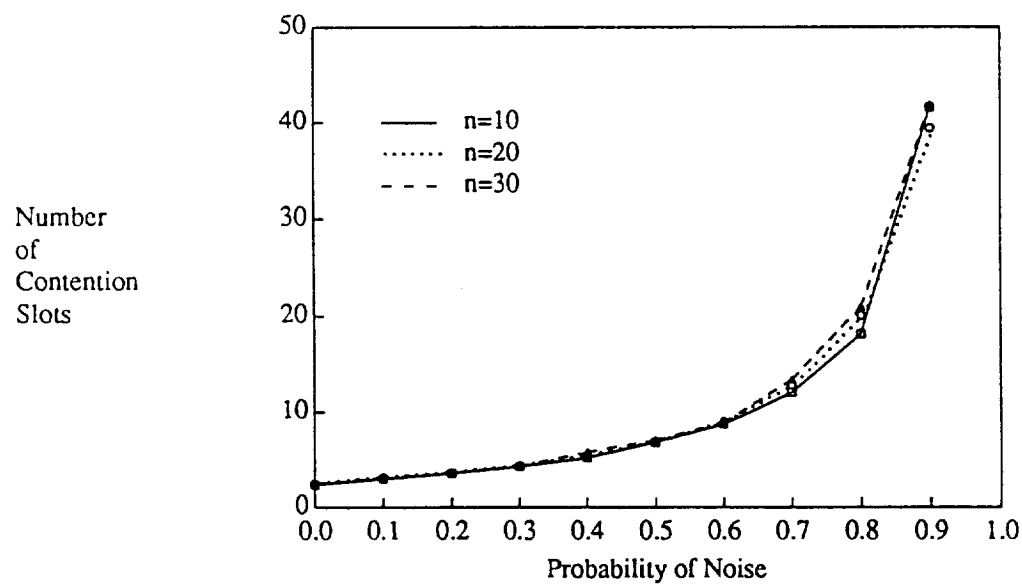


Figure 4.4. Original Dynamic Programming Algorithm
$\delta_1=(10n)^{-1}$, $\delta_2=1$

4.1.4. Two-Phase Window-Control Algorithm

The two-phase model is the same as the one outlined in Chapter 3. It used the same lookup table as the dynamic programming algorithm, was written in Fortran, and was simulated on a SUN 3/50.

Figure 4.5 illustrates the effects of varying δ_2 for the two-phase model when $n=10$. Note that for low probabilities of noise ($p \leq 0.3$), the performance is about the same as the original algorithm, but for high probabilities of noise the performance degrades much more rapidly than the original algorithm as δ_2 is increased. This is due to the fact that for low probabilities of noise, the two-phase algorithm has a higher probability of correcting itself when the contention parameters get outside the window. However, at higher probabilities of noise, the probability to correct this problem is much smaller, and it is more efficient to just restart the algorithm. The effects of noise on the two-phase protocol are much more extreme when δ_2 is small, as can be seen by the steeper performance curves.

Nevertheless, the performance of the two-phase algorithm is relatively independent of n, as illustrated in Figure 4.6. The algorithm is simulated for $\delta_2=1$ and n = 10, 20 and 30, and the performance is very similar to that of the original algorithm.

4.1.5. Approximate Recurrence Relation

Chapter 3 developed an approximate recurrence relation (Equation 3.17) to minimize the number of contention slots to resolve contention with the probability of noise added as a parameter. This algorithm was written in C and simulated on a SUN 3/260 with floating point accelerator. No binary phase was used, implying $\delta_2=1$ as in the other window-control algorithms.

The approximate recurrence relation estimates that the probability that b is a true upper bound is equal to the probability of noise in a contention slot, $1-p$. Let p_1 be the probability that b is a true upper bound. In Equation (3.7), $\Pr(Q_1) = p_1$ and $\Pr(Q_2) = (1-p_1)$. It turns out that $p_1 = 1-p$ is not a good approximation, since b may be a false upper bound due to previous false collisions. A much better heuristic approximation can be made by assuming a smaller probability that b is true, such as $p_1 = (1-p)^2$.

Table 4.1 contains a comparison of the theoretical and simulated performance of the approximate recurrence equation for both values of p_1. The theoretical performance is the expected number of contention slots to resolve contention as calculated by the approximate recurrence equation. During calculation, the optimal window for each

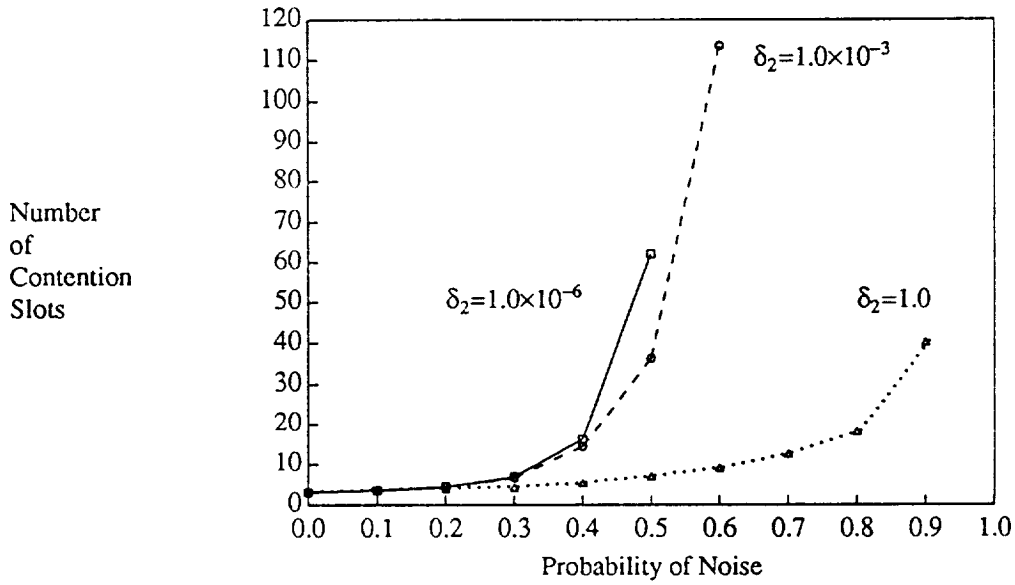


Figure 4.5. Two-Phase Algorithm
(n=10)

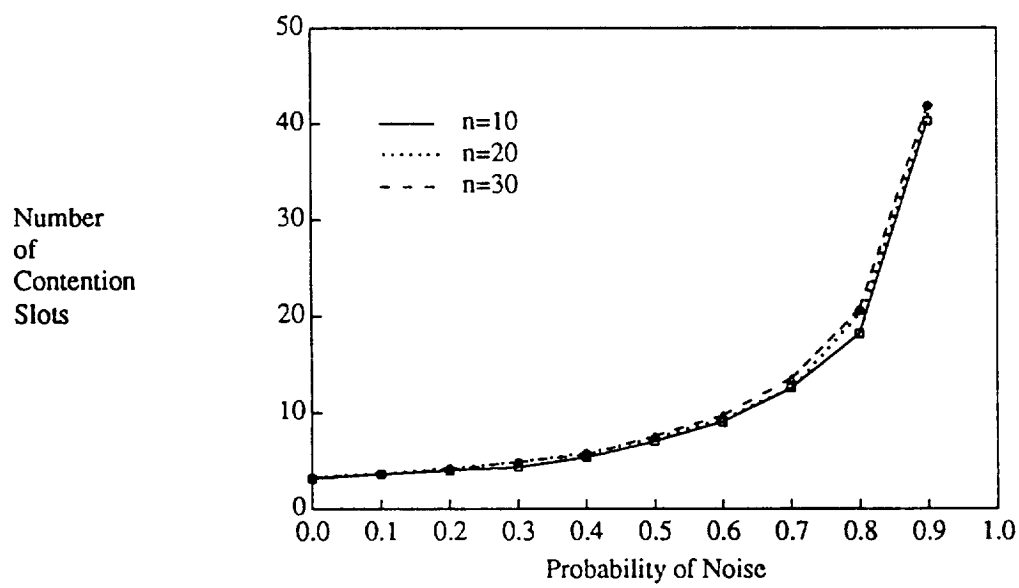


Figure 4.6. Two-Phase Algorithm
$\delta_1=(10n)^{-1}$, $\delta_2=1$

pair of lower and upper window boundaries was computed and stored in a table. The simulated performance is the actual performance of the network when using these windows. The correlation between the simulated and theoretical performance is dependent on the accuracy of the approximation. In Table 4.1, the simulated performance is much closer to the theoretical performance when p_1 is approximated by $(1-p)^2$. The overall performance is also improved by using $p_1 = (1-p)^2$.

It is possible to obtain even further improvements by developing an even more accurate approximation of p_1. Such an approximation should yield a better overall performance and simulated values that are even closer to the

Table 4.1. Comparison of the Performance of the Approximate Recurrence Equation for Various p_1's.

Dynamic Programming Algorithm (n=10)				
p = probability of noise during contention slot, p_1 = estimated probability that b is a true upper bound				
Probability	$p_1 = 1-p$		$p_1 = (1-p)^2$	
of Noise	Theory	Simulation	Theory	Simulation
0	2.5	2.4	2.5	2.4
0.1	2.6	2.9	2.7	2.8
0.2	2.8	3.4	2.9	3.2
0.3	3.0	4.0	3.3	4.0
0.4	3.3	4.9	3.9	4.8
0.5	3.7	6.0	4.8	6.0
0.6	4.5	8.0	6.2	7.3
0.7	5.8	11.4	8.6	9.6
0.8	8.4	16.7	13.5	15.1
0.9	16.5	36.5	28.5	28.5

Table 4.2. Comparison of the Performance of the Approximate Recurrence Equation for Various n's

Dynamic Programming Algorithm				
Probability	n=20		n=30	
of Noise	Theory	Simulation	Theory	Simulation
0	2.6	2.5	2.6	2.5
0.1	2.8	3.0	2.8	2.9
0.2	3.0	3.3	3.0	3.4
0.3	3.4	3.9	3.4	3.9
0.4	4.0	4.8	4.0	4.9
0.5	4.9	6.2	4.9	6.3
0.6	6.3	7.6	6.4	7.7
0.7	8.9	9.5	8.9	9.9
0.8	13.8	15.1	14.0	15.0
0.9	29.2	29.6	29.4	30.2

theoretical values. Such developments are left to future research. At any rate, both approximations perform better than the original dynamic programming algorithm, which does not account for noise.

The performance of the approximate recurrence relation is also relatively independent of the number of stations, similar to the other window-control algorithms. Table 4.2 compares the simulated expected number of contention slots of n=20,30 and $p_1 = (1-p)^2$, under various probabilities of noise; (n=10 is shown in Table 4.1).

There is one drawback to the approximate recurrence relation-- the time required to create the lookup table. Table 4.3 compares the time to create the lookup tables for the simulations. The long computation time prohibits simulation of higher values of n.

4.1.6. Comparison of Algorithms

Figure 4.7 contains a comparison of all the algorithms for n=20 and δ_2=1. For low levels of noise, all window-control algorithms perform better than the binary exponential backoff algorithm. For higher levels of noise, most window-control methods, except binary-divide, still perform better than the binary exponential backoff algorithm. All algorithms suffer from a logarithmic degradation in performance as the probability of noise increases.

Several interesting observations can be made by comparing the performance of the window-control algorithms. As expected, the pure binary-divide algorithm has the worst performance. However, the performances of the original dynamic programming algorithm and two-phase algorithm are almost identical when $\delta_2 = 1$. The approximate recurrence equation has the best performance for all levels of noise. The improvements gained are most pronounced at higher levels of noise. Therefore, it is worthwhile to include noise as a parameter in the formation of the recurrence equation.

Table 4.3. Time to Create Lookup Table for Simulation of Approximate Recurrence Equation

Time to Compute Lookup Table for Approximate Recurrence Equation (in minutes)			
n=10	n=20	n=30	n=40
9.9	81.0	280.4	456.8

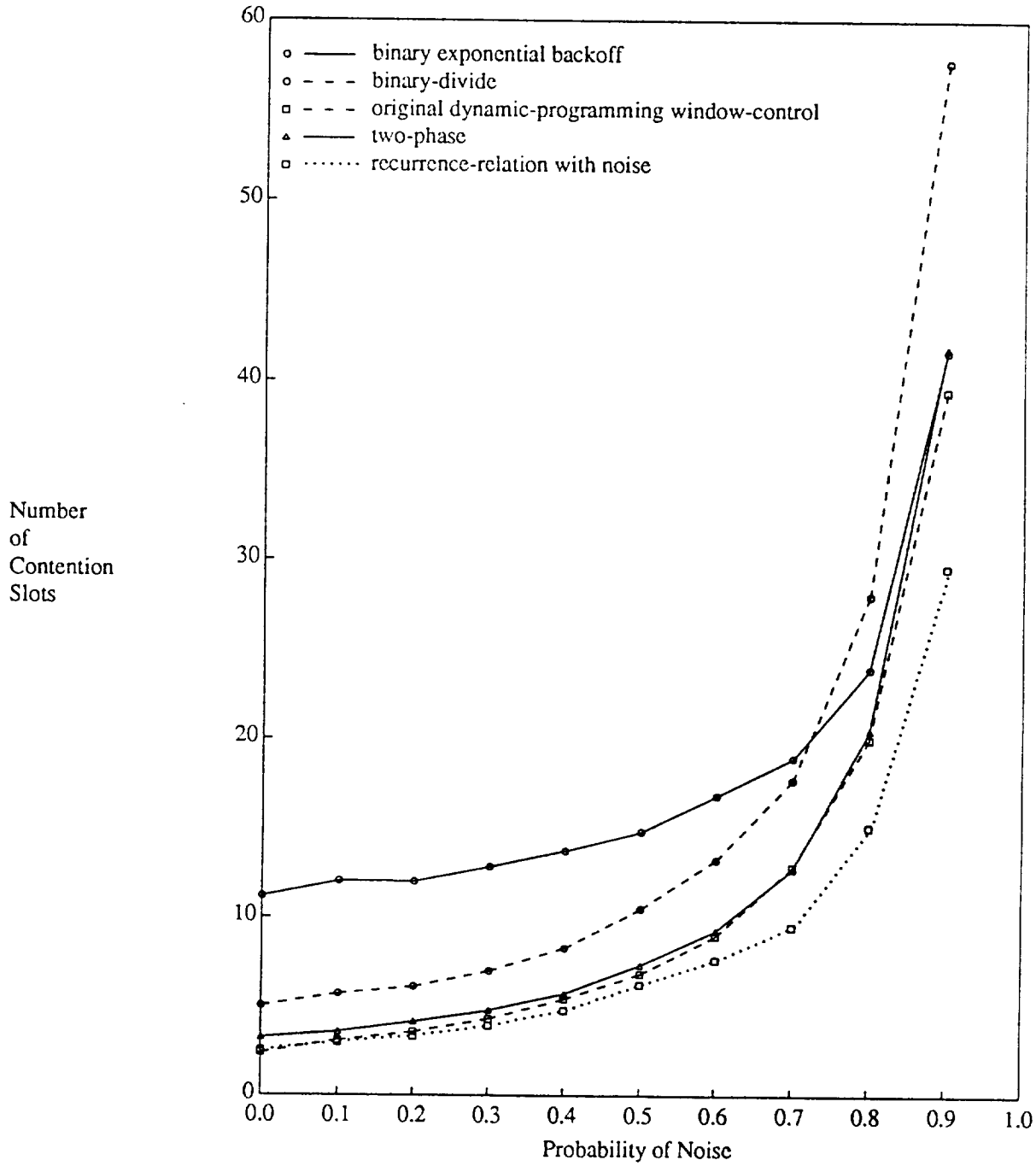


Figure 4.7. Comparison of Bus Contention Algorithms
(δ=1, n=20)

4.2. Asynchronized Windows

This section looks at the effects of noise when it causes individual stations to detect different states of the network. In the worst case, each station may have a noisy link to the network with a given probability of noise that is independent from that of other links. The simulation models the two-phase algorithm with $\delta_1 = 1/(10n)$ and δ_2=0.1. It compares the case where all stations detect all noise for n=10, to the case where all stations have independent but equal probabilities of detecting noise for n=10, 20 and 30.

In the simulation, restarts were allowed only upon the detection of an idle line, as outlined in Section 3.3.2. To keep track of the true minimum, contention parameters were not regenerated during a restart. To guarantee that each contention parameter differed by at least $\delta = \delta_2 \cdot \delta_1 = 1/(100n)$, the contention parameters were modified by including each station's unique identification number. The identification number was an integer, I, in the interval [0,n). Contention parameters were formed by selecting a random integer in the interval [0,100), and by adding to it I/n. The contention parameter was scaled to the range [0,1) by dividing the result by 100. Figure 4.8 illustrates the effects of noise on isolating a station (not necessarily the minimum), when the windows remain synchronized and when the windows become asynchronized.

It is interesting that the performance for the asynchronized windows is better than the performance for synchronized windows. This becomes more evident at higher levels of noise, but should not be surprising due to the operation of the two-phase protocol. If the proper minimum is not isolated due to noise, some other contention parameter may lie within its designated window and just happen to be the only one transmitting. Some stations may reduce their windows faster than others and be able to restart the algorithm sooner or some stations may lag behind and still happen to be within the window bounds. Either of these cases opens up greater possibilities of isolating a station more quickly when noise is significant.

The window-control algorithm can be used just as a means for resolving contention without finding the minimum. In such cases, it has the same function as the binary exponential backoff algorithm. Since the performance of the window-control algorithm for isolating stations is improved when the windows become out of phase, the same arguments hold concerning the superiority of the window-control algorithms to the binary exponential backoff algorithm.

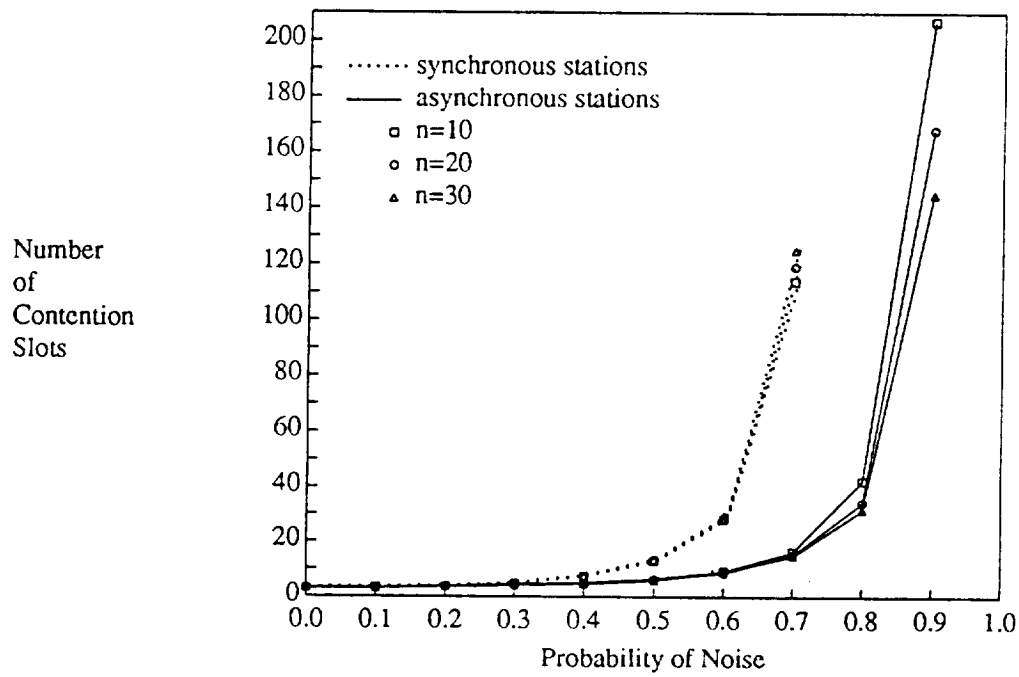


Figure 4.8: Two-Phase Algorithm with Independent Noise Detection
$\delta_1=(10n)^{-1}$, $\delta_2=0.1$

Table 4.4 shows the percent of true minimums identified at the termination of the algorithm. Note that the percent of true minimums decreases as n is increased. This decrease is more pronounced at higher probabilities of noise. To find the true minimum, the algorithm would have to be restarted after the broadcast phase with modified initial window boundaries, as outlined in Section 3.3.2. As a result, the number of contention slots to resolve contention would actually be greater than the number shown in Figure 4.8. One would also have to add in the time to complete the broadcast phase for each restart. Therefore, the time required to isolate the minimum may increase dramatically if the windows become out of phase.

One should note that in reality, it is very unlikely that all stations on the network will sense the network differently. Instead, it is more likely that one or two stations or groups of stations may detect noise differently than the others. As a result, the percent of true minimums in Table 4.4 and the performance curves of the independent stations in Figure 4.8 are lower bounds.

Table 4.4. Percent of True Minimums

Independent Stations			
Probability	% true minimums		
of Noise	n=10	n=20	n=30
0	100	100	100
0.1	94.1	93.9	93.6
0.2	85.2	86.4	86.1
0.3	80.1	81.9	77.7
0.4	72.6	70.9	70.3
0.5	68.5	64.9	62.9
0.6	61.0	57.7	54.6
0.7	55.6	49.7	45.6
0.8	51.5	43.3	39.4
0.9	50.1	41.1	35.6

4.3. Load Estimation

This section simulates the effects of noise on load estimation. The number of contending stations was estimated by a moving average of previous windows using Equations (2.12) and (2.13). These load estimating equations were used in the approximate greedy window-control outlined in Chapter 2. The approximate greedy window-control algorithm was chosen because it was the simplest of the load dependent window-control algorithms to simulate. The simulation assumed that there were 100 stations in the system. Since contention only occurred when there were two or more contending stations, the estimated value of n was constrained to the interval, [2,100]. The simulation was written in C and performed on a SUN 3/260. As in previous simulations, n was the number of contending stations, and $\delta_1 = 1/(10n)$.

Figure 4.9 illustrates the effects of noise on the approximate greedy window-control algorithm with load estimation. In all cases, the performance when the channel load is estimated is worse than when the load is exactly known. The performance is much worse for smaller values of n, especially when the probability of noise is high. This problem is due to the fact that false collisions may cause the minimum contention parameter to be isolated by a window with an upper bound smaller than that which is actually needed. The net result is that the network appears to be more heavily loaded than it actually is. Consequently, noise has a greater effect on load estimation when n is small.

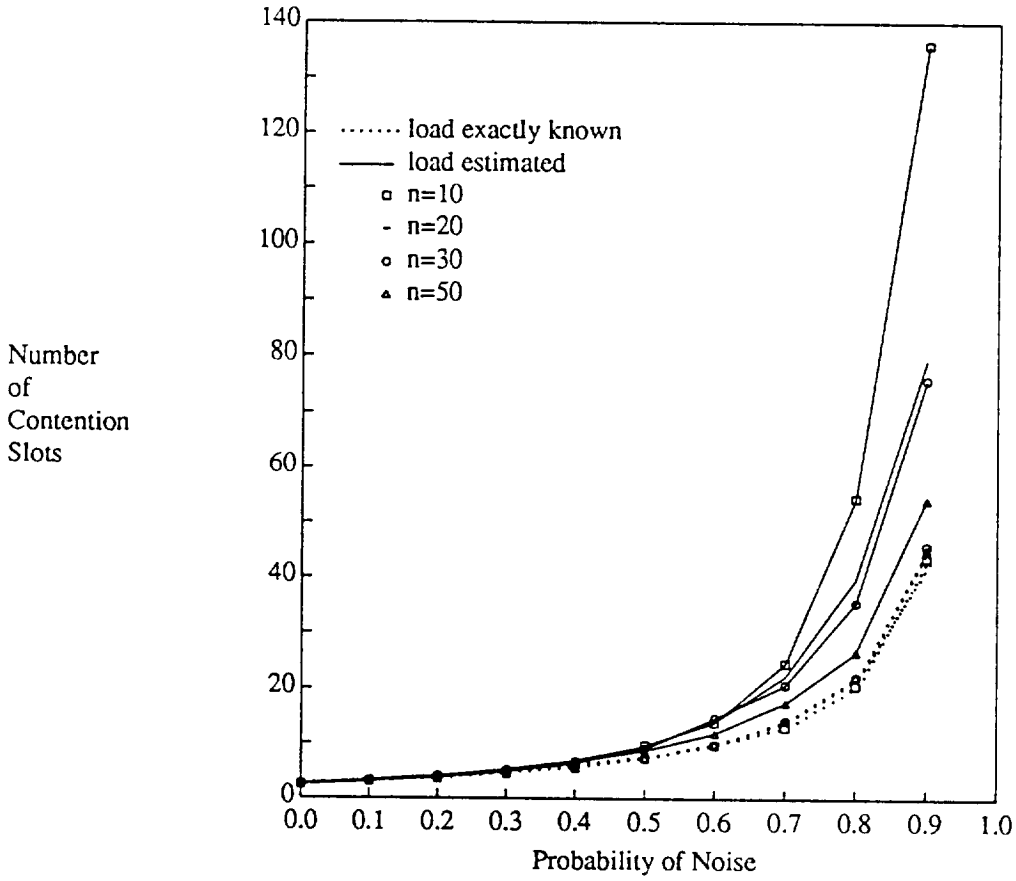


Figure 4.9. Approximate Greedy Window-Control with Load Estimation
(n=10,30,50, $\delta_1=(10n)^{-1}$, $\delta_2=1$)

One possibility to correct for this phenomenon is to include a scaling factor into the load estimation formula, $n_e(t)$, based on some function of the probability of noise. A heuristic approximation to this scaling factor based on the observation of $n_e(t)$ for various probabilities of noise is $(1-p)/0.8$. Figure 4.10 illustrates the performance when $n_e(t)$ is multiplied by $(1-p)/0.8$ for 10, 30 and 50 stations. Note that the performances of the estimated and known channel loads are much closer.

Better approximations to the scaling factor for load estimation can be made by a more detailed analysis of the problem. This is left to future research. Nevertheless, the approximation presented in this section shows that it is possible to correct for the errors caused by noise in load estimation.

Figure 4.11 illustrates the performance of various bus contention algorithms with load estimation for n=20. Modifications were made for the dynamic programming algorithm and approximate recurrence relation because of the large amount of time required to create the lookup tables. Consequently, tables were only created for n equal to multiples of five between five and fifty inclusive. The scaling factor in the load estimation equation was not used for the approximate recurrence relation, since better performance was obtained by omitting it. The scaling factor is not as critical to the performance of the approximate recurrence relation, since choice of the windows is dependent on noise, unlike the other window-control algorithms. As a result, the window used to isolate the minimum can be used to determine the load of the network more accurately in the approximate recurrence relation than in the other algorithms.

Note that the load dependent window-control algorithms have a worse performance than the performance obtained when the load is exactly known, which can be seen by comparing Figure 4.7 to Figure 4.11. Nevertheless, the performance of the window-control algorithms is still better than the exponential backoff algorithm for most levels of noise. Also, the performance of the approximate recurrence relation is better than the performance of the other bus contention algorithms in the presence of noise, even when the load is estimated.

4.4. Implementation of a Window-Control Algorithm on Ethernet in the Presence of Noise

Implementation of a window-control algorithm on a 10-Mbit/s Ethernet network is restricted by the stringent real-time requirement that each contention slot has a duration of less than 60 μs [16]. The only algorithm that can be computed quickly enough in real time to satisfy this time requirement is the binary-divide algorithm. The approximate greedy window-control algorithm may be used if a lookup table is employed to compute the square root [13]. To use any of the other window-control algorithms, the sequence of windows would have to be precomputed and stored in a lookup table.

Because of its superior performance for all probabilities of noise, the approximate recurrence relation ought to be used in implementations where noise is a significant factor. Chapter 3 discussed how the sequence of windows could be stored on an MCS 8396 using a four-level binary decision tree with each (n,p) pair requiring 0.03 Kbytes of memory. One practical arrangement is to let n range from 5 to 100 by 5's, and let p range from 0 to 0.9 by 0.1. The required memory would then be 6 Kbytes, which can fit on the MCS 8396. Interpolation could be used for

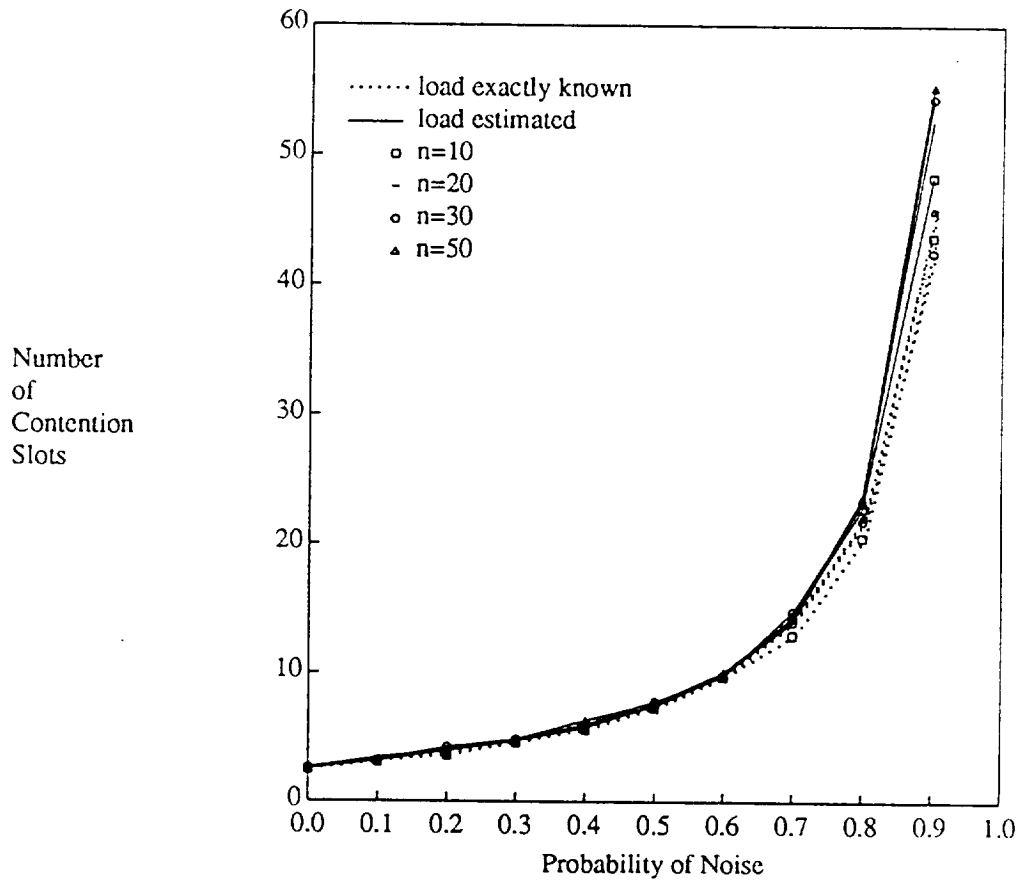


Figure 4.10. Approximate Greedy Window-Control with Improved Load Estimation (n=10,30,50, $\delta_1=(10n)^{-1}$, $\delta_2=1$)

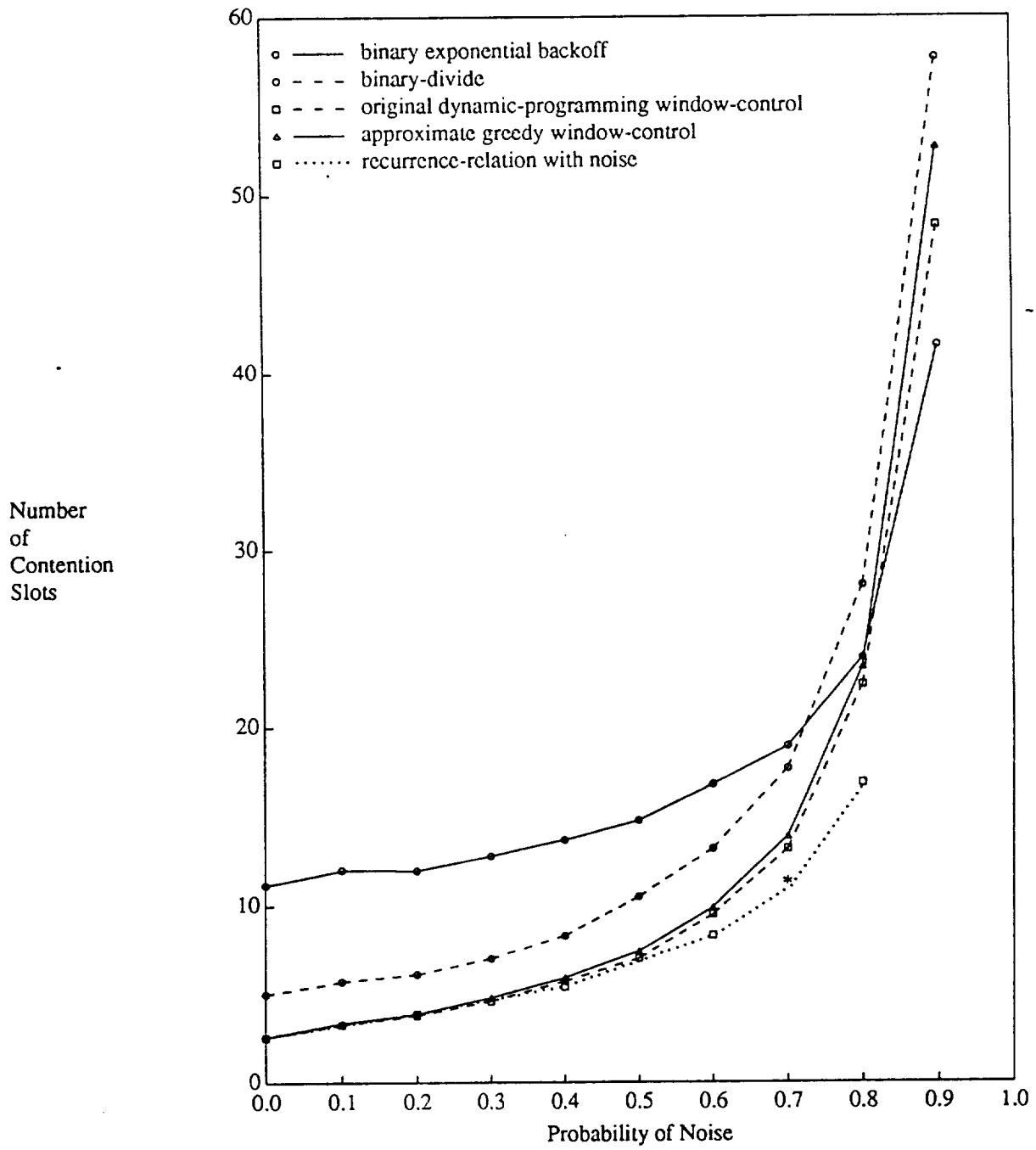


Figure 4.11. Comparison of Bus Contention Algorithms with Load Estimation (δ=1, n=20)

intermediate values of n and p. Nevertheless, the one drawback to this method is the enormous amount of time required to create the lookup tables.

In situations where the probability of noise is low, ($p \leq 0.3$), the original dynamic programming algorithm has a performance that is almost as good as the approximate recurrence relation. The advantage of the dynamic programming algorithm is that only one table is required for each n. Consequently, the intermediate values of n may also fit into the ROM of the MCS 8396. Furthermore, the computation time to create the tables in the original dynamic programming algorithm is about 1/4 the time required to create each table in the approximate recurrence relation with the same n. As a result, one might want to consider using the original dynamic programming algorithm if the probability of noise in the system is low most of the time.

In order to get the maximum benefit from the window-control algorithms, the channel load has to be estimated as accurately as possible, and the probability of noise has to be estimated for the approximate recurrence relation. The channel load can be estimated using the moving window technique with appropriate scaling to account for noise. The probability of noise can be estimated by the contending stations from the percentage of errors in transmitted messages. Coding theory techniques could be used to determine this value.

To guarantee that the window-control algorithm operates properly if the windows get out of phase, the modifications of Section 3.3.2 should be included. The major modifications are that a broadcast phase must be added to verify that the isolated station is the true minimum, and the algorithm should never be restarted unless the contending station detects that the line is idle. The broadcast phase can be omitted for use in bus contention where it is not necessary to isolate the minimum. Finally, the contending station should abort contention if it continues to sense collision on the network after it has reduced its window size to δ and the timeout period has expired.

5. CONCLUSIONS

The main result of this study is that by making a few adjustments to the window-control protocols, they can still be used to resolve bus contention when noise is present. One major adjustment is to include a termination condition, δ, for when the window size becomes too small, and to restart the algorithm with the original window when this occurs. Another adjustment is needed to account for the case where all stations do not detect the noise. In this case, the window boundaries may become out of phase, and the wrong station may be isolated. A broadcast phase must be added to verify that the proper station has been identified.

Another conclusion of this study is that the performance of most window-control algorithms in the presence of noise is superior to the performance of the binary exponential backoff algorithm. The gain in performance is greater at low probabilities of noise and high channel loads. Since normal operation of the network is likely to have low probabilities of noise, the window-control algorithms are preferable to the binary exponential backoff algorithm for resolving bus contention. Improvements to the dynamic programming algorithm outlined in Chapter 2 can be made by including noise as a parameter in a recurrence equation. This study developed an approximate recurrence relation that had superior performance to previously developed window-control algorithms in the presence of noise.

One drawback to the performance of the window-control algorithms is due to the effect of noise on load estimation. These effects are greater when the number of contending stations is small and the probability of noise is high, but can be corrected by adjusting the load estimation equation to account for noise.

Future research should investigate several of the issues discovered in this study. A good implementation of the broadcast phase should be realized with minimum added cost to resolve bus contention. Also, one should develop a better method for estimating the load of the system when noise is present. Additionally, one should investigate methods for estimating the probability of noise on the system for use in the approximate recurrence equation.

Previous discussion assumed that the stations were all synchronized initially and that they all began iterations of the window-control algorithms at the same time. In actuality, some stations may begin iterations before the others. Furthermore, their clocks might become skewed so that their contention slots overlap. Future work could be to investigate the problems of initializing the algorithms and synchronizing the clocks. In light of the study of asyn-

chronized windows, it is very likely that the performance of the window-control algorithms will continue to be better than that of the exponential backoff method.

APPENDIX A. SIMULATION PROGRAMS IN FORTRAN

```
c ************************* Fortran Programs ******************************
c * binary exponential backoff, dynamic programming window-control,
c * two-phase, and two-phase with independent noise detection algorithms
c ************************************************************************
c
c Each program consists of a declaration of variables, Lookup Table Generation
c (except binary exponential backoff), Simulation, and Data Analysis.
c The Lookup Table Generation and Data Analysis are exactly the same for
c all programs, and are therefore, listed only once in detail.
c
c
c Note:
c Operation of all Fortran programs requires that the output from a
c random number generator be piped into the standard input. The random
c generator function is written in C and the executable file is called
c 'randgen'.
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c          Dynamic Programming Window-Control
c
c 1. Compute optimal window size of the window protocol with continuous i.i.d.
c distributions :
c   - using dynamic programming formulation
c   - evaluation of dynamic programming is truncated when interval
c     is less than 1.0 / (5 * no. of contending stations)
c   - binary-divide window search is used after truncation point
c     in dynamic programming algorithm
c   - algorithm is restarted if minimum has not been identified
c     in binary-divide window search and window size is less than delta2
c
c 2. Evaluate the dynamic programming window search scheme by simulations
c   - assume that the no. of contending stations is known
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c variables :
c  n,nr : no. of contending stations
c  dw : size of automic interval
c  cn(400,400) : matrix to store intermediate window size and expected
c              no. of contentions and the search tree
c  tx(i) : contending parameter of station i
c  tb(i) : contention parameter for binary phase of station i
c  wb(30) : working space for histogram
c  rr: random number read in from standard input file
c  vflag: specifies whether or not simulation is due to a new set of
c        parameters or is just a continuation due to the 95% confidence
c        interval not being satisfied
c  ccc: confidence interval calculated from the variance and number of trials
c  pfc: random number used to determine the probability of noise during a
c      contention slot
c  icount: total number of trials for a given simulation
c  fctst: threshold of noise
c  delta2: resolution of window size for binary phase
c  la,lb: lower,upper bounds of window in dynamic programming phase
c  a,b: lower,upper bounds of window in binary-divide phase
```

```
c  w,lw: next upper bound of window
c
c
      double precision rr
      integer n,la,lb,lw
      real dw,tx(40),cn(400,400),tb(40)
      real a,b,w
      real nr,jj,ccc,delta2
      integer wb(30)
      common ns,nr,dw
      logical vflag
c
c file (cc.data) : store the evaluation results
c
      open(unit=1,file='cc.data',status='new')
      write(6,1200)
1200  format('+++++  window control using dynamic programming  +++++',
     .  /,'+++++  continuous iid (discrete approximation)  +++++')
      write(6,1210)
1210  format('+++++  (dw = 1.0 / r * no. of contending stations) +++++')
c
c
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c Lookup Table Generation:
c iterate on no. of contending stations assuming there is no noise
c (This portion of the program was developed by J. Y. Juang [17] )
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
2000  do 7700 ns=10,30,10
c
c initialize parameters
c
      nr=float(ns)
      do 7700 iter=10,10
      n=ns*iter
      dw=1.0/float(n)
c
c
      do 3010 i=1,n
      do 3010 j=i,n
3010  cn(i,j)=0.0
c
c i : lower bound of last window
c i+k : upper bound of last interval
c j : candidate of optimal window
c
c iterate on lower and upper bound of last window
c
      do 4400 k=1,n-1
      do 4400 i=1,n-k
      wmin=9999.0
c exhaustive enumeration on all possible candidates of optimal window
      do 4300 j=i,i+k-1
      call prob(i,j,i+k,pg,pl,pr)
      tw=pl*cn(i,j)+pr*cn(j+1,i+k)
      if(tw .ge. wmin) goto 4300
      kwin=j
```

```
      wmin=tw
4300  continue
      cn(i,i+k)=1.0+wmin
      cn(i+k,i)=float(kwin)
c     write(1,4350) i,j,kwin,cn(i,i+k)
c4350 format('lower bound : ',i3,' upper bound : ',i3,' window : ',
c    . i3,' exp. contentions : ',f7.5)
4400  continue
c
c save evaluation results
c
c     write(6,5100) ns,iter,cn(1,n)
c5100 format(//,'++++++ no. of contending stations : ',i4,/,
c    .'+++++ resolution factor r (n = r * no. of station) :',i4,/,
c    .        '>>>>> ave. no. of contentions (analysis) : ',f9.3)
c     write(6,5150)
c5150 format(//,'<<<<< matrix of optimal windows >>>>>')
c     do 5200 i=1,n
c     write(6,5300) i
c     do 5200 j=1,n/10
c     write(6,5400) (cn(i,k), k=10*j-9,10*j)
c5200 continue
c5300 format(/,'===== row ',i3,' =====')
c5400 format(10(f9.5,1x))
c
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c Simulation
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c format standard input file
133   format(d15.10)
c
c set window resolution for binary phase
      delta2=1
c
7900  do 7710 j=0,90,10
      jj=float(j)
      fctst=jj*0.01
      npkt=800
      vflag=.false.
8001  rewind 1
c
c initialize number of contention slots
c
8000  kstep=0
c
c generate contention parameters
c
8110  do 8100 i=1,ns
      read(5,133) rr
      tx(i)=rr
8100  continue
      la=1
      lb=n
8500  kstep=kstep+1
c
c generate noise
c
```

```
      read(5,133) rr
      pfc=rr
c
c contention resolutions
c
      if(lb .eq. la) goto 8700
c     if lb=la then go to the start of the binary phase, otherwise
c     retrieve the  discrete window value and convert it to integer
8590  w=cn(lb,la)
      lw=ifix(w)
c     convert to real window value
      w=dw*w
c
c collision detections
c
8601  ntrx=0
      do 8600 i=1,ns
      if(tx(i) .gt. w) goto 8600
      ntrx=ntrx+1
8600  continue
c
c collision?
      if (pfc .le. fctst .or. ntrx .gt. 1) then
        lb=lw
        goto 8500
c success?
      else
        if ( ntrx .eq. 1) goto 9000
      endif
c no transmission?
      la=lw+1
      goto 8500
c
c
c
c Binary Phase
c
8700  if (delta2 .eq. 1.0) then
      kstep=kstep-1
      goto 8110
            endif
c regenerate contention parameters for binary phase
      w=dw*float(lb)
      do 8720 i=1,ns
      read(5,133) rr
      if(tx(i) .gt. w) tb(i)=1.1
      if(tx(i) .le. w) tb(i)=rr
8720  continue
c
c contention resolution using binary-divide algorithm
8730  a=0.0
      b=1.0
8800  w=0.5*(a+b)
      z=w-a
c
c  restart evaluation if window has becomes too small.
c
      if (z .le. delta2) goto 8110
c
c collision detection
```

```
8621  ntrx=0
    do 8820 i=1,ns
    if(tb(i) .gt. w) goto 8820
    ntrx=ntrx+1
8820  continue
c
c collision ?
    if( pfc .le. fctst .or. ntrx .gt. 1) then
      b=w
      goto 8800
    endif
c
c success ?
    if (ntrx .eq. 1) goto 9000
c
c no transmission ?
    a=w
    goto 8800
c
c
c contention resolved. collect data
c
9000  write(1,9100) kstep
9100  format(i5)
    npkt=npkt-1
    if(npkt .gt. 0) goto 8000
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c Data Analysis
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c initialize sum, variance and histogram to 0 for new simulations
c (e. g. not continuations due to failure to satisfy requirements
c  for the 95% confidence interval)
c
    if (.not. vflag) then
      s=0.
      v=0.
      do 99 i=1,30
99      wb(i)=0
      icount=0
      endif
    rewind 1
c calculate mean and store number of contention slots in appropriate wb(i)
100  read(1,110,end=199)ncp
110  format(i5)
    s=s+float(ncp)
    icount=icount+1
      incp=ncp
c histogram has bins for number of contention slots greater than 24
      if (ncp .gt. 25 .and. ncp .le. 50) incp=25
      if (ncp .gt. 50 .and. ncp .le. 100) incp=26
      if (ncp .gt. 100 .and. ncp .le. 200) incp=27
      if (ncp .gt. 200 .and. ncp .le. 1000) incp=28
      if (ncp .gt. 1000 .and. ncp .le. 9999) incp=29
      if (ncp .gt. 9999) incp=30
    wb(incp)=wb(incp)+1
    goto 100
```

```
199  xmean=s/icount
c calculate variance
200  rewind 1
     vload=0
210  read(1,110,end=300)ncp
     v=v+(float(ncp)-xmean)**2
     goto 210
300  varian=v/icount
c calculate 95% confidence interval and run another 800 trials if necessary
     conf=1.96*sqrt(varian/icount)
     rewind 1
c
c distribution of no. of contentions
c
     write(6,500) icount
500  format('***** no. of packets transmitted : ',i5)
     ccc=0.05*xmean
     if (conf .gt. ccc) then
       npkt=800
       vflag=.true.
       rewind 1
       goto 8001
       endif
c print results
     write(6,400) xmean,varian,conf
400  format(/,'<<<<< ave. no. of contentions (simulation) : '
    . ,f9.4,/
    .     '<<<<< variance of contention period : ',f9.2,
    .'confidence=',f9.4)
     write(6,502)
502  format(//,'-- ncp --- frequency --- probability --- cum.--')
     cum=0.
     do 550 i=1,30
     k=i
     if (k .eq. 25) k=50
     if (k .eq. 26) k=100
     if (k .eq. 27) k=200
     if (k .eq. 28) k=1000
     if (k .eq. 29) k=4999
     if (k .eq. 30) k=5000
     pr=float(wb(i))/icount
     cum=cum+pr
     if (k .lt. 25) then
       write(6,560)k,wb(i),pr,cum
     else
       write(6,561)k,wb(i),pr,cum
     endif
550  continue
560  format(2x,i4,7x,i5,7x,f6.4,7x,f6.4)
561  format(x,'<',i4,7x,i5,7x,f6.4,7x,f6.4)
c
     write(6,570) j1
     write(6,580) j
570  format (//,'noise level= ',i5,'%')
580  format ('continued noise= 'i5,'%')
     tm=etime(at,bt)
     write(6,601) tm,at,bt
601  format ('elapsed time=',3(f10.2,3x))
7710 continue
7700 continue
```

```
      stop
      end
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c subroutine for computing probability of collision, success, and idle
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
      subroutine prob(n1,nw,n2,pg,pl,pr)
c
      real nr
      common ns,nr,dw
c
      pg=1.0
      pl=0.0
      pr=0.0
      if(n1 .ge. n2-1) return
c
      a=dw*float(n1-1)
      b=dw*float(n2)
      w=dw*float(nw)
c
      x=nr*(w-a)*((1.0-w)**(ns-1) - (1.0-b)**(ns-1))
      y=(1.0-a)**ns - (1.0-b)**ns - nr*(b-a)*(1.0-b)**(ns-1)
      z=(1.0-w)**ns - (1.0-b)**ns - nr*(b-w)*(1.0-b)**(ns-1)
      if (y .le. 0.0) return
      pg=x/y
      pr=z/y
      pl=1.0-pg-pr
      return
      end
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c end of Dynamic Programming Algorithm
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c
c
c
c
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c                Two-Phase Algorithm
c
c The two-phase algorithm is exactly the same as the original dynamic
c programming algorithm except for the method by which windows are
c updated after an idle contention slot is detected.
c After an idle contention slot, the lower bound of the window, a, is
c raised to the value, w; the upper bound of the window, b, is raised
c to the value of the upper bound from the previous contention slot.
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c New variables (variables from dynamic programming algorithm are retained):
c  kdeep= level of iterations within dynamic programming part of algorithm
c  bdeep= level of iterations within binary divide algorithm
c  kw(),bw()= arrays that store previous upper bounds for backtracking
c           within the dynamic prog. alg. and the binary alg.
c
      real kw(10),bw(20)
```

```
      integer kdeep,bdeep
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c include  Lookup Table Generation here
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c Simulation
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c generate contending parameters
c
      delta2=1.0
133   format(d15.10)
7900  do 7710 j=70,90,20
      vflag=.false.
      jj=float(j)
      fctst=jj*0.01
      npkt=800
8001  rewind 1
c
c contention resolutions
c
8000  kstep=0
8555  la=1
      lb=n
      kdeep=1
      kw(kdeep)=n
      w=cn(lb,la)
c
c generate contention parameters
c
8500  do 8100 i=1,ns
      read(5,133) rr
      tx(i)=rr
8100  continue
c
c generate noise
c
      read(5,133) rr
      pfc=rr
c
      kstep=kstep+1
      if(lb .eq. la) goto 8700
c     if lb=la, then go to the start of the binary phase
8601  ntrx=0
      lw=ifix(w)
c   convert to real window value
      w=dw*w
c
c collision detections
c
      do 8600 i=1,ns
      if(tx(i) .gt. w) goto 8600
      ntrx=ntrx+1
8600  continue
c
c collision?
      if (pfc .le. fctst .or. ntrx .gt. 1) then
        if (kdeep .eq. 30) goto 8500
```

```
      kdeep=kdeep+1
      kw(kdeep)=lw
      lb=lw
      w=cn(lb,la)
      goto 8500
c success?
    else
      if ( ntrx .eq. 1) then
      goto 9000
      endif
    endif
c no transmission?
c
c
    la=lw+1
    lb=kw(kdeep)
    w=lb
    if (kdeep .gt. 1) kdeep=kdeep-1
    if (lb .lt. la) lb=la
    goto 8500
c
c  Start of Binary Phase
c
8700  if (delta2 .eq. 1) then
            kstep=kstep-1
            goto 8555
            endif
c
c regenerating contending parameters
c
    w=dw*float(lb)
    bdeep=1
    bw(bdeep)=1.0
    do 8720 i=1,ns
    read(5,133) rr
    if(tx(i) .gt. w) tb(i)=1.1
    if(tx(i) .le. w) tb(i)=rr
8720  continue
c
c contention resolution using binary-divide algoirthm
8730  a=0.0
    b=1.0
    w=0.5*(a+b)
8800  z=w-a
c
c  restart evaluation if window has become too small.
c
    if (z .le. delta2) then
      lb=n
      la=1
      kdeep=1
      kw(kdeep)=n
      w=cn(la,lb)
      w=dw*w
      goto 8555
      endif
c
c collision detection
8621  ntrx=0
    do 8820 i=1,ns
```

```
      if(tb(i) .gt. w) goto 8820
      ntrx=ntrx+1
 8820 continue
 c
 c collision ?
      if( pfc .le. fctst .or. ntrx .gt. 1) then
        b=w
        bdeep=bdeep+1
        bw(bdeep)=b
        w=0.5*(a+b)
        goto 8800
      endif
 c
 c success ?
      if (ntrx .eq. 1) then
        goto 9000
        endif
 c
 c no transmission ?
 c
      if (bdeep .eq. 0) then
        la=la+1
        if (kdeep .gt. 1) kdeep=kdeep-1
        lb=kw(kdeep)
        if (lb .lt. la) lb=la
        goto 8500
      else
        a=w
        b=bw(bdeep)
        w=b
        bdeep=bdeep-1
      endif
      goto 8800
 c
 c contention resolved. collect data
 c
 9000 write(1,9100) kstep
 9100 format(i5)
      npkt=npkt-1
      if(npkt .gt. 0) goto 8000
 c
 cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
 c
 c include Data Analysis here
 c
 cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
 c
 c include subroutine for computing
 c     probability of collision, success, and idle here
 c
 cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
 c end of Two-Phase Algorithm
 cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
 c
 c
 c
 c
 c
 cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
 c         Two-Phase Algorithm with Independent Noise Detection
```

```
c
c  This program uses the same protocol as the two-phase algorithm
c  except that the noise is detected independently for each station.
c  Consequently, arrays were set up to keep track of all the necessary
c  information (such as window boundaries) for each station.
c  The resolution of the dynamic programming lookup table is 1/10n,
c  and the resolution of the binary window size in the binary-divide
c  phase is 1/10.  This leads to a net resolution of 1/100n.
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c  All variables are the same as those used in the two-phase algorithm,
c  except the following variables were converted to arrays:
c     la,w,lb: la(40),w(40),lb(40)
c     a,b: la2(40),lb2(40)
c     kdeep,bdeep: kkdeep(40),bbdeep(40)
c     kw(10),bw(20): kw(40,10),bw(40,20)
c     pfc: pfc(40)
c
c  The following new variables were added:
c     mintx: station with minimum contention parameter
c     newtx: station isolated by two-phase algorithm
c     truecnt: number of true minimums at end of simulation
c     wait(40): used to keep track of which stations are in wait state
c              Wait state is entered when the window size becomes less
c              than 1/100n.
c     binary(40): used to keep track of which stations are in binary-divide
c               phase
c     done: flag to denote that some station has transmitted successfully
c
      integer mintx,newtx,truecnt
      logical binary(40),wait(40),done
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c include Lookup Table Generation here
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c Simulation
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c generate contending parameters
c
133   format(d15.10)
7900  do 7710 j=0,90,10
      vflag=.false.
      jj=float(j)
      fctst=jj*0.01
      truecnt=0
      npkt=800
8001  rewind 1
c generate contention parameters for first set of stations
8000  mintx=1
      do 8100 i=1,ns
      read(5,133) rr
      n1=int(100*rr)
      n2=(i-1.0)/(nr+0.0)
      tx(i)=(n1+n2)/100
c set mintx to minimum of contending parameters
      if (tx(i) .lt. tx(mintx)) mintx=i
```

```
 8100  continue
 c initialize contention for first set of stations
      kstep=0
 6000  do 6003 i=1,ns
      kw(i,1)=n
      la(i)=1
      lb(i)=cn(n,1)
      kkdeep(i)=2
      kw(i,kkdeep(i))=lb(i)
      binary(i)=.false.
      wait(i)=.false.
      done=.false.
      la2(i)=0
      lb2(i)=1
 6003  continue
 c
 c
 6010  ntrx=0
 c  generate noise + transmissions
 c     include line 6011 here instead of where it is now for synchronous
 c              noise detection
      do 6013 i=1,ns
 6011  read(5,133) rr
      w(i)=dw*lb(i)
      if (.not. wait(i)) then
        if (binary(i)) then
          if ( tb(i) .le. lb2(i)) ntrx=ntrx+1
        else
          if ( tx(i) .le. w(i) ) ntrx=ntrx+1
        endif
      endif
      pfc(i)=rr
 6013  continue
 c
 c
 c analyze state of system for each station
 c and adjust window accordingly
 c
      do 6023 i=1,ns
      if (wait(i)) goto 6200
 c                                  goto start of wait state
      if ( lb(i) .le. la(i) ) goto 6100
 c                  goto start of binary phase
 c collision?
      if (pfc(i) .le. fctst .or. ntrx .gt. 1) then
        llb=lb(i)
        lla=la(i)
        w(i)=cn(llb,lla)
        lw=ifix(w(i))
        w(i)=dw*w(i)
        kkdeep(i)=kkdeep(i)+1
        kw(i,kkdeep(i))=lw
        lb(i)=lw
 6083     goto 6023
        endif
 c success?
      if (ntrx .eq. 1) then
        if ( tx(i) .le. w(i) ) then
            done=.true.
            newtx=i
```

```
      else
        tx(i)=1.1
      endif
      goto 6023
    endif
c no transmission?
      la(i)=lb(i)+1
      if (kkdeep(i) .gt. 1) kkdeep(i)=kkdeep(i)-1
      lb(i)=kw(i,kkdeep(i))
    goto 6023
c
c start of binary phase
c
6100  if (.not. binary (i)) then
c           initialize if necessary
      binary(i)=.true.
      bbdeep(i)=1
      lb2(i)=w(i)
      la2(i)=w(i)-dw
      bw(i,bbdeep(i))=lb2(i)
      if ( tx(i) .gt. w(i)) then
        tb(i)=1.1
      else
        tb(i)=tx(i)
      endif
    endif
c collision?
    if (pfc(i) .le. fctst .or. ntrx .gt. 1) then
      lb2(i)=0.5*(la2(i)+lb2(i))
      bbdeep(i)=bbdeep(i)+1
      bw(i,bbdeep(i))=lb2(i)
6183    goto 6500
      endif
c success?
    if (ntrx .eq. 1) then
      if ( tb(i) .le. lb2(i) ) then
          done=.true.
          newtx=i
      else
          tx(i)=1.1
      endif
      goto 6500
    endif
c no transmission?
      if (bbdeep(i) .eq. 1) then
        binary(i)=.false.
        la(i)=la(i)+1
        if (kkdeep(i) .gt. 1) kkdeep(i)=kkdeep(i)-1
        lb(i)=kw(i,kkdeep(i))
      else
        bbdeep(i)=bbdeep(i)-1
        la2(i)=lb2(i)
        lb2(i)=bw(i,bbdeep(i))
      endif
6500  if (( lb2(i)-la2(i)) .lt. 1.0/(200*ns)) then
c enter wait state if window is too small
      lb(i)=cn(n,1)
      la(i)=1
      binary(i)=.false.
          kkdeep(i)=2
```

```
        kw(i,kkdeep(i))=lb(i)
        wait(i)=.true.
        endif
      goto 6023
c
c start of wait state
c
6200  if (pfc(i) .gt. fctst .and. ntrx .eq. 1) then
            tx(i)=1.1
      else
            if (pfc(i) .gt. fctst .and. ntrx .eq. 0) wait(i)=.false.
      endif
c
6023  continue
      kstep=kstep+1
      if ( .not. done .and. kstep .le. 10000) goto 6010
      if (mintx .eq. newtx) truecnt=truecnt+1
9000  write(1,9100) kstep
9100  format(i5)
      npkt=npkt-1
      if (npkt .gt. 0) goto 8000
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c include Data Analysis here (plus the following lines)
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
      write(6,580) truecnt
580   format ('number of true minimums= ',i5)
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c include subroutine for computing
c     probability of collision, success, and idle here
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c  end of Two-Phase Algorithm with Independent Noise Detection
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c
c
c
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c                Binary Exponential Backoff Algorithm
c
c  This program simulates the exponential backoff algorithm of an ethernet
c  network in the presence of noise.
c  ns=#of stations
c  slot(i)=next slot that station i will attempt to transmit
c  doub(i)=number of transmission attempts station i has made
c        (new wait time= (rand#)* 2**doub(i) if i<=10
c                    or 1024 (2**10)    if i>10 )
c        (station i drops out of contention if i=16)
c  kstep=number of contention slots to resolve conflict
c  kcount=number of stations transmitting in a given contention slot
c  npkt=number of runs per simulation
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c
      double precision rr
```

```
      integer ns,npkt
      integer slot(60),doub(60)
      integer wb(30),wj,wj1
      logical vflag
c
c
c initialization simulation
c
133   format(d15.10)
      write(6,5110)
5110  format('Ethernet Exp Backoff Model')
      do 7720 ns=10,10
      write(6,5100) ns
5100  format(///,'number of contending stations:',i5)
      do 7700 j=0,90,10
      vflag=.false.
      fctst=float(j)/100.0
      npkt=800
8001  rewind 1
2000  kstep=0
c initialize noise for the first slot and the next slot that
c a station may transmit if it detects a collision
      read(5,133) rr
      do 1001 i=1,ns
        doub(i)=1
        read(5,133) rr
              slot(i)=2.0*rr+1
1001  continue
c
c count the number of transmissions in the current slot
c
1000  kcount=0
      kstep=kstep+1
      do 2001 i=1,ns
              if (slot(i) .eq. kstep) then
                kcount=kcount+1
            doub(i)=doub(i)+1
            if (doub(i) .gt. 16) then
              slot(i)=0
              goto 2500
            endif
c
c determine next slot that station will transmit if collision is detected
c
            read(5,133) rr
            if (doub(i) .le. 10) then
              slot(i)= 2**doub(i)*rr +1 +slot(i)
            else
              slot(i)= 1024*rr +1 +slot(i)
            endif
c             write (6,999) kstep,i,slot(i)
c999          format(i5,4x,i5,4x,i5)
          endif
2001  continue
2500  if (kstep .gt. 5000) goto 9000
c
c add noise
c
      read(5,133) rr
      pfc=rr
```

```
c If there is a collision (kcount>1 or noise) or no tranmission (kcount=0)
c then repeat above procedure for the next contention slot.
c If there is a successful transmission, terminate contention.
      if (pfc .le. fctst) kcount=2
      if (kcount .ne. 1) goto 1000
c
c     write(6,998) kstep
c998  format(//,'kstep=',i5)
c
9000  write(1,9100) kstep
9100  format(i5)
5000  npkt=npkt-1
      if (npkt .gt. 0) goto 2000
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c
c include Data Analysis here
c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c  end of Binary Exponential Backoff Algorithm
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
```

```
/*  This program generates a continuous stream of random numbers
 *  and is written in C. The corresponding executable file is
 *  called 'randgen'.  The output from randgen must be piped
 *  into the standard input of all Fortran programs in this Appendix.
 *  (The random number had to be written in C, because the standard
 *   random number generator in the Fortran library cycled after a
 *   certain point.) */

#include <stdio.h>

main()
{
  int i;
  double z;
  srand(1);
  for (;;)
    {
    z= rand()/2147483648.0;
    printf("%5.10f0,z);
    }
}  /* end of random number generator */
```

APPENDIX B. SIMULATION PROGRAMS IN C

```
/*************************************************************************
 *            C programs
 *
 *        includes the programs for the approximate recurrence relation,
 *        binary-divide algorithm, greedy approximate window-control,
 *        and load estimation
 *************************************************************************/

#include <stdio.h>
#include <math.h>
#include <sys/types.h>
#include <sys/times.h>

#define NMAX 799 /*number of trials in a given run */

/*************************************************************************
 * Approximate Recurrence Relation:
 *  -create a lookup table from the recurrence relation outlined in
 *   section 3.3.1.3
 *  -simulate contention using the lookup table
 *
 * The lookup table has a resolution of 1/(10n), and the algorithm
 * is restarted with new contention parameters if contention is not
 * resolved when the window size is less than 1/(10n).
 *************************************************************************/

/*************************************************************************
 * global variables:
 *
 *************************************************************************/
float Nx[401][401]; /* lookup table for computing N(a,b)
                For current window [a,b), Nx[a][b] contains the
                expected minimum number of contention slots to
                resolve contention. Nx[b][a] contains the
                optimal window for interval [a,b).
                Nx[b][a] is shared by both subroutines:
                N() and S(). */
float Sx[401][401]; /* lookup table for computing S(a,b)
                For current window [a,b), Sx[a][b] is the estimated
                probability of success and Sx[b][a]=Nx[b][a]. */
double p,       /* probability of noise */
    dw;       /* window resolution-- 1/(10n) */
int n;        /* number of initial contending stations */


/*************************************************************************
 * subroutines
 *************************************************************************/

/* prob(), return random # Uniform[0,1)
 */
   double
   prob()
   {
     static double divis = 2147483648.0;
```

```
        double y;

        y=rand();
        return (y/divis);
    }


 double
 g2(a,w,b) /* compute g in dynamic programming formulation */
 double w,a,b;
 {
  double pow(),x,y,z1,z2,z,p1;
  /* compute g2 assuming false collision-- b invalid */
  p1=(1.0-p)*(1.0-p);
  x=n*(w-a)*pow((1.0-w),(n-1.0))*(1.0-p);
  if (a < 1.0)
    {
    y=pow((1.0-a),(n-0.0));
    z1=x/y;
    }
  else
    z1=0.0;
 /* compute g2 assuming true collision -- b valid */
  x=n*(w-a)*(pow((1.0-w),(n-1.0))-pow((1.0-b),(n-1.0)));
  if (a < 1.0)
    {
    y=pow((1.0-a),(n-0.0))-n*(b-a)*pow((1.0-b),(n-1.0))-pow((1.0-b),(n-0.0));
    z2=x/y;
    }
  else
    z2=0.0;
 /* compute z */
  z=(1.0-p1)*z1 + p1*z2;
  return(z);
}


double
r2(a,w,b) /* compute r in dynamic programming formulation */
double w,a,b;
{
 double pow(),x,y,z,z1,z2,p1;
 /* compute r2 assuming false collision-- b invalid */
 p1=(1.0-p)*(1.0-p);
 if (a < 1.0)
   {
   x=(1.0-w)/(1.0-a);
   y=n+0.0;
   z1=pow(x,y)*(1.0-p);
   }
 else
   z1=0.0;
/* compute r2 assuming true collision-- b valid */
 if (a < 1.0)
   {
   x=pow((1.0-w),(n-0.0))-n*(b-w)*pow((1.0-b),(n-1.0))-pow((1.0-b),(n-0.0));
   y=pow((1.0-a),(n-0.0))-n*(b-a)*pow((1.0-b),(n-1.0))-pow((1.0-b),(n-0.0));
   z2=x/y;
   }
 else
```

```
  z2=0.0;
 /* compute z */
  z=(1.0-p1)*z1 + p1*z2;
  return(z);
}


double
N(a,b) /* Compute the estimated minimum number of contention slots to resolve
        contention given the current window [a,b). N() is dependent
        on minimizing the term N()/S() for smaller subwindows.
        The terms N() and S() are computed recursively until (b-a)<1,
        at which point the following boundary conditions are used:
          N(a,b)=1, S(a,b)=g(a,b).
      */
 int a,b; /* a = lower bound;  b = upper bound */
 {
   double N(),S(),g2(),r2();
   double aa,ww,bb,wtest,ntest,stest,nmin,zmin,smax,g,r,l;
   int w,wmin;

   wmin=a;
   if (Nx[a][b] > 0.0) /* do not recompute Nx[a][b] if already in table*/
            {
            nmin=Nx[a][b];
            }
   else if (b <= (a+1) )
            {
            if (b == (a+1) )
              {
              Nx[a][b]=1.0;
              Nx[b][a]=a;
              }
            nmin=1.0;
            }
   else
            {
            aa=dw*(a-1.0);
       bb=dw*(b-0.0);
       zmin=9999.0;
            for (w=a; w<b; w++)
              {
              ww=dw*w;
              g=g2(aa,ww,bb);
              r=r2(aa,ww,bb);
              l=1.0-g-r;
        /* recursion equation for N(a,b) */
              ntest=(1.0+l*N(a,w)+r*N((w+1),b));
         Nx[b][a]=w;
        /* calculate S(a,b) using same w as N(a,b) */
         stest=S(a,b);
         if (stest == 0) /* to prevent possible divide by zero in wtest */
           stest=0.0000001;
        /* calculate N()/S(); store minimums in appropriate tables */
         wtest=ntest/stest;
         if (wtest < zmin)
           {
                zmin=wtest;
                nmin=ntest;
           smax=stest;
```

```
        wmin=w;
        }
            }
          Nx[a][b]=nmin;
          Nx[b][a]=wmin;
      Sx[a][b]=smax;
          }
  return(nmin);
}


double
S(a,b) /* Compute the estimated probability of success given the current
       window [a,b). This value is used in N(a,b) to calculate the
       minimum N(a,b)/S(a,b)  */
int a,b; /* a = lower bound; b= upper bound */
{
  double S(),g2(),r2();
  double s,aa,ww,bb,zz;
  double g,r;
  int w,wmin;

  aa=dw*(a-1.0);
  bb=dw*b;
  if(Sx[a][b] > -1.0 ) /* do not recompute Sx[a][b] */
   {
   zz=Sx[a][b];
   }
  else if (b <= (a+1))
   {
   zz=g2(aa,bb,bb);
   Sx[a][b]=zz;
   if (b !=a)
    Sx[b][a]=a;
   }
  else
   {
   w=Nx[b][a]; /* choose w equal to same value chosen by N(a,b) */
   ww=dw*w;
   g=g2(aa,ww,bb);
   r=r2(aa,ww,bb);
    /* recursion relation for S(a,b) */
   zz=(g+(1.0-g-r)*S(a,w)+r*S((w+1),b));
   }
  return(zz);
}


/*****************************************************************
 * main program for Approximate Recurrence Relation
 *****************************************************************/
main()
{
 /*** The following variables have the same definitions as in    ***
  *** the original dynamic programming algorithm              ***/
   int i,j,k,la,lb,lw,ns,pp,wb[41];
   int npkt,kstep,ntrx,icount,ncp,done,vflag;
```

```
  double w,sum,v,varian,xmean,conf;
  float pfc,ccc,tx[41];
/****************************/
 int cc[1000], /* array storing number of contention slots to resolve
            contention for each iteration of the algorithm */
      cflag; /* flag to determine whether confidence interval is
            satisfied */
 float n1,s1,z1; /* temporary values for calculation of N(1,ns)/S(1,ns)=
              theoretical number of contention slots to resolve
              contention */
 double z,
      N(),S(); /* procedures as defined above */
 struct tms buffer; /* used to determine time required to simulate */
 long int tm,tm1;  /*   algorithm and to create lookup tables   */

 srand(1);
 for(pp=20; pp<=20; pp+=10)
 {
/* formulate dynamic programming table */
  /* initialize matrices */
  for (i=1; i<=300; i++)
  for (j=1; j<=300; j++)
    {
    Nx[i][j]=0.0;
    Sx[i][j]= -1.0;
    }

/* iterate on number of stations */
  n=40;
  ns=10*n;
  dw=0.1/(n+0.0);
  p=(pp+0.0)/100.0;
  la=1;
  lb=ns;
  /* N() creates table */
  z=N(la,lb);
  n1=Nx[la][lb];
  s1=Sx[la][lb];
  printf("number of stations= %d0,n);
  printf("number in theory(N),(S)=");
  printf("%4.5f %4.5f0,n1,s1);
  printf("total number of slots= %4.5f0,n1/s1);
  printf("user time, system time0);
  buffer.tms_utime=times(&buffer);
  buffer.tms_stime=times(&buffer);
  tm1=buffer.tms_stime;
  tm=buffer.tms_utime;
  printf("%ld---%ld0,tm,tm1);
  fflush(stdout);

 /*****************************************************************
  * simulate results
  *
  *
  *****************************************************************/
   cflag=1;
   vflag=1;
   while (cflag == 1) /* repeat NMAX trials until 95% confidence is
                established */
   {
```

```
    for (npkt=0; npkt<=NMAX; npkt++)
    {
     kstep=0;
     done=1;
     while (done == 1) /* repeat contention until minimum has been isolated */
     {
     la=1;
     lb=ns;
     for (k=1; k<=n; k++)
      tx[k]=prob();
      while (done == 1 & lb > la)
      {
      lw=Nx[lb][la];
      w=dw*lw;

      /* collision detections */
      for (k=1,ntrx=0; k<=n; k++)
       if (tx[k] <= w)
        ntrx++;

     /* generate noise */
     pfc=prob();

     /* determine state of network */
     if (pfc <= p || ntrx > 1)  /*collision*/
            lb=lw;
     else if(ntrx == 1)       /*success*/
            done=0;
     else                     /* no transmission */
            la=lw+1;
     kstep++;
     }
    }
   /* contention resolved, store result */
   cc[npkt]=kstep;
  } /* end npkt loop */
/*****************************************************************
 * data analysis
 *
 *
 *****************************************************************/
  if (vflag == 1)
   { /* initialize variables if new est of iterations, otherwise
       add values in cc[i] to previous set(s) */
   sum=0.0;
   v=0.0;
   vflag=0;
   for (k=1; k<=30; k++)
    wb[k]=0;
   icount=0;
   }
  for (npkt=0; npkt<=NMAX; npkt++)
   { /* calculte mean and place ncp into appropriate bin for histogram */
   ncp=cc[npkt];
   icount++;
   sum+=(ncp+0.0);
   if (ncp > 4999)
    ncp=30;
   else if (ncp > 1000)
    ncp=29;
```

```
 else if (ncp > 200)
  ncp=28;
 else if (ncp > 100)
  ncp=27;
 else if (ncp > 50)
  ncp=26;
 else if (ncp > 25)
  ncp=25;
 wb[ncp]++;
 xmean=sum/icount;
 }
for (npkt=0; npkt<=NMAX; npkt++)
 { /* calculate variance */
 ncp=cc[npkt];
 v+=(ncp-xmean)*(ncp-xmean);
 }
varian=v/icount;
conf=1.96*sqrt((double)(varian/icount));
printf("number of packets transmitted= %d0,icount);
printf("avg. no. of contentions (sim)= %9.3f0,xmean);
printf("variance,confidence= %10.3f %9.3f0,varian,conf);
fflush(stdout);
/* if confidence is not great enough, do more iterations */
ccc=0.05*xmean;
if (conf < ccc )
  cflag=0;
} /* end while(!cflag) loop */

/* print distribution */
printf("noise(percent)= %d0,pp);
printf("---ncp--- ---frequency---0);
for (k=1; k<=30; k++)
{
printf("%d",k);
printf("        ");
printf("%d0,wb[k]);
}
printf("user time, system time0);
buffer.tms_utime=times(&buffer);
buffer.tms_stime=times(&buffer);
tm1=buffer.tms_stime/3600;
tm=buffer.tms_utime/3600;
printf("%ld---%ld0,tm,tm1);
fflush(stdout);
}
}
/*********************************************************************
* end of main program for Approximate Recurrence Relation
*********************************************************************/
.sp 5
/*********************************************************************
* This program models both the greedy approximate window-control
* and binary-divide algorithms with or without load estimation.
* The appropriate sections are outlined be /* ... */ and can
* be substituted in, depending on which simulation is desired.
*
* As in all the other programs, delta = 1/(10n), and the algorithm
* is restarted if the window size becomes smaller than delta.
*********************************************************************/
#include <stdio.h>
```

```
#include <math.h>
#include <sys/types.h>
#include <sys/times.h>

#define NMAX 799 /*number of trials in a given run */
/* prob(), return random # Uniform[0,1)
 */
   double
   prob()
   {
      static double divis = 2147483648.0;
      double y;

      y=rand();
      return (y/divis);
   }

/*****************************************************************/
main()
{
  int i,j,k,pp,cc[1000],wb[41];
  int N;
  int npkt,kstep,ntrx,icount,ncp,done,vflag,cflag;
  double p,dw,n,sum,v,varian,xmean,conf;
  float pfc,ccc,tx[51];
  struct tms buffer;
  long int tm,tm1;
  double a,b,w,
      wmv,  /* wmv = moving average of previous windows used to
                 estimate n  */
      C,D;  /* temporary values used in the calculation of the
             approximate greedy window-control algorithm */

 srand(1);
 for(pp=0; pp<=90; pp+=10)
 {

/* iterate on number of stations */
  N=30;
  n=N+0.0;
  wmv=1.0/n;  /* initialize moving window average to 1/n */
  dw=0.1/(n+0.0);
  p=(pp+0.0)/100.0;
  printf("0umber of stations= %d0,N);

/*****************************************************************
 * simulate results
 *
 *
*****************************************************************/
 cflag=1;
 vflag=1;
 while (cflag == 1)
 {
 for (npkt=0; npkt<=NMAX; npkt++)
 {
 kstep=0;
 done=1;
 while (done == 1)
 {
```

```
a=0.0;
b=1.0;
for (k=1; k<=N; k++)
 {
 tx[k]=prob();
 }
 while (done == 1 & (b-a)>dw)
 {
 /* compute new window */
   /*****************************************************
    * use the following calculations for the approximate
    * greedy window-control algorithm:
    *   C=((n-1)*(a+b)+2) / n;
    *   D=(a+b + (n-2)*a*b) / n;
    *   w=(C-sqrt(C*C-4*D))/2;
    *****************************************************
    * or use this calculation for binary-divide:
    *   w=(a+b)/2.0;
    *****************************************************/

  /* collision detections */
  for (k=1,ntrx=0; k<=N; k++)
   if (tx[k] <= w)
    ntrx++;

  /* generate noise */
  pfc=prob();

  /* determine state of network */
  if (pfc <= p || ntrx > 1)  /*collision*/
    {
        b=w;
        }
  else if(ntrx == 1) /*success*/
        done=0;
  else /* no transmission */
        {
    a=w;
        }
   kstep++;
   }
  if (kstep > 5000)
   break;
  }
  /* contention resolved: store result,                    */
  /* compute moving window average and update estimation of n */
  cc[npkt]=kstep;
  /*****************************************************
   * add these calculations if using load estimation:
   *     wmv=(wmv+w)/2;
   *     n=1/(log(1/(1-wmv)));
   *****************************************************/
  } /* end npkt loop */
/*****************************************************************
 *
 * include data analysis here (exactly the same as the last program)
 *
 *****************************************************************/
/* end of program
 *****************************************************************/
```

REFERENCES

[1] W. Chu, L. J. Holloway, M. T. Lan, and K. Efe, "Task allocation in distributed data processing," *IEEE Computer*, vol. 13, pp. 57-69, Nov. 1980.

[2] G. Fielland and D. Rogers, "32-bit computer system shares load equally among up to 12 processors," *Electronic Design*, vol. 32, pp. 153-168, Sept. 6, 1984.

[3] D. D. Gajski and J. K. Peir, "Essential issues in multiprocessor systems," *IEEE Computer*, vol. 18, pp. 9-27, June 1985.

[4] Y. C. Chow and W. Kohler, "Models for dynamic load balancing in a heterogeneous multiple processor system," *IEEE Transactions on Computers*, vol. C-28, pp. 334-361, May 1979.

[5] Y. T. Wang and R. J. T. Morris, "Load sharing in distributed systems," *IEEE Transactions on Computers*, vol. C-34, pp. 204-217, March 1985.

[6] L. M. Ni and K. Hwang, "Optimal load balancing strategies for a multiple processor system," *Proceedings: 1981 International Conference on Parallel Processing*, pp. 352-357, 1981.

[7] V. Lo and J. W. S. Liu, "Task assignment in distributed multiprocessor systems," *Proceedings: 1981 International Conference on Parallel Processing*, pp. 358-360, 1981.

[8] B. W. Wah and A. Hicks, "Distributed scheduling of resources on interconnection networks," *Proceedings: 1982 National Computer Conference, AFIPS Press*, pp. 697-709, 1982.

[9] J. Y. Juang and B. W. Wah, "Optimal scheduling algorithms for multistage resource sharing interconnection networks," *1984 IEEE Computer Software and Applications Conference*, pp. 217-224, 1984.

[10] F. A. Briggs, K. S. Fu, K. Hwang, and B. W. Wah, "PUMPS architecture for pattern analysis and image database management," *IEEE Transactions on Computers*, vol. C-31, pp. 969-983, Oct. 1982.

[11] J. Y. Juang and B. W. Wah, "Unified window protocols for contention resolution in local multi-access networks," *3rd Annual Conference of the IEEE Computer and Communications Societies*, 1984.

[12] A. S. Tanenbaum, *Computer Networks*. New Jersey: Prentice Hall Inc., 1981.

[13] B. W. Wah and J. Y. Juang, "Resource scheduling for local computer systems with a multiaccess network," *IEEE Transactions on Computers*, vol. C-34, pp. 1144-1156, Dec. 1985.

[14] J. Y. Juang and B. W. Wah, "Channel allocation in multiple contention bus networks," *IEEE INFOCOM Conference*, 1986.

[15] W. Stallings, *Data and Computer Communications*. New York, New York: Macmillan Publishing Co., 1985.

[16] Digital Equipment Corp., Intel Corp., and Xerox Corp., *The Ethernet: Local Area Network Data-Link Layer and Physical Specifications*. Version 1.0, Sept. 30, 1980.

[17] J. Y. Juang, *Resource Allocation in Computer Networks*. Ph. D. Dissertation, Purdue University, Aug. 1985.

[18] G. B. Thomas and R. L. Finney, *Calculus and Analytic Geometry*. Reading, Massachusetts: Addison-Wesley Publishing Co., 1980.

[19] W. Mendenhall and R. L. Scheaffer, *Mathematical Statistics with Applications*. North Scituate, Massachusetts: Duxbury Press, 1973.

To Appear: Proceedings, Princeton Workshop on Algorithms, Architecture, and Technology Issues in Concurrent Computing, October 1987.

N94-70424

APPLICATIVE ARCHITECTURES FOR FAULT-TOLERANT MULTIPROCESSORS

Madhumitra Sharma
Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign

and

W. Kent Fuchs
Computer Systems Group
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign

Abstract: This paper proposes functional programming frameworks for the design of highly reliable multiprocessor systems. In contrast to imperative programming environments, a functional environment offers elegant, relatively simple, and efficient solutions to concurrent error detection and recovery problems in multiprocessors. Specific fault tolerance mechanisms for upset exposure, fault containment, secure task assignment, and recovery are developed for a class of applicative multiprocessor architectures. Verification of abstract behavioral characteristics of applicative tasks is used for exposing faults during the execution of tasks. The fault containment mechanism is based on isolation of stack and heap segments of tasks. A protocol for secure task assignment is defined between system components. The architecture permits incremental, distributed, and asynchronous backups of system state. Finally, recovery is accomplished, even in the worst cases, by re-execution of a small number of tasks.

Acknowledgment: This research was supported in part by the National Aeronautics and Space Administration (NASA) under Contract NASA NAG 1-613 in cooperation with the Illinois Computer Laboratory for Aerospace Systems and Software (ICLASS), by the Joint Services Electronics Program (U.S. Army, U.S. Navy, and the U.S. Air Force) under Contract N00014-84-C-0149, by the National Science Foundation under Grant No. US NSF DCR84-10110, by the U. S. Department of Energy under Grant No. US DOE-DE-FG02-85ER25001, and by the IBM Donation.

1. INTRODUCTION

Functional programming frameworks are proposed in this paper for the design of highly reliable multiprocessor systems. In contrast to imperative programming environments, a functional environment offers elegant, relatively simple, and efficient solutions to concurrent error detection and recovery problems in multiprocessors. Functional languages are also known to offer features appropriate for development of large parallel software systems including elegant data and procedure abstraction techniques, easily identifiable parallelism, and the relative ease of proving program correctness [Vegd84]. Together, these characteristics make them viable candidates for designing highly reliable multiprocessors. This is illustrated in this paper through the development of mechanisms for efficient and inexpensive concurrent error detection and recovery in a class of applicative multiprocessor architectures.

A general approach to overall system structuring for fault tolerance has been described as being based on *(1) the use of idealized fault tolerant components, (2) recursive structuring, and (3) atomic actions* [Rand84]. Such overall system structuring helps in reducing the complexity of fault tolerance mechanisms and thus leads to higher reliability. The applicative architecture presented herein is well structured with respect to the above criteria. It employs abstract components defined such that each component can appropriately handle faults within itself and those in others it interacts with. Thus, they qualify as *idealized fault tolerant components*. Further, the architecture is *recursively structured* since units within the system are identical and the functionality of each such unit matches that of the system. Finally, the proposed design is based on *applicative tasks*, which are even more suitable for recovery purposes than are *atomic actions*. When an atomic action fails, the machine state has to be restored to that which existed before the atomic action began execution. An applicative task, on the other hand, can simply be re-executed from the task definition because the corrupted machine state can be ignored by future computation. Since no state restoration is required, the functional approach is also free from the *Domino*

Effect.

Applicative systems exploiting large–grain parallelism are considered in this paper. While studies on fault–tolerance aspects of such systems have not been reported in literature, researchers have studied fault tolerance issues in machine architectures exploiting fine–grain parallelism – namely, dataflow and reduction architectures.

Several fault tolerance techniques for dataflow machines have been presented by Hughes [Hugh83], Leung and Dennis [Leun80], Misunas [Misu76] and Srini [Srin85]. These techniques, which include checkpointing the entire system state, encoded interconnection, and triple–modular redundancy, do not take advantage of the functional environment characteristics. Grit has outlined fault tolerance mechanisms for reduction–based multiprocessors [Grit84] and more recently, recovery issues for similar systems have been studied by Lin and Keller [LiKe86]. The target systems in these studies exploit parallelism at a very *fine grain*. Functional programming is, however, also very attractive for exploiting parallelism at higher levels. Parallel systems with a coarser grain of parallelism incur less overhead and may be considered more pragmatic. Large–grain parallelism has been employed in systems such as the Rediflow multiprocessor [Kell84], AMPS [Kell79], and in LISP systems such as those proposed in [Hals84] and [Gabr84], which permit side effects.

The work in [Grit84] and [LiKe86] is most closely related to this paper. The specific differences are again primarily due to the granularity at which parallelism is exploited. In Grit's paper, each task reduces a sub–expression and returns a result value to its parent. In architectures considered in our paper, data structures such as trees and linked lists are allowed. Consequently, a node does not simply receive a single result value from its child, but may receive a pointer to an entire tree or list. Elements of the data structure might have been computed by various other nested functions. This consideration significantly changes the problem of identifying the set of tasks to be re–executed for recovery from errors. Also, while Grit's is a message passing model, our architecture uses a shared memory system (in the sense of a single global address space). Pointer based data structures necessitate shared memory. Finally, in the design of fault tolerance mechanisms, Grit has assumed the processors to be *fail–stop*; that is, a processor becomes

silent when a fault occurs. In our study, processors are capable of corrupting the system and mechanisms have been designed to minimize, detect and recover from such corruption. Lin and Keller's work [LiKe86] is devoted to recovery issues alone. They, too, consider processors to be fail–stop. The discussion is at an abstract "process" level and does not consider hardware or software implementation issues like organization of stacks and heaps and synchronization between tasks.

We begin with a definition of the system architecture in Sec. 2. Sec. 3 describes a set of mechanisms for upset exposure and fault containment. In Sec. 4, we propose a task assignment protocol called the *Dialogue*. The Dialogue defines messages between processor and memory modules and serves to establish a mutual watchdog relationship between them. Sec. 5 discusses implementation of replication schemes within the functional framework. Secs. 6 and 7 are devoted to recovery issues. In Sec. 6, it is shown that applicative architectures permit distributed, asynchronous, and incremental backups of the system state for recovery. Finally, recovery procedures are presented in Sec. 7.

2. SYSTEM ARCHITECTURE

In this section, an abstract system architecture for an applicative multiprocessor system is defined. This definition will serve as the target architecture for fault tolerance techniques described subsequently. In specifying the architecture, we maintain a level of generality and abstraction such that a large class of implementation strategies may be covered.

2.1. Multiprocessor Organization

The multiprocessor consists of a pool of processing elements (PEs), a pool of memory modules (MMs), and an interconnection network (IN). Since no assumptions are made regarding the interconnection network, a large class of machine architectures are covered. Thus, widely differing configurations such as shown in Figures 1a and 1b are permitted. Figure 1a represents a pool of processors sharing a multiport memory system via a multistage interconnection network as in the CEDAR multiprocessor. Figure 1b, on the other hand, shows a configuration where processor–memory modules are connected together with an

interconnection network as in hypercube architectures.

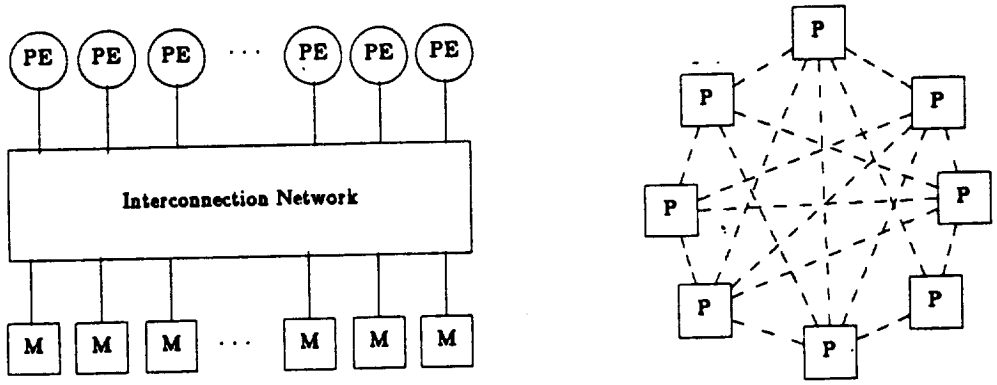
2.2. Programming Environment

Programs are written in a functional language like *pure LISP* and are executed concurrently on the multiprocessor. Parallelism is achieved by evaluating function arguments (and, possibly, the function definition itself) in parallel. In order that the system be efficient, not all expressions are evaluated as separate tasks. Instead, processes are spawned only for *major* function applications. *Major* functions may be identified either by a compiler or by the programmer.

It should be noted that single assignment languages are permitted. It shall also become evident that programs are required to be functional only externally, that is, as seen by the system memory and other processors. A task could internally use side–effects as long as they are limited to PE registers or local memory.

2.3. Operation

A request for the application of a *main* function to a given set of arguments initiates a computation. The *main* function application is evaluated by a recursive spawning of subtasks that evaluate



(a) (b)

Fig. 1. Possible Multiprocessor Configurations

subexpressions. The subtasks may execute on the same PE as the current task or may be assigned to another one. After creating children tasks, a parent suspends itself until results from the children become available. When the children return results, the parent continues execution and finally dies, producing results for its parent. Such operation results in a dynamically growing and collapsing task tree.

2.4. Data and Control Structures

Task Nomenclature

Ancestors and successor tasks for a given task have to be identified at run time for the purpose of recovery. We define task-ids so that this identification becomes trivial. A task-ID of "1" is assigned to the root task. Assuming a binary task tree, the children of a task N are assigned IDs of $(2N)$ and $(2N+1)$. The nomenclature is easily extended to allow an arbitrary number of children for each task. For example, if a task has three children, IDs of $(4N)$, $(4N+1)$ and $(4N+2)$ are assigned to the children.

Task Descriptors

A *Task Descriptor (TD)* is illustrated in Figure 2a. It defines an applicative task completely. A task

Task ID
Pointer to Function
Pointer to Argument List
Pointer to Environment

Fig. 2a. Task Descriptor

Task ID
Task Status
Result (value/pointer)

Fig. 2b. Task Token

creates a subtask simply by placing a TD for the subtask on a task queue.

Task Tokens

Task tokens (Figure 2b) are used for synchronization between a parent task and its children. When a task is created, the parent also creates a corresponding token. The child returns its result via the token. The status field in the token is used to indicate one of five states of the task:

X : Task does not exist.
C : Task has been created but has not yet been scheduled for execution.
A : Task is active, i.e., it is currently being executed by some PE.
S : Task is suspended. It spawned subtasks and is waiting for results.
D : Task is complete. The result field in the token contains the value of, or pointer to, the result.

Task tokens are organized in a tree. Each token includes pointers to tokens for subtasks. Thus, given the task-ID, and given that the location of the root token is fixed, the token of any task can always be located.

Heap Segments

As in LISP, data cells for each task are allocated from a heap. The heap is implemented as a distributed set of subheaps – one in each memory module. Heaps are broken down into *Heap Segments* – each segment being a contiguous set of cells. Each activation of a task uses a new heap segment.

Control Stack

Each task also gets a *Stack Segment* to be used as a control stack for calls to non–task functions (functions that are executed sequentially within a task and not as independent tasks). When a task spawns subtasks and suspends itself, the corresponding stack segment also becomes dormant. When it resumes, it continues to use the same stack segment. Finally, when the task completes and returns its results to the

parent, the stack segment is purged.

2.5. Information Transfer Relationships Between Tasks

A task is assumed to spawn all of its subtasks at one time and immediately suspend itself. Thus, in the absence of faults, each task is active only twice in its lifetime – once, when it first comes up for execution after creation, and next, when it resumes execution after a suspension. The state A, therefore, consists of two substates, $A1$ and $A2$. $A1$ corresponds to the first activation of the task and $A2$ to the second. This assumption is necessary only for simplicity in illustrating the fault tolerance mechanisms. Extension of the techniques to allow an arbitrary number of activations can be made easily.

Notation and Definitions :

- T_i denotes a task with task-ID $= i$.
- T_i^a represents the a'th activation of a task whose task-id is i. a can be 1 or 2 corresponding to states A1 and A2 of the task. Each T_i^a is referred to as a "task".
- H_i^a is the Heap Segment for task T_i^a.
- Σ_i is the Stack Segment for task T_i.
- $A(T_i^a)$ is defined as the set of all ancestors of the task T_i^a.
- $S(T_i^a)$ is the set of all successors of the task T_i^a.
- $A_T(T_i^a)$ is defined as the set of all tasks in $A(T_i^a) \cup S(T_i^a)$ that are executed before T_i^a.
- $S_T(T_i^a)$ is defined as the set of all tasks in $A(T_i^a) \cup S(T_i^a)$ that are executed after T_i^a.

The arrows in Figure 3 indicate the order of execution of the tasks and the flow of information between them. For example, for the task T_1, $A(T_1)=\left\{T_0\right\}$, $S(T_1)=\left\{T_3, T_4\right\}$. Similarly, for the task T_1^2, $A_T(T_1^2)=\left\{T_0^1, T_1^1, T_3, T_4\right\}$ and $S_T(T_1^2)=\left\{T_0^2\right\}$. Thus, a task T_i^a can reference any data produced by tasks in

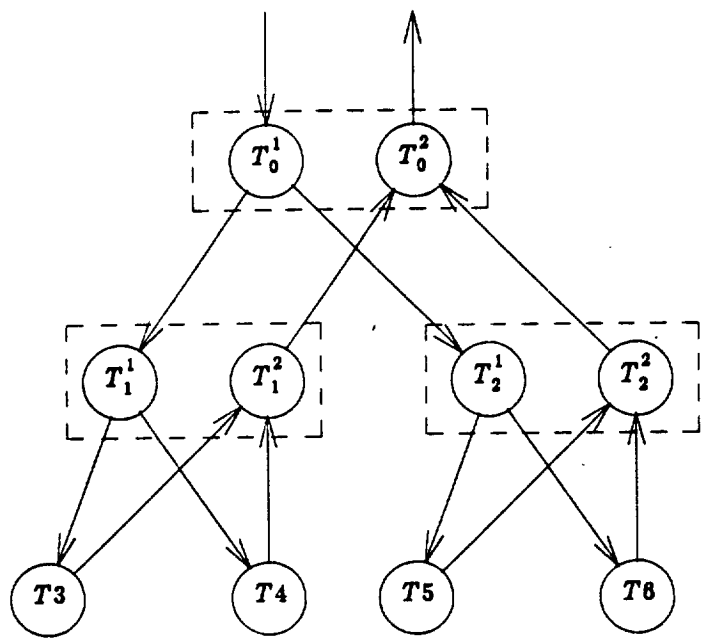


Fig. 3. Information transfer relationship between tasks

$A_T(T_i^a)$ and conversely, results produced by T_i^a can be referred to only by tasks in $S_T(T_i^a)$.

3. UPSET EXPOSURE AND ERROR CONTAINMENT

Hardware mechanisms described in this section limit error propagation between tasks and, at the same time, permit concurrent detection of errors due to a large class of failures in the system. The approach is based on verification of abstract behavior of applicative tasks and very little extra hardware is required to achieve a large fault coverage.

3.1. Motivation

In a functional environment, each data (heap) cell is written only once. All subsequent references to the cell are read references. Further, the cell is written as soon as it is allocated from the heap – for instance, as a result of a *cons* operation. Cells allocated to and written by *(and only by)* a task T_i^a constitute the *Heap Segment* H_i^a of the task. Only *future* activations of tasks in $A(T_i^a) \bigcup S(T_i^a)$ can read this cell. We have defined this set as $S_T(T_i^a)$. An error in a cell in H_i^a can, therefore, propagate only to tasks in $S_T(T_i^a)$. Reasoning along similar lines, an error in the stack segment Σ_i or in the token τ_i can

propagate only to tasks in $S_T(T_i^a)$.

Definition : We define the *domain* D_i^a of a task T_i^a as the set $H_i^a \bigcup \Sigma_i \bigcup \tau_i$.

We can therefore state,

Property 1 : An error in D_i^a can only propagate to tasks in $S_T(T_i^a)$.

Consider the occurrence of a fault during the execution of a task, T_i^a. If due to this fault, the error is confined to D_i^a, then it can propagate only to $S_T(T_i^a)$. However, if the fault causes T_i^a to overwrite data in another write–space D_j^b, then the error could propagate to both, $S_T(T_i^a)$ and $S_T(T_j^b)$. Further, since a number of tasks in the latter set might already have been executed, recovery from the error in D_j^b may be expensive. Therefore, the domain D_i^a should be shielded from faults not "associated" with the execution of T_i^a.

Definition : We define a fault–class F_1 as the set of all faults, under the presence of which

(1) D_i^a can be corrupted only while T_i^a is executing, and

(2) T_i^a cannot corrupt anything but D_i^a.

In terms of this definition, our strategy for fault containment will be that of covering as large a set of hardware and software failures as possible within F_1.

3.2. System Design for Isolation of Heap and Stack Segments

We confine all three components – the heap segment H_i^a, the stack segment Σ_i^a, and the task token τ_i, of a domain D_i^a (corresponding to a task T_i^a) – within the same physical memory module. Thus, during the execution of a task T_i^a, the processor executing the task writes only to this MM. Further, no other processor can write to this module during the execution of T_i^a. In other words, for no pair of tasks T_i^a and T_j^b, executing concurrently at a given time, are the domains D_i^a and D_j^b in the same memory module.

Each processor can be considered as being coupled to a MM. In a distributed system such as shown in Figure 1a, this MM might be the processor's local memory. In a shared memory multiprocessor such as shown in Figure 1b, it may be any memory module. Under fault–free operation, a PE will write only to

it's MM. Therefore, in the absence of faults, no two concurrent tasks write data cells in the same memory module. Of course, any task may, at any time, read data from any other memory module.

Heap space in each memory module is written in a monotone increasing order. A cell $i+1$ in a module is written only after i has been written. Further, since only one task can be allocated heap cells from a given memory module, the heap segment H_i^a of a task T_i^a essentially consists of contiguous cells. If a PE executes tasks T_{i1}^{a1}, T_{i2}^{a2}, T_{i3}^{a3}, etc. in that order, then the corresponding memory module will have heap and stack segments as shown in Figure 4.

Thus, a simple counter at the memory module, serving as a *top-of-the-heap pointer*, can be used to check write accesses by the PE as illustrated in Figure 5. The counter is incremented at every heap segment write access to the memory module. If the processor attempts a "heap segment write" to a location other than the cell pointed to by the counter, the write is aborted and an error is signaled. Thus a task T_i^a is not allowed to destroy the contents of a heap cell not on its heap segment H_i^a.

Another up/down counter is used to point to the top of the stack segment currently in use. When a PE starts using a stack segment Σ_i, it loads the counter with a pointer to the top of the segment. Subsequently, for each stack access (*push* or *pop*), the pointer follows the top of the stack. When a PE presents an address for a *push* or *pop* operation, the address is compared with the contents of this counter. An attempt to write an illegal location in the stack space is flagged as an error.

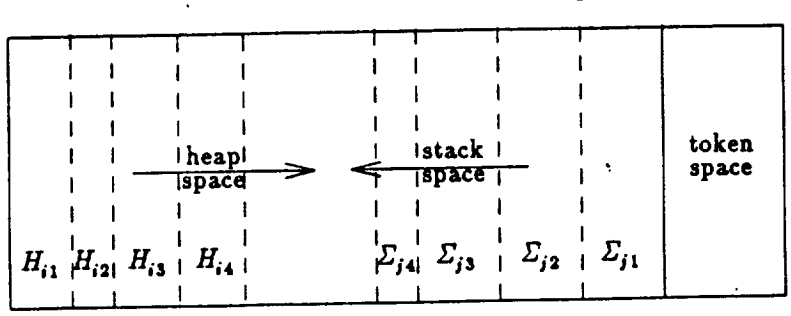


Fig. 4. Heap, Stack, and Token Spaces in Memory Modules

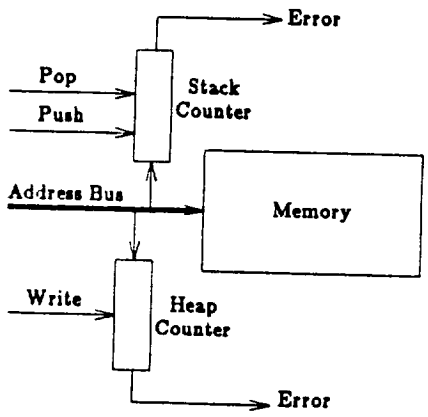


Fig. 5. Stack and Heap Counters at Memory Modules

3.3. Error Containment and Fault Coverage

The above scheme realizes the goal of isolating task domains from errors in the execution of other tasks. The consistency checks enforced by the counters are very strong since the counters identify the specific memory locations that can be written by the processor. Consequently, all transient and permanent addressing faults in the PE, except for those involving heap segment read accesses, will be detected in addition to any other PE malfunction leading to inconsistent memory references. Permanent interconnection network faults leading to misrouted or dropped messages will also be detected.

4. SECURE TASK ASSIGNMENT

We define a task assignment protocol, called the *Dialogue*, which essentially results in a two-way "watchdog" relationship between a PE and the corresponding MM. The *Dialogue* defines messages to be used during task assignment phases and is aimed at achieving redundancy in information pertaining to the assignment of tasks. The protocol is based on the following primitive messages between a PE and a MM :

```
request_dialogue ( MM-id );

begin_dialogue ( PE-id );
```

```
suspend_dialogue ( PE-id or MM-id );
resume_dialogue ( PE-id or MM-id );
exit_dialogue ( MM-id );
respond_if_alive ( PE-id or MM-id );
```

We illustrate the protocol with the following examples.

A PE indicates to an MM that it wishes to start executing a new task by sending to the MM a *request_dialogue* message. This results in a packet

```
< PE-id, req_dialogue, MM-id >
```

to be sent to the MM. The MM responds with a *begin_dialogue* message. The PE then executes a task corresponding to a task descriptor or stack segment residing on the particular MM. When a task is complete, the PE explicitly indicates this by sending an `exit_dialogue` message to the MM.

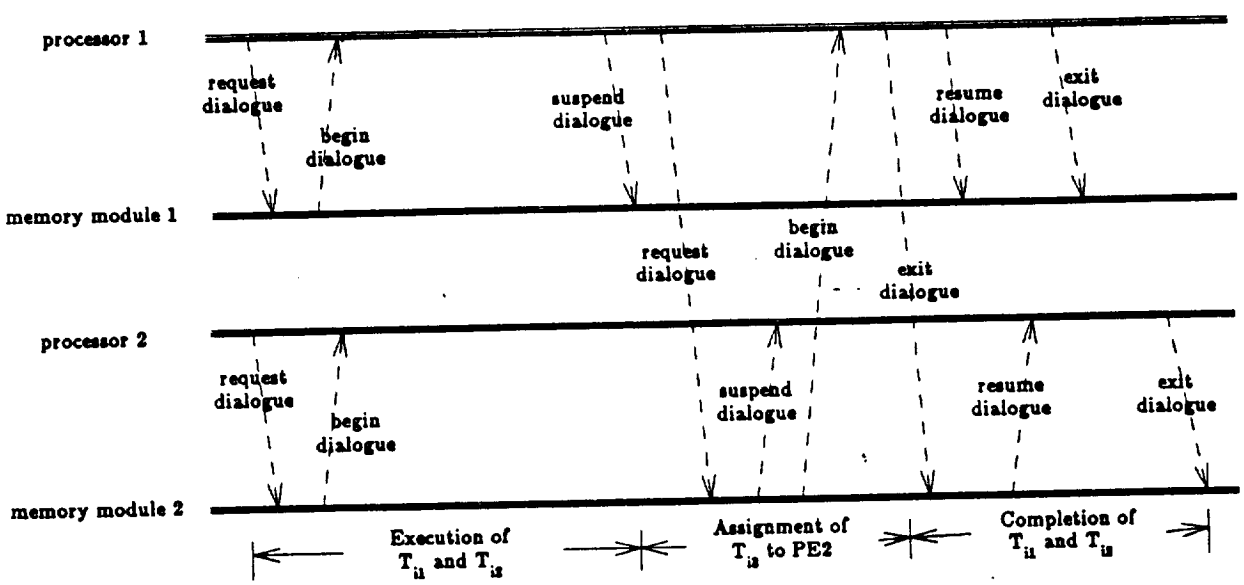


Fig. 6. *Dialogue :* **The Task Assignment Protocol**

A PE may suspend the current dialogue and request permission to join another dialogue when it wishes to place a task descriptor in the stack space in a memory module other than the one it is currently in dialogue with. Such suspension and subsequent resumption is accomplished using *suspend_dialogue, and resume_dialogue* and is illustrated in Figure 6. The figure depicts two PE–MM pairs executing tasks T_{i1} and T_{i2} respectively. At some point during the execution of T_{i1}, a new task T_{i3} is to be created and assigned to PE2. PE1 suspends its current dialogue and requests a dialogue with MM2. In response, MM2 suspends its dialogue with PE2 and begins one with PE1. PE1 places a task descriptor for T_{i3} on a fresh stack segment in MM2 and exits from the dialogue. It then resumes its dialogue with MM1 while MM2 resumes its dialogue with PE2. The `respond_if_alive` message is a diagnostic message and is used either when an inconsistent message is received or when a response expected from a partner is not received in time.

With this protocol, redundancy in information pertaining to the assignment and execution of tasks is achieved. While a task is being executed on a PE, the corresponding MM knows the ID of the task and both expect well-defined responses from each other. An error is signaled when a partner does not meet the protocol definition due to a failure. Further, any erroneous messages received by a PE–MM pair participating in a dialogue from another PE or MM can be detected.

5. ERROR DETECTION BY REPLICATION

Error detection techniques described in Secs. 3 and 4 are not adequate for detection of errors such as incorrect arithmetic computation within the processor. Redundant computation using a second set of resources (processor, memory, and interconnection) and a subsequent comparison of results may be employed for detection of such errors. This section discusses how dual-redundancy may be incorporated within the applicative task framework. The approach serves not only to detect an error, but also to identify the particular task or set of tasks required to be re-executed for recovery.

The task tree is duplicated by invoking two copies of the top-level (main) function. We assume that the scheduling mechanism guarantees that the resources used by the two copies of each task are distinct. It

is also assumed that the scheduling mechanism is such that the two copies of each task execute close to each other in time so as to minimize error detection latency.

A compiler inserts *Compare_results* tasks in the task tree which compare results from sets of twin tasks. Note that a comparison of results of every set of twin tasks is not required. The *Compare_results* tasks may be inserted at random points in the task tree (and one at the top). Choosing the density of "compare points" essentially involves a tradeoff between the cost of the comparisons and the error detection latency. Since tasks may return single values (atoms) or may return pointers to data structures, comparisons for tasks returning atoms will cost much less than for those returning larger data structures. Consequently, it is preferable that a large fraction of the comparison points cover tasks that return atoms rather than large data structures.

While the *Compare_results* task appears in both copies of the program, it is executed only once for each pair of twin tasks. For the twin task that completes first, the compare task does nothing because results from the second are not yet available. When the second completes, the parent compare task actually carries out a comparison of the results. Once an error is found as a result of the comparison, the domain D_i^a in which the error first appeared is to be identified. This is accomplished by comparing results of tasks along the erroneous path in the task tree.

6. SYSTEM MEMORY BACKUPS

Backups of the system memory onto a secondary storage are required for recovery after the detection of an error. While systems based on imperative programming need to synchronize and backup the entire system memory at every checkpoint, our architecture permits distributed, unsynchronized and incremental memory backup.

Data cells on heap segments are written only once during the execution of an application. At any given time, the value of the *top-of-heap* pointer indicates what locations have been written until that point in time. Thus the heap space may be backed up incrementally – only memory areas written after the previous backup have to be stored. Further, the backup process need not synchronize with the

computation.

Unlike the heap space, memory space used by stack segments is written more than once. Stack segments are allocated when a task is spawned, are purged when the task completes, and may subsequently be reused for another task. However, incremental backups are still possible. A stack segment for a task may be copied out to a backup store when it spawns off child tasks and suspends itself.

The memory space allocated for task tokens is the only area which may be modified by the processors at random. The token space in each memory module, therefore, has to be backed up periodically in entirety. However, this should constitute only a small fraction of the memory space. In this case too, the backup may proceed independently of the computation.

Task Descriptors may be written out to a backup store when a task is created. Since TDs for any task can be regenerated if needed by re–executing its parent, the recovery procedures do not require a backup of all TDs. Thus, only TDs for tasks at, say, every n–th level in the task tree are stored. There is essentially a tradeoff to be made between the overhead of re–executing ancestor tasks to regenerate a given TD, and storing a large number of TDs.

It is the "write–once" characteristic and the consequent determinacy of applicative environments which leads to the inexpensive backup scheme. In contrast, imperative environments not only require a synchronization of all processors at a barrier, but also have to save the entire system memory at each checkpoint. Distributed checkpointing schemes may lead to the potentially disastrous *domino effect.*

7. RECOVERY PROCEDURES

7.1. Recovery From Errors Detected in D_i^a During Execution of T_i^a

When an error due to a fault of class F_1 is detected during the execution of a task T_i^a, recovery is

accomplished by the following procedure :

Recovery Procedure P1 :

If $a = 1$, then

(1) Kill any tasks in $S_T(T_i^a)$ that have already been spawned.

(2) Purge the stack segment Σ_i.

(3) Re–execute the parent task T_j^1 using the task descriptor, TD_j^1 to reconstruct the task descriptor TD_i^1.

(4) Retry T_i^1 using TD_i^1 and with a new heap segment $\hat{H}_i^1$ and using a new stack segment Σ_i.

If $a = 2$, then steps 1 and 2 are the same. Step 3 and 4 become :

(3′) Re–execute T_{i_1} using the task descriptors TD_i^1 to reconstruct Σ_i for T_i^2.

(4′) Retry T_i^2 using Σ_i and with a new heap segment $\hat{H}_i^2$.

Property 2 : Recovery procedure P1 preserves correctness of results.

Proof : Consider steps 1 and 2. From the information transfer relationship between tasks shown in Figure 2, it is clear that any results computed in $S_T(T_i^a)$ are not referred to by any tasks in $\overline{S_T(T_i^a)}$. Thus killing any tasks in this set that may already have been spawned does not affect any uncorrupted tasks in $\overline{S_T(T_i^a)}$. Of course, the tokens for the killed tasks are reset to state X and their stack segments purged.

In step 3, T_j^1, the parent of T_i^1, is re–executed in a new heap segment $\hat{H}_j^1$ to reconstruct Σ_i. T_j^1 thus has two heap segments, H_j^1 and $\hat{H}_j^1$. While the re–execution of T_i^1 will use the latter, other children of T_j^1 will continue to use the former. H_i^1 and $\hat{H}_i^1$ are identical except for pointers that point within the same heap segment. That is, a pointer p in H_i^1 to a cell within H_i^1 will not be equal to the corresponding pointer $\hat{p}$ in $\hat{H}_i^1$. All pointers in H_i^1 to cells not in H_i^1 will be identical to corresponding ones in $\hat{H}_i^1$. Further, any atoms in H_i^1 will be identical to those in $\hat{H}_i^1$. The information contained in the two heap segments is, thus, equivalent. Similarly, the new task descriptor for T_i^a is also equivalent to first.

Correctness for the case a=2 (steps 3′ and 4′) can be established along similar lines.

7.2. Recovery from Errors Detected in D_i^a After Execution of T_i^a

A. Lost Data Cells

Case (1) : *The lost data cell is available in backup store.*

The contents are simply restored from the secondary backup memory.

Case (2) : *The lost data cell was created since the last backup.*

In this case, T_i^a and all tasks in $S_T(T_i^a)$ have to be re-executed. Note that since T_i^a was executed after the last backup, the set $S_T(T_i^a)$ contains only a small number of tasks. Damage, therefore, is small.

B. Lost Stack Segments

Stack segments are created when a task is created and disappear when the task is complete. Therefore, while a stack segment Σ_i exists, a task could be in states C, A1, A2, or S. One or more cells may be lost during this time. Recovery for cases when a Σ_i is lost while the task T_i^a is in one of the active states, $A1$ or $A2$ has already been considered in Section 7.1. Here, we consider the case when the task is in state C or S:

Case (1) : Task-ID is not lost.

In this case, Σ_i is reconstructed using the steps 3 and 3′ in procedure P1 in Sec 7.1.

Case (2) : Task-ID is lost.

Recovery in this case may appear difficult. Since the task-ID is not known, which stack segment to reconstruct is not known. The approach taken is that of purging the stack segment and waiting for an eventual deadlock state where there are no active tasks in the system. In this state, all tasks in $A(T_i^a)$ are suspended because the parent of T_i^a is waiting for results from T_i^a. Furthermore, tasks in $S(T_i^a)$ will all be in state D (done) or X (do not exist). From this situation, the Task-ID can be inferred. Subsequently, Σ_i can be reconstructed and computation can proceed.

Table 1. Cost of recovery from each type of error

Error Class	Type of Error	Recovery Cost
E_1 (Sec. 7.1)	Errors in D_i^e detected during execution of T_i^e	Re–execution of single task
E_2 (Sec. 7.2A)	Lost data cells in D_i^e detected by task other than T_i^e	If available in backup, then recovered from back-up else re–execution of few tasks
E_3 (Sec. 7.2B)	Lost cells in Σ_i detected by task other than T_i^e	Re–execution of single task
E_4 (Sec. 7.2C)	Lost token T_i detected by task other than T_i^e	Re–execution of single task in worst case
E_5 (Sec. 7.3)	Loss of PE	Re–execution of single task. system continues to operate with performance degradation corresponding to loss of PE
E_6 (Sec. 7.3)	Loss of memory module	Re–execution of a small number of tasks

C. Lost Tokens

The token for a task T_i is referenced by the scheduler to determine whether the task is done and by the parent task which uses the result field of the token. In both cases, the token is examined only under the situation where the parent is suspended. A token can be reconstructed as follows. We first check the status of the children of T_i. *If* all children have a status X or D, then reconstruct the task descriptor for T_i and re–execute T_i to reconstruct the token. *Else*, since the children have been created and are active or suspended, the task T_i is suspended. Therefore, the status field in the token is set to S.

7.3. Recovery from Loss of a Memory Module or PE

The loss of a memory module implies the loss of heap segments, stack segments, and tokens for all tasks corresponding to that module. However, if a spare memory module is available and can be reconfigured into the system so as to replace the defective one, recovery can be easily accomplished. The procedure consists of the following steps:

(1) Restore, from backup storage, the saved contents of the lost memory module. Since the restored contents represent the state at the time of the backup, some information may be lost.

(2) For each heap segment H_i^e lost, kill all tasks in $S_T(T_i^e)$ and re-execute T_i^e. Once again, it should be noted that this set $S_T(T_i^e)$ is small because T_i^e was executed after the last memory backup for the module.

When a task loses a PE, the task is simply re-executed on another PE when one becomes available. The performance of the system degrades due to the loss of the PE.

A summary of recovery costs for each case discussed above is presented in Table 1. It is clear that even for such cases as loss of an entire memory module, the cost of recovery is limited to re-execution of a small set of tasks.

8. CONCLUSIONS AND FUTURE RESEARCH

This paper has provided evidence that functional languages are highly amenable to fault tolerant multiprocessor design. Three primary characteristics of functional environments are exploited in the proposed architecture :

- *Well-defined behavioral characteristics of applicative tasks,* which lead to upset exposure techniques relying on verification of the behavior, and which signal errors on detection of any illegal behavior.
- *Determinacy,* or the property that given a set of arguments, the application of the function will always produce the same result. Determinacy leads to inexpensive recovery procedures.
- *Side-effect free nature and the single assignment rule,* as a result of which we can do away with expensive recovery cache mechanisms and can design recovery procedures free from the Domino

Effect. Further, the single assignment rule permits incremental machine state backups.

Issues pertaining to garbage collection and other non–functional operating system tasks have not been discussed in this paper. Future work is needed to incorporate them within the functional framework for fault tolerance. Approaches for handling data structures such as (infinite) streams, exception mechanisms such as *catch/throw*, and for lazy/eager evaluation strategies, are also candidates for future research.

9. REFERENCES

[Gabr84] R.P. Gabriel and J. McCarthy, "Queue-based Multi-processing LISP," *Proceedings, 1984 ACM Symposium on LISP and Functional Programming,* pp 25–43.

[Grit84] D. H. Grit, "Towards Fault Tolerance in a Distributed Applicative Multiprocessor," *Proceedings, International Symposium on Fault Tolerant Computing,* Jun 1984, pp 272–277.

[Hals84] R.H. Halstead, Jr., "Implementation of Multilisp : LISP on a Multiprocessor," *Proceedings, 1984 ACM Symposium on LISP and Functional Programming,* pp 9–17.

[Hugh83] J. L. A. Hughes, "Error Detection and Correction Techniques for Data-Flow Systems," *Proceedings, International Symposium on Fault Tolerant Computing,* June 1983, pp 318–321.

[Kell79] R. M. Keller, G. Lindstrom, and S. Patil, "A Loosely-Coupled Applicative Multi-processing System," *AFIPS Conference Proceedings,* June 1979, pp 613–622.

[Kell84] R. M. Keller, F. C. H. Lin, and J. Tanaka, "Rediflow Multiprocessing," *Proceedings, COMPCON Spring 84,* Feb. 1984, pp 410–417.

[Leun80] C. K. C. Leung and J. B. Dennis, "Design of a Fault-Tolerant Packet Communication Computer Architecture," *Proceedings, International Symposium on Fault Tolerant Computing,* 1980, pp 328–335.

[LiKe86] F.C.H. Lin and R.M. Keller,"Distributed Recovery in Applicative Systems," *Proceedings, 1986 International Conference on Parallel Processing,* pp 405–412.

[Misu76] D. P. Misunas, "Error Detection and Recovery in a Data-flow Computer," *Proceedings, 1976 Conference on Parallel Processing,* August 1976, pp 123–131.

[Rand84] B. Randell, "Fault Tolerance and System Structuring," *Proceedings, 4th Jerusalem Conference on Information Technology,* May 1984.

[Srin85] V.P. Srini, "A Fault Tolerant Dataflow System," *IEEE Computer,* March 1985, pp 54–68.

[Vegd84] S.R. Vegdahl, "A Survey of Proposed Architectures for the Execution of Functional Languages," *IEEE Trans. on Computers,* Dec. 1984, pp 1050–1071.

Proceedings Mathematics Computer Performance and Reliability Workshop, Rome, Italy, May 1987, pp. 337-352.

N94-70425

57-62
5
p 16

2nd International Workshop on Applied Mathematics and Performance/Reliability Models of Computer/Communication Systems University of Rome II, 1987

A MEASUREMENT-BASED PERFORMABILITY MODEL FOR A MULTIPROCESSOR SYSTEM

M.C. Hsueh and R.K. Iyer †
Computer Systems Group, Coordinated Science Laboratory
University of Illinois at Urbana-Champaign, Urbana, IL 61801

K.S. Trivedi ‡
Computer Science Department, Duke University, Durham, NC

This paper describes a measurement-based performability model based on real error-data collected on a multi-processor system. Model development from the raw error-data to the estimation of cumulative reward is described. Both normal and failure behavior of the system are characterized. The measured data show that the holding times in key operational and failure states are not simple exponential and that a semi-Markov process is necessary to model the system behavior. A reward function, based on the service rate and the error rate in each state, is then defined in order to estimate the performability of the system and, to depict the cost of different failure types and recovery procedures.

1. Introduction

The development of realistic models to describe the failure behavior of computer systems is a difficult problem. Although many authors have addressed the modeling issue and have significantly advanced the state of the art there is little or no validation of these models with field data. It is, therefore, extremely valuable to model the failure and recovery process in a production system using real error data. Apart from providing useful information on how failures occur, this process also provides insight into the interaction between various system components.

In this paper we build a semi-Markov model to describe the resource-usage/failure/recovery process in a large mainframe system. The model is based on low-level error and performance data collected on an IBM 3081 system during its normal operation. Both the normal and erroneous behavior of the system are modeled. The results, therefore, provide an understanding of the different failure and recovery processes and their relationship to various types of resource usage. Hardware and the software reliability and their interaction is also modeled. A reward function based on the service rate and the error rate in each state is then defined in order to estimate the performability of the system, and to depict the cost of different failure types and recovery procedures.

Section 2 surveys related research in the modeling area and motivates our current research. Section 3 uses statistical clustering to build a state-transition model of resource usage. Different types of component failures and recovery procedures are modeled in Section 4. A semi-Markov model of resource-usage/failure/recovery process is described in Section 5, and an analysis of the model behavior is performed. In Section 6 we extend the model of section 5 to a semi-Markov reward model in order to include the inter-relationships between workload and failure/recovery processes. Section 7 highlights important conclusions arising out this work.

† This work was supported in part by NASA grant NAG-1-613, in part by the IBM Corporation and in part by the Joint Services Electronics Program (U.S. Army, U.S Navy, U.S. Air Force) under contract number N00014-84-C-0149.

‡ This work was supported by the Air Force Office of Scientific Research under contract AFOSR-84-0132.

PRECEDING PAGE BLANK NOT FILMED

2. Related Research

Analytical models for hardware failure have been extensively investigated (see [1] for a summary). Although the time for different components to fail is usually assumed to be exponentially distributed, time-dependent failure rates and graceful degradation [2, 3] have been considered along with performability issues. Non-exponential distributions can also be modeled using the method of stages [4], but for complex systems this method can result in an explosion of states. A job/task flow based model is described in [5]. Failure occurrence is assumed to be a linear function of the service requests from a job/task flow. As shown in [6], the assumption of linearity may result in underestimating the effect of the workload, especially when the load is high.

Most software reliability models usually refer to the development, debugging and testing phases of the software [7]. Few of these models have been applied to the operational phase. In [8] and [9], software failures in an operating environment are studied. Both studies found that at least 60% of system failures were software related. There is little explicit study of hardware/software reliability. In [10], software failures related to hardware problems in the operating system are analyzed and it is shown that errors in the hardware/software interface are often fatal. In [11], a methodology for joint hardware/software model construction and model processing using Stochastic Petri Nets is described.

With the exception of software reliability growth models, very few of either the hardware or the software models have been validated with real data. Exceptions are the hardware and software model discussed in [12] and, a measurement-based model of workload dependent failures discussed in [6]. Both, however, only describe the external behavior of the system and do not provide insight into component level behavior. It is, therefore, highly instructive to construct a detailed model based on low-level error data from a production system. Toward this end in the following sections, we construct a joint resource-usage/failure/recovery model using error and workload data from a production system.

3. Resource Usage Characterization

In this section we build a state-transition model to describe the variation in system activity. It will be seen in Section 5 that this approach allows failure to be considered as a transition from normal activity. System activity is characterized by a number of resource usage parameters. A statistical clustering technique is employed to reduce the potential many to many transitions of the workload vector to a small number of states representative of those found in the data. The data for our studies came from an IBM 3081 system running the MVS operating system. The system consists of dual processors with two time-multiplexed channel sets. Together these two sets allow a maximum of 24 subchannels to be simultaneously active in each I/O cycle. The workload data were collected using the IBM MVS/370 system Resource Management Facility (RMF) [13]. The results presented here are based on three months of sampled RMF data, with a sample time of 500 milliseconds. The sampled data is then averaged on an hourly basis. Four different resource usage measures were selected:

CPU - fraction of the measured interval for which the CPU is executing instructions.

SIO - number of successful Start I/O and Resume I/O instructions issued to the channel.

DASD - number of requests serviced on the direct access storage devices, and

CHB - fraction of the measured interval for which the channel was busy and the CPU was in the wait state (this parameter is usually used to measure the degree of contention in our system).

At any interval of time the measured workload is represented by a point in 4-dimensional space, (CPU, CHB, SIO, DASD). Cluster analysis is used to divide the workload into similar classes according to a pre-defined criterion.[1] This allows us to concisely describe the dynamics of system behavior and extract a structure that already exists in the workload data.[2]

[1] Potentially, we can have uncountably large number of points in the workload space. Intuitively, only a countable number of combinations of four measures do in fact occur. Further, it is seen that they usually occur in clusters.

[2] Similar clustering techniques are also used for workload characterization in [14].

Each cluster (defined by its centroid) is then used to depict a system state and, a state-transition diagram (consisting of inter-cluster transition probabilities and cluster sojourn times) is developed. A k-means clustering algorithm [15] was used for cluster analysis. The algorithm partitions an N-dimensional population into k sets on the basis of a sample. The k non-empty clusters, $C_1, C_2, \ldots, C_k$, sought are such that the sum of the squares of the Euclidean distances of the cluster members from their centroids is minimized, i.e.,

$$\sum_{j=1}^{k} \sum_{i} \| x_i - \bar{x}_j \|^2 \rightarrow \text{minimum}$$

where $x_i \in C_i$ and $\bar{x}_j$ is the centroid of cluster C_j.

Two types of workload clusters were formed. In the first case CPU and CHB were selected to be the workload variables. This combination was found to best describe the CPU-bound load (nearly 60% of the observations have a CPU usage greater than 0.72). In the second case the clusters were formed considering SIO and DASD as workload variables. This combination was found to best describe the I/O workload. Table 1 shows the results for these two cases. An examination of the

Cluster id	% of obs	Mean of CPU	Mean of CHB	Std dev of CPU	Std dev of CHB
W_1	7.44	0.0981	0.1072	0.0462	0.0436
W_2	0.50	0.1126	0.5525	0.0433	0.0669
W_3	2.73	0.1547	0.3801	0.0647	0.0755
W_4	12.41	0.3105	0.1637	0.0550	0.0459
W_5	0.74	0.3639	0.3819	0.0365	0.1923
W_6	17.12	0.5416	0.1287	0.0560	0.0511
W_7	22.58	0.7207	0.0848	0.0576	0.0301
W_8	36.48	0.9612	0.0168	0.0362	0.0143
R^2 of CPU =	0.9724				
R^2 of CHB =	0.8095				
overall R^2 =	0.9604				

(a) CPU bound workload

Cluster id	% of obs	Mean of SIO	Mean of DASD	Std dev of SIO	Std dev of DASD
W_1	8.89	16.80	0.95	6.80	1.30
W_2	36.05	41.59	2.99	7.51	1.92
W_3	1.48	44.37	20.62	8.55	4.18
W_4	1.73	60.07	38.84	6.77	8.42
W_5	42.72	67.34	5.19	7.92	3.72
W_6	0.49	87.30	31.19	3.87	9.84
W_7	7.9	96.20	6.02	8.73	3.34
W_8	0.74	141.10	10.10	10.28	8.50
R^2 of SIO =	0.8861				
R^2 of DASD =	0.7176				
overall R^2 =	0.8751				

(b) I/O workload

Table 1. Characteristics of workload clusters

table shows the dynamics of system behavior. We see in Table 1(a) that about 36% of the time the CPU is highly loaded (0.96) and almost 76% of the time the CPU load is above 0.5. Since the measured system is a two-processor machine, we may say that 76% of the time at least one of the processors is busy. Note that, with increasing CPU usage, CHB (CPU wait and channel busy) decreases. This indicates that resource contention is not a problem in the measured system. In Table 1(b) (the I/O load), both W_2 and W_3 have a very close channel start I/O rate (SIO) but the disk service rate (DASD) of W_3 is as much as 10 times that of W_2. This is most likely due to the fact that some I/O requests result in a burst of data while the others in only a few words. Burst transfer however occurred only 4% of the time ($W_3 + W_4 + W_6$). This result may be due to the fact that our measurements were made during work hours, but I/O bound jobs are normally executed during off-work hours. A state-transition diagram of I/O activity is shown in Figure 1. Notice that a null state, W_0, has been incorporated to represent the state during the non-measured period. The transition probability from state i to state j, p_{ij}, is estimated from the measured data using:

$$p_{ij} = \frac{\text{observed no. of transitions from state } i \text{ to state } j}{\text{observed no. of transitions from state } i}.$$

Figure 1 also confirms our previous observation that most often the I/O workload fluctuates back and forth between two moderate levels, W_2 and W_5 (0.69 and 0.56) and that there are occasional requests for burst I/O.

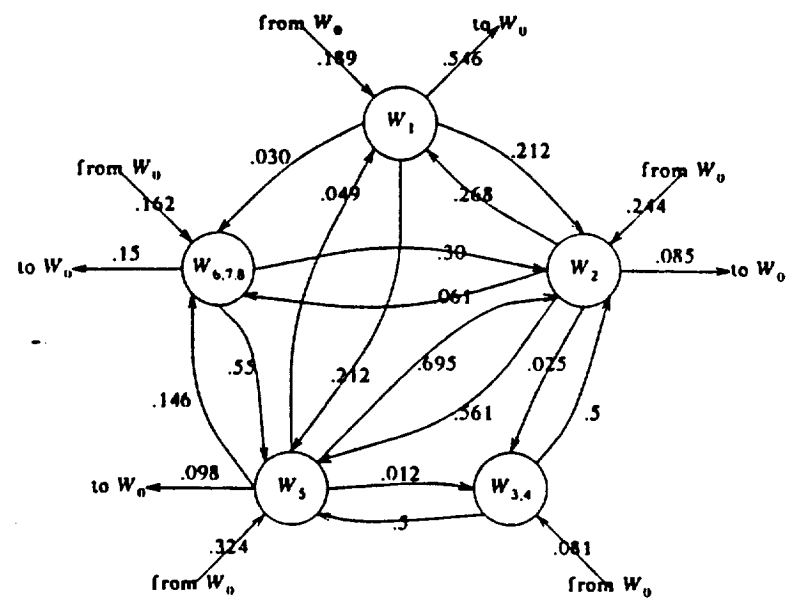


Figure 1. State-transition diagram of resource usage (I/O Workload)

4. Failure and Recovery Characterization

In this section the collection and characterization of the errors and the error recovery is discussed. A state-transition diagram to describe different failure and recovery states is developed. The severity of failures and effectiveness of recovery are obtained.

Data on six different types of errors were collected. These data are automatically logged by the operating system as the errors occur. Details of the logging mechanism are described in [16].

(1) CPU-related errors - those that affect the normal operation of the CPU; the errors may originate in the CPU itself, in the main memory, or in a channel

(2) Temporary channel errors -those that are recovered by channel retry and do not result in the termination of the channel control program

(3) Temporary (soft) disk errors - those I/O errors that are recovered by correcting the data or by retrying the hardware instruction

(4) Temporary (hard) disk errors - those I/O errors that are recovered by software instruction retry or by functional recovery routine

(5) Permanent disk errors - those I/O errors that are not correctable and can not be recovered by retrying the operation

(6) Software errors - software incidents that are due to invalid supervisor calls, program checks and other software exception conditions

Due to the manner in which errors are detected and reported in a computer system, it is possible that a single fault may manifest itself as more than one error, depending on the activity at the time of the error. The different manifestations may not all be identical [17]. The system recovery usually treats these errors as isolated incidents. In order to address this problem, and to ensure that the analysis is not biased by error records relating to the same problem, two levels of data reduction were performed. First, a coalescing algorithm described in [6] was used to analyze the data and merge observations which occur in rapid succession and relate to the same problem. Next, a reduction technique described in [17] to automatically group records most likely to have a common cause, was used. By using these two methods, errors were classified into five different classes. These classes are called error (or failure) events since they may contain more than one error and are defined as follows.

CPU : that caused errors to be logged only as CPU-related errors

CHAN : that caused errors to be logged only as channel errors

SWE : that caused errors to be logged only as software errors

DASD : that caused errors to be logged only as direct access storage device errors

MULT : that caused errors affecting more than one type of components

Table 2 lists the frequencies of different types of failures. Notice that about 17% of failures are classified as multiple failures (MULT). A MULT error is mostly due to a single cause but the fault has non-identical manifestations, provoked by different types of system activity. Since the manifestations are non-identical, recovery may be complex and hence can (as will be seen later) impose considerable overhead on the system.

Type of failure	Frequency	Percent
CPU	2	0.04
CHAN	119	2.23
MULT	924	17.33
SWE	1923	36.07
DASD	2364	44.33
total	5332	100.00

Table 2. Frequency of failures

When an error is detected in the measured system, an appropriate recovery procedure is invoked depending on the severity of the error. The recovery procedures were divided into four categories in increasing order of recovery cost. The recovery cost was measured in terms of the system overhead required to handle an error. The lowest level (hardware recovery), involves the use of an error correction code (ECC) or hardware instruction retry and has minimal overhead. If hardware recovery is not possible (or unsuccessful), the next level, i.e., software controlled recovery, is

invoked. This could be simple, e.g., terminating the current program or task in control, or complex, e.g., invoking specially designed recovery routine(s) to handle the problem. The third level of recovery, alternative (ALT), involves transferring the tasks to functioning processor(s) when one of the processors experiences an un-recoverable error. If no on-line recovery is possible, the system is brought down for off-line (OFFL) repair. Figure 2 shows a flow chart of the recovery process.

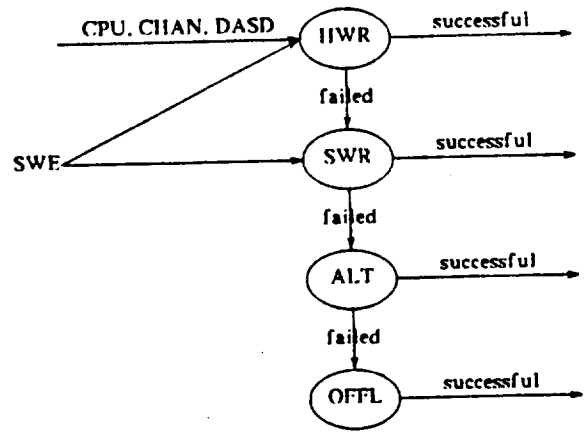


Figure 2. Flow chart of recovery processes

Table 3 lists the distribution of successful recovery due to various levels. About 73% of the errors were successfully handled through hardware recovery and most of the others were recovered from by the software.

Recovery Procedure	Percent
HWR	73.35
SWR	26.56
ALT	0.02
OFFL	0.07

Table 3. Percentage of recovery procedures

5. Resource-Usage/Failure/Recovery Model

In this section we combine the separate workload, failure and recovery models developed so far into a single model shown in Figure 3. The model has three different classes of states: normal operation states (S_N), failure states (S_F), and recovery states (S_R).

Under normal conditions, the system makes transitions from one workload state to another. The occurrence of an error results in a transition to one of the failure states. The system then goes into one or more recovery modes after which, with a high probability, it returns to one of the "good" workload states. The state transition diagram shows that nearly 98.3% of hardware recovery requests and 99.7% of software recovery requests are successful. Thus the error detection, fault isolation and on-line recovery mechanisms allow the measured system to handle an error efficiently and effectively. In only less than 1% of the cases is the system not able to recover.

Figure 4 shows the state-transition diagram of a multiple failure (a MULT event) i.e. given that a multiple failure has occurred. The model shows that disk and software errors are strongly correlated in multiple failures. From the diagram, it is seen that in about 65% of the cases a multiple

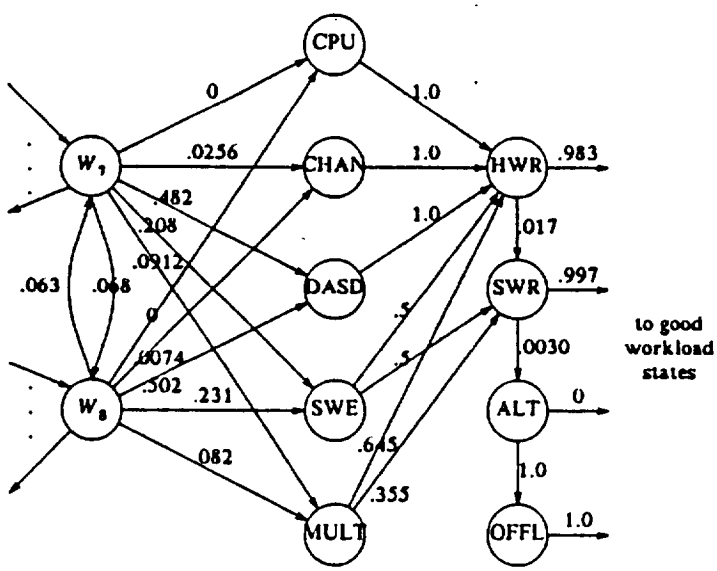


Figure 3. State-transition diagram of resource-usage/failure/recovery model

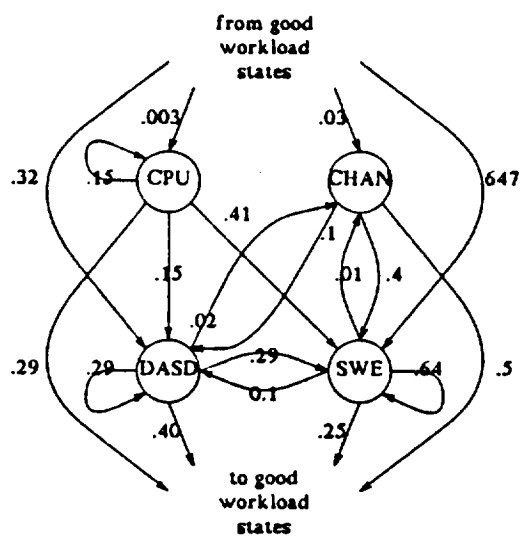


Figure 4. State-transition diagram given a multiple failure (MULT)

failure starts as a software error (SWE) and in 32% of the cases it starts as a disk error (DASD). Given that a disk error has occurred, there is nearly a 30% chance that a software error will follow. It is also interesting to note that there is a 64% chance that one software error will be followed by another different software error.

5.1. Waiting and Holding Time Distributions

Table 4 shows the characteristics of both the workload and failure states in terms of their waiting

State	# of obs	Mean waiting time	Standard deviation	Std Error of mean
W_1	53	1263.71	1384.20	190.13
W_2	2	289.65	1.19	0.84
W_3	20	698.79	913.30	204.22
W_4	130	1203.05	1130.28	99.13
W_5	11	613.74	421.73	127.16
W_6	147	1380.86	1588.76	131.04
W_7	268	1071.31	1004.46	61.36
W_8	266	1612.72	2576.35	157.97

(a). CPU Workload states

State	# of obs	Mean waiting time	Standard deviation	Std Error of mean
CHAN	13	5.08	18.31	5.08
SWE	201	41.35	103.35	7.29
DASD	401	120.86	223.89	11.18
MULT	77	293.28	262.84	29.95

(b). Failure states

Table 4. Characteristics of waiting time (seconds) in workload and failure states

times[3]. Only the CPU workload is shown in the table. Results for the I/O workload were similar. States W_2 and W_5 in Table 4(a) were ignored since they had very few observations. An examination of the mean and standard deviation of the waiting times indicates that not all waiting times are simple exponentials. This is particularly pronounced in Table 4(b) which refers to the failure states. Figure 5 shows the densities of waiting and holding times for one of the workload states, W_8. Figure 5(a) shows the waiting time for W_8 and 5(b) and 5(c) represent the holding time from this state to DASD and SWE failure states. These densities are fitted to phase-type exponential density functions [4].

$$f(t) = \sum_{i=1}^{n} a_i g_i(t)$$

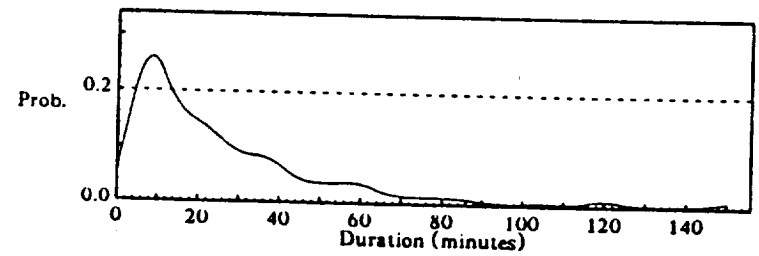
where $a_i \geq 0$, $\sum_{i=1}^{n} a_i = 1$ and n is the number of phases. The $g_i(t)$ function can be an exponential, multi-stage hyperexponential, or muti-stage hypoexponential density function. The densities in Figure 5 were fitted to the following functions (tested by using the Kolmogorov-Smirnov test [4] at a 0.01 significance level).

(1) waiting time : $f(t) = 0.000146e^{-0.002t} + 0.000939e^{-0.00103t} + 0.000033e^{-0.0002102t}$

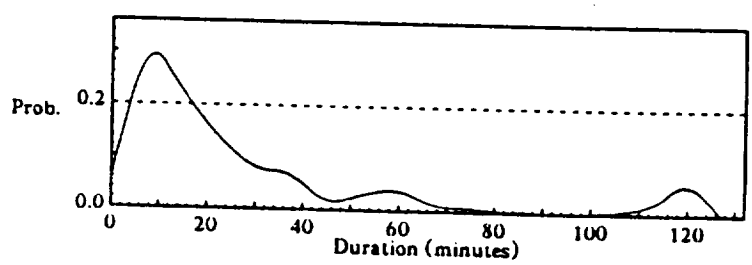
(2) to a DASD failure : $f(t) = 0.00094e^{-0.004t} + 0.0008355(e^{-0.000937t} - e^{-0.006595t})$

(3) to a SWE failure : $f(t) = 0.00085e^{-0.005t} + 0.000701(e^{-0.000716t} - e^{-0.004688t})$

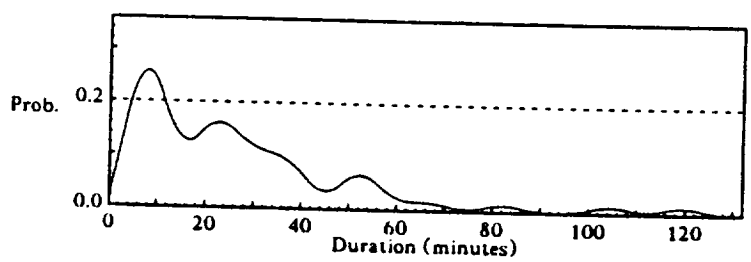
[3] The waiting time for state i is the time that the process spends in state i before making a transition. The holding time for a transition from state i to state j is the time that the process spends in state i before making a transition to state j [18].



(a) Waiting time density of state W_8 (CPU = 0.96)



(b) Holding time density from state W_8 to state DASD



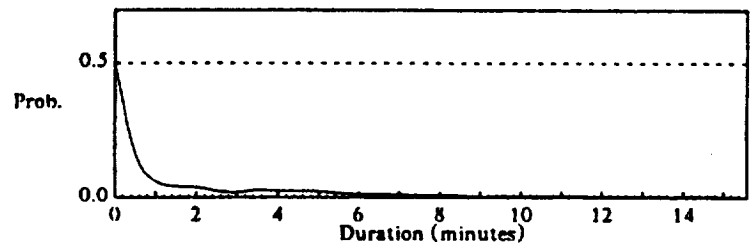
(c) Holding time density from state W_8 to state SWE

Figure 5. Waiting and holding time densities

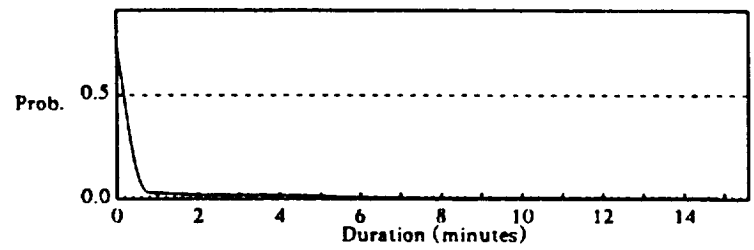
5.2. Recovery Distributions

In our data the selection of the destinations from any state of S_R was found to be independent of the holding time distribution. Further, for our system the time taken for each type of recovery can reasonably be considered constant. The overall recovery time, i.e., the duration of an error event (or the holding time in an error state), however was not constant, since an error event may involve more than one recovery attempt. This time is computed as the time difference between the first detected error and the last detected error, caused by the same event. The duration of an error event can be used to measure the effectiveness of recovery and also the severity of error. Figure 6 shows examples of failure duration densities for three different types of errors. Again, the following phase-type exponential densities were fitted to the functions below (tested at a 0.01 significant level).

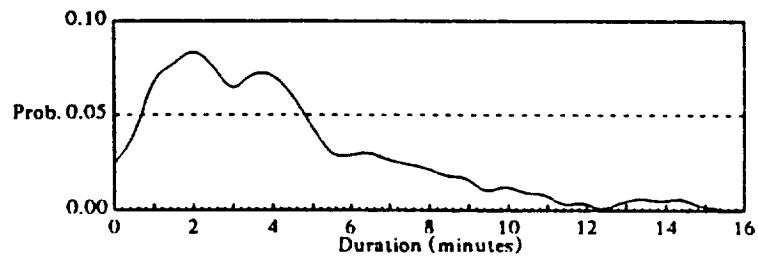
(1) DASD : $f(t) = 0.0375e^{-0.15t} + 0.007e^{-0.1t} + 0.008635e^{-0.014562t} + 0.0001861e^{-0.0021377t}$
(2) SWE : $f(t) = 0.041181e^{-0.044518t} + 0.0002704e^{-0.0036075t}$
(3) MULT: $f(t) = 0.004371(e^{-0.003817t} - e^{-0.0301092t})$



(a) DASD failure duration density



(b) SWE failure duration density



(c) MULT failure duration density

Figure 6. Failure duration densities

In summary, we have developed a state transition model which describes the normal and failure behavior of the system. Some key characteristics of the model are that the waiting time in some of the workload and in most failure states cannot be modeled as simple exponentials. Furthermore, the holding times from a given workload state to different failure state are dependent on the destinations. Thus, the overall system is modeled as a complex irreducible semi-Markov process.

5.3. Effect of Workload

Now that we have an overall model, we show the usage of this model to examine those states which are crucial from a failure viewpoint. For the computations in this section we consider the state OFFL of figure 3 as recurrent making the overall model ergodic. We then compute the steady state probability of being in a specified workload state and making a transition to a specified failure state. Table 5 shows the probabilities of a failure occurring at various load levels. In this table, "time" refers to the mean holding time to the selected state. An important relation between failure probability and holding time in a workload state is seen in this table. The failure probabilities appear to be not only a function of resource usage [6], but also related to the length of the holding time in a resource usage state. For example, in Table 5(a), the probability of a channel failure is

almost the same for two different CPU loads, 0.96 and 0.54. The mean holding time before a channel error occurs at the lower load is larger than that at the higher load, i.e. 1304.96 seconds versus 668.18 seconds. When the holding times are similar, however, (or increasing with increased usage), the failure probabilities do increase with increasing resource usage. A similar phenomenon also exists for the I/O workload (see Table 5(b)). Thus, not only does a higher workload result in a higher failure probability (for similar holding times), but the failure rate also increases with increased holding time in a particular state. In other words, the failure rate appears to be a function of the absolute amount of resource consumed in a given state, be it through increased workload and/or increased holding times. An explanation for this apparent "wear out" phenomenon is not clear (since a large majority of our errors are transient), but it certainly calls into further question the validity of the frequently used constant failure rate assumption in reliability modeling.

CPU	Failure state								Total
	CHAN		SWE		DASD		MULT		
Load	Time	Prob	Time	Prob	Time	Prob	Time	Prob	Prob
0.96	668.18	0.0011	1609.71	0.0786	1218.62	0.1296	1641.20	0.0285	0.2377
0.72	596.28	0.0032	1118.12	0.0492	971.62	0.0990	757.09	0.0146	0.1661
0.54	1304.96	0.0010	1507.92	0.0471	1070.10	0.0489	722.26	0.0052	0.1027

Time - in seconds.

(a). CPU workload

DASD	Failure state								Total
	CHAN		SWE		DASD		MULT		
Load	Time	Prob	Time	Prob	Time	Prob	Time	Prob	Prob
96.20	0	0	256.23	0.0022	434.54	0.0162	578.97	0.0046	0.023
67.34	898.35	0.0049	1243.35	0.0987	1170.84	0.1840	928.12	0.0262	0.3138
41.59	4522.67	0.0214	1516.92	0.0875	1148.18	0.1198	1286.67	0.0234	0.2521

Time - in seconds.

(b). DASD workload

Table 5. Holding time and transition probabilities to failure states

6. Performability Analysis

In this section we use the workload/failure/recovery model to evaluate the performability of the system. Reward functions are used to depict the performance degradation due to failures and also due to different types of recovery procedures. Toward this end, we define a reward rate for each state of the workload/failure/recovery model. We propose the reward rate, r_i (per unit time):

$$r_i = \begin{cases} \dfrac{s_i}{s_i + e_i} & \text{if } i \in S_N \cup S_F \\ 0 & \text{if } i \in S_R \end{cases}$$

where, the s_i and e_i are the service rate and the error rate in state i, respectively. Thus one unit of reward is given for each unit of time when the process stays in the good states S_N. The penalty paid depends on the number of errors generated by a failure. With an increasing number of errors the penalty per unit time unit increases, and accordingly, the reward rate decreases. Zero reward is assigned to recovery states. Based on this proposal, reward rates for the failure states are as shown in Table 6.

The reward rate of the modeled system at time t is a random variable $X(t)$. Therefore the expected reward rate $E[X(t)]$ can be evaluated as:

State	DASD	SWE	CHAN	MULT
r_i	0.5708	0.2736	0.9946	0.2777

Table 6. Reward rates for failure states

$$E[X(t)] = \sum_i p_i(t) r_i \ .$$

The cumulative reward by time t can be derived from $Y(t) = \int_0^t X(\sigma) d\sigma$. Therefore, the expect cumulative reward is given by [3]:

$$E[Y(t)] = E\left[\int_0^t X(\sigma) d\sigma\right] = \sum_i r_i \int_0^t p_i(\sigma) d\sigma \ .$$

where $p_i(t)$ is the probability of being in state i at time t. In order to solve for $p_i(t)$ and henc other measures, we convert the semi-Markov process into a Markov chain using the method c stages [4, 19]. Thus the state probability vector $P(t) = (....p_i(t)....)$ can be computed by solving th set of differential equations of the form:

$$\frac{d}{dt} P(t) = P(t) Q$$

where Q is transition rate matrix of the Markov chain [18]. In order to study the impact of different types of failures, the irreducible semi-Markov process was converted to one with absorbing states in the following manner. Analyses were performed on the resulting Markov reward model of the system using SHARPE (the Symbolic Hierarchical Automated Reliability and Performance Evaluator)[4] developed at Duke University:

a) with OFFL as the absorbing state (OFFL),
b) with MULT and OFFL as the absorbing states (MULT),
c) with SWE, MULT and OFFL as the absorbing states (SWE),
d) with DASD, MULT and OFFL as the absorbing states (DASD), and
e) with DASD, SWE, MULT and OFFL as the absorbing states (ALL).

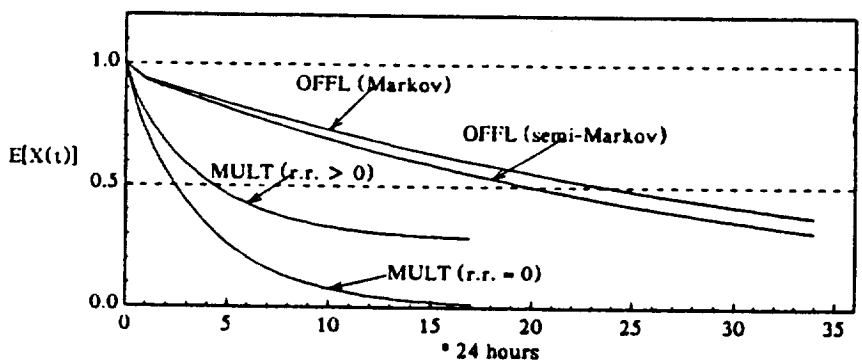
In case (a) we assess system performability in which all but off-line failures are not recovered from. In case (b) we discontinue recovering from multiple failures. Since multiple failures happen much more frequently than OFFL, we expect to measure the impact of SWE and DASD on the reward to a MULT failure. In case (c) we not only stop recovering from multiple and off-line failures but we also stop the recovery from software failures. In case (d) we recover from SWE failures but stop recovery from DASD failures. Finally, in case (e) we do not recover from any failures. We compare these scenarios first using the expected instantaneous reward rate $E[X(t)]$, then using the time- averaged expected accumulated reward $E[Y(t)]/t$. In all but case (a) and (e) we consider two variations: when a state such as DASD (MULT or SWE) is made absorbing, we can either let the reward rate in such a state be non-zero or we can set its reward rate to zero. The impact of the zero assignment is that upon reaching the absorbing state, the system continues to operate in a degraded mode. In the latter case, we conservatively assume that the system stops functioning when it reaches the absorbing state(s).

In Figure 7(a), we plot $E[X(t)]$ for cases (a) and (b). In the latter case we use two different assumptions for the reward rate for the MULT state (b1: r_{MULT}=0.27777, b2: r_{MULT}=0). We also

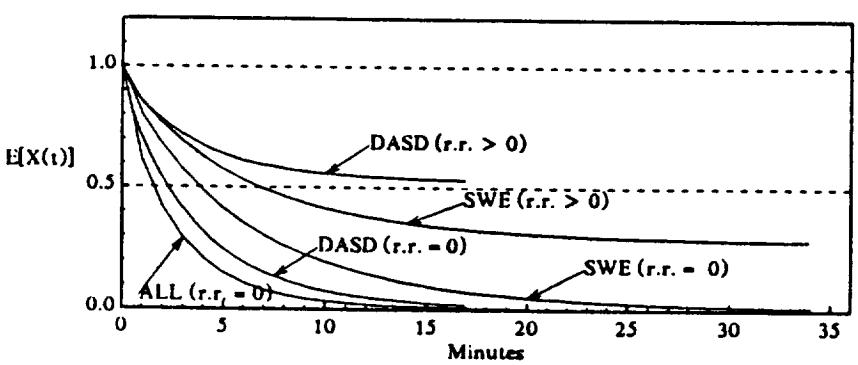
[4] SHARPE is a modeling tool . It provides several model types ranging from reliability block diagrams to complex semi Markov models [19].

plot E[X(t)] for case (a) with the assumption that all states have exponentially distributed holding times. We note that such a Markovian assumption leads to an overestimation of the system's capability to perform useful work. We also note that not recovering from multiple failures considerably degrades the system's performability. Furthermore, changing from case b1 to the conservative case b2 further reduces the system's effectiveness drastically.

In Figure 7(b), we plot the E[X(t)] for cases c, d and e. In each case, except case e, we have two versions with reward rates for absorbing states being non-zero and zero, respectively. Note that not recovering from SWE failures degrades system effectiveness considerably compared with the effect of not recovering from DASD failures, provided we assume that absorbing states continues to provide service in a degraded mode. On the other hand, if we assume that absorbing states are system failure states, i.e., zero reward rates for absorbing states, then not recovering from DASD failures is more severe than not recovering from SWE failures. This behavior is explained by the fact that the reward rate in the DASD state is about twice that in the SWE state. Figures 8(a) and 8(b) are the counterparts of Figures 7(a) and 7(b) where the measure plotted is E[Y(t)]/t rather than E[X(t)]. The trends are similar.



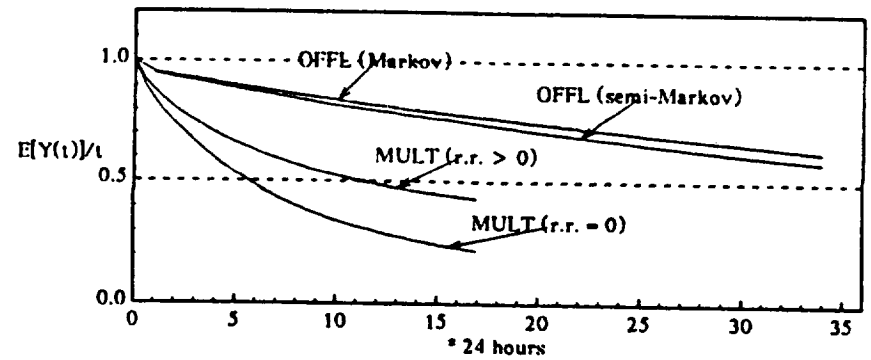
(a)



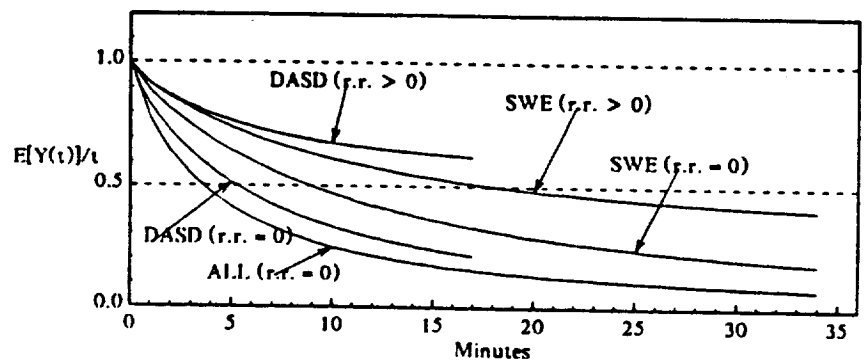
(b)

(r.r. : reward rate)

Figure 7. The expected reward rate, E[X(t)]



(a)



(b)

(r.r. : reward rate)

Figure 8. The time-averaged accumulated reward, $E[Y(t)]/t$

Finally, in Figure 9, we plot the distribution function of $Y(\infty)$, the accumulated reward until system failure, for two cases: Markov versus semi-Markov. Both assume that the OFFL state is the only absorbing state. Once again we note that the Markovian assumption implies an overestimation of the system's performability.

7. Conclusion

In this study, we have proposed a methodology to construct a model of resource usage, failure and recovery in a computer system, using real data from a production system. The semi-Markov model obtained is capable of reflecting both the normal and failure behavior of our measured system. The errors are classified into various types, based on the components involved. Both hardware and software errors are considered, and the interaction between the system components (hardware and software) is reflected in a multiple failure model. The proposed reward measure allows us to predict the performability of the system based on the service and error rates. It is suggested that other production systems be similarly analyzed so that a body of realistic data on computer failure and recovery models is available.

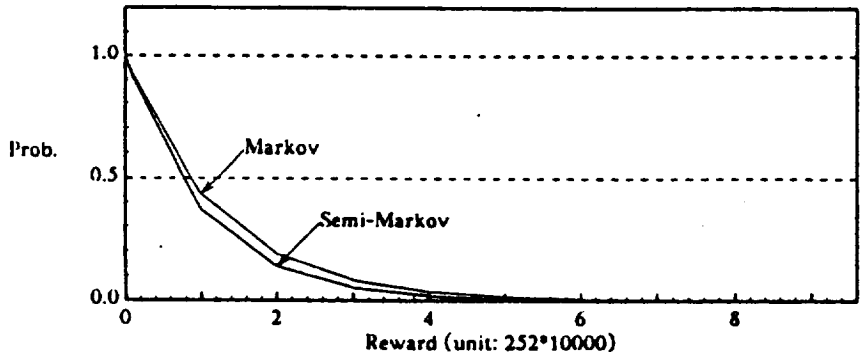


Figure 9. Distribution of accumulated reward until system failure

Acknowledgment

The authors would like to thank John Gerardi and the members of Reliability, Availability and Serviceability Group at IBM-Poughkeepsie for access to the data and for their valuable comments and suggestions. Thanks are also due to Dr. W.C. Carter for valuable discussions during the initial phase of this work. We also thank L.T. Young for his careful proofreading of the draft of this manuscript.

Reference

[1] Geist, R.M. and Trivedi, K., "Ultrahigh Reliability Prediction for Fault-Tolerant Computer Systems," *IEEE Transactions on Computers*, pp. 1118-1127, December 1983.

[2] Meyer, J.F., "Closed-Form Solutions of Performability," *IEEE Transactions on Computers*, pp. 648-657, July 1982.

[3] Smith, R.M., Trivedi, K.S., and Nicola, V.F., "The Analysis of Computer Systems Using Markov Reward Processes," 1987.

[4] Trivedi, K.S., *Probability & Statistics with Reliability, Queuing, and ComputerScience Applications*. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1982.

[5] Schoen, O., "On a Class of Integrated Performance/Reliability Models based on Queuing Networks," in *Proceedings of the 16th International Symposium on Fault-Tolerant Computing*, Vienna, Austria, pp. 90-95, July 1-4, 1986.

[6] Iyer, R.K., Rossetti, D.J., and Hsueh, M.C., "Measurement and Modeling of Computer Reliability as Affected by System Activity," *ACM Transactions on Computer Systems*, vol. 4, no. 3, pp. 214-237, August, 1986.

[7] Musa, J., "The Measurement and Management of Sofware Reliability," *IEEE Proceedings*, vol. 68, pp. 1131-1143, September 1980.

[8] Castillo, X. and Siewiorek, D.P., "A Workload Dependent Software Reliability Prediction Model," in *Proceedings of the 12th International Symposium on Fault-Tolerant Computing*, Santa Monica, California, pp. 279-285, June 22-24, 1982.

[9] Velardi, P. and Iyer, R. K., "A Study of Software Failure and Recovery in the MVS Operating System," *IEEE Transation on Computers*, vol. c-33, no. 6, pp. 564-568, June, 1984.

[10] Iyer, R. K. and Velardi, P., "Hardware-Related Software Errors: Measurement and Analysis," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 2, pp. 223-231, February 1985.

[11] Beounes, C and Laprie, J.C., "Dependability Evaluation of Complex Computer Systems: Stochastic Petri Net Modeling," in *Proceedings of the 15th International Symposium on Fault-Tolerant Computing*, Ann Arbor, Michigan, pp. 364-369, June 19-21, 1985.

[12] Castillo, X., *A Compatible Hardware/Software Reliability Prediction Model*, PhD Thesis, Carnegie-Mellon University, July, 1981.

[13] IBM Corperation, "MVS Architecture Resource Measurement Facility (RMF) Reference and User's Guide," *IBM publications*, vol. LC28-1138-1.

[14] Ferrari, D., Serazzi, G., and Zeigner, A., *Measurement and Tuning of Computer Systems*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1981.

[15] Spath, H. *Cluster Analysis Algorithms*. West Sussex, England: Ellis Horwood Ltd., 1980.

[16] IBM Corperation, *Environmental Record Editing & Printing Program*. International Business Machines Corporation, 1984.

[17] Iyer, R.K., Young, L.T., and Sridhar, V., "Recognition of Error Symptoms in Large Systems," in *Proceedings of the 1986 IEEE-ACM Fall Joint Computer Conference*, Dallas, Texas, pp. 797-806, November 2-6, 1986.

[18] Howard, Ronald A., *Dynamic Probabilistic Systems*. New York: John Wiley & Sons, Inc., 1971.

[19] Sahner, R.A. and Trivedi, K.S., "SHARPE: Symbolic Hierachical Automated Reliability and Performance Evaluator," in *User's Guide*, Durham, NC, September 1986.

Proceedings COMPSAC 87 Computer Software & Applications Conference, October 7-9, 1987, Tokyo, Japan. pp. 354-360

REV. ANN. 89A10935

A MEASUREMENT-BASED MODEL OF SOFTWARE RELIABILITY IN A PRODUCTION ENVIRONMENT

M.C. Hsueh and R.K. Iyer

Computer Systems Group
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
Urbana, IL 61801

Abstract

In this paper we build a semi-Markov model to describe the software error and recovery process in a large mainframe system. The model is based on low-level error data from the MVS operating system running on an IBM 3081 machine. The semi-Markov model developed provides a quantification of system error characteristics and the interaction between different types of errors. As an example, we provide a detailed model and analysis of multiple errors, which constitute approximately 17% of all software errors and result in considerable recovery overhead.

1. Introduction

The problem of modeling software reliability during the development, debugging and validation phases of the software cycle is a well researched area. However, there are few studies which model software error and recovery processes in a fully operational production environment. The difficulties are partly due to the fact that, unlike computer hardware, which is reasonably modularized, each software system can have its own peculiar characteristics. At this stage, it is extremely valuable to develop a comprehensive model quantifying the software error and recovery processes in a production system using real data. In addition to providing useful information on how and when errors occur in the real world, this process provides the quantification of the interaction among different types of errors; an important result for developing analytical models.

In this paper a state-transition model to describe the software error and recovery processes in a complex operating system is described. Measurements were made on an MVS (Multiple Virtual Storage) system running on an IBM 3081 mainframe. Time-stamped low level error and recovery data from MVS, collected during the normal operation of the system, formed the basis for developing the model. The semi-Markov model developed from the real data provides a quantification of the system error characteristics and also gives an insight into the interaction between the various software error and recovery processes occurring during normal system operation.

1.1. Related Research

Most software reliability models usually refer to the development, debugging and testing phases of the software [1,2] and [3,4]. Few of these models have been applied to the operational phase of the software. In [5] and [6], software errors in an operating environment are studied. Both studies found that at least 60% of of system errors were software related. There has been little explicit study of hardware/software reliability. In [6], software errors related to hardware problems in the operating system are analyzed and it is shown that errors in the hardware/software interface are often fatal. In [7], a resource-usage/reliability model was developed from real data and it was seen that about 36% of the detected errors (not necessarily system failures) were related to software problems.

With the exception of software reliability growth models, which have been validated with real data, there are few, if any, models of software reliability in an operational environment. Exceptions are the hardware and software model discussed in [8] and a measurement-based model of workload dependent errors discussed in [9]. Both, however, only describe the external behavior of the system and do not provide insight into component-level behavior.

It is therefore highly instructive to develop a detailed model based on low-level error data from a production system. Toward this end, in the following sections we construct a error/recovery model for the MVS operating system. Software problems of differing severity are modeled. Multiple errors are also considered and the effect of on-line recovery routines is taken into account.

In the next section, we describe the error data and the recovery characterization. An overall model is developed in Section 3, and in Section 4 we quantify key model characteristics. Finally, in Section 5, we summarize the results and highlight the important conclusions of this research.

2. Error and Recovery Characterization

In this section the collection and characterization of the software error and error recovery data are discussed. A state-transition diagram is developed to describe the different error and recovery states. This allows us to determine the serverity of errors and effectiveness of recovery.

2.1. Error Characterization

Error data based on different causes were collected. Information on software errors is automatically logged by an operating system module. Details of the logging mechanism are described in [10]. Due to the manner in which errors are detected in a system, it is possible that a single fault may manifest itself as more than one error (an error burst). Depending on the activity at the time of the error the different manifestations may not all be identical [11]. The recovery system usually treats these errors as isolated incidents. In order to address this problem, particularly to ensure that the analysis is not biased by error records relating to the same problem, two levels of data reduction [7] were performed. This reduction also allowed the grouping of similar types of errors. As a result the software errors were classified into eight classes.

This work was supported in part by NASA grant NAG-1-613, in part by the IBM Corporation and in part by the Joint Services Electronics Program (U.S. Army, U.S Navy, U.S. Air Force) under contract number N00014-84-C-0149.

PRECEDING PAGE BLANK NOT FILMED

These eight classes are called error events, since they may contain more than one error, and are defined as follows.

(1) Control (CTRL) — incidents indicating the invalid use of control statements and invalid supervisor calls

(2) Deadlocks (DLCK) — incidents indicating system or operator detected endless loop, endless wait state or violation of system or user-defined time limits

(3) I/O and Data Management (I/O) — incidents indicating problems occurred during I/O management or during the creation and processing of data sets

(4) Storage Management (SM) — incidents indicating errors in the storage allocation/de-allocation process or in virtual memory mapping

(5) Storage Exceptions (SE) — incidents indicating addressing of nonexistent or inaccessible memory locations

(6) Programming Exceptions (PE) — incidents indicating program errors other than storage exceptions

(7) Others (OTHR) — incidents indicating that problems occurred which do not fit the above categories

(8) Multiple Errors (MULT) — incidents indicating more than one type of the errors listed above

Table 1 lists the frequencies of the different types of software error events defined above. The table shows that more than one half (52.5%) of software errors were I/O and data management errors and another 11.4% of the errors were storage management errors. A significant percentage (17.4%) of errors were classified as multiple errors and are specifically modeled in the following sub-section.

Table 1. Frequency of software errors.

Type of Errors	Frequency	Percent
Control	213	7.72
Deadlock	23	0.84
I/O & Data Management	1448	52.50
Program Exceptions	65	2.43
Storage Exceptions	149	5.40
Storage Management	313	11.35
Others	66	2.32
Multiple Error	481	17.44
Total	2758	100.00

2.2. Multiple Errors

A multiple error most often is due to a single fault that has non-identical manifestations provoked by different types of system activity. Since the manifestations are not identical, recovery may be complex. Figure 1 shows the state-transition diagram of a multiple error developed from the data. The transition probability from state i to state j, p_{ij}, is estimated from the measured data using:

$$p_{ij} = \frac{\text{observed number of transitions from state } i \text{ to state } j}{\text{observed number of transitions from state } i}.$$

This figure not only illustrates the possible interactions among different software errors but also provides detailed information on the occurrence of transitions. For example, if a program exception error (PE) occurs, there is about a 63% chance that a storage exception error (SE) will follow. Further, there is more than a 50% chance that one storage error will be

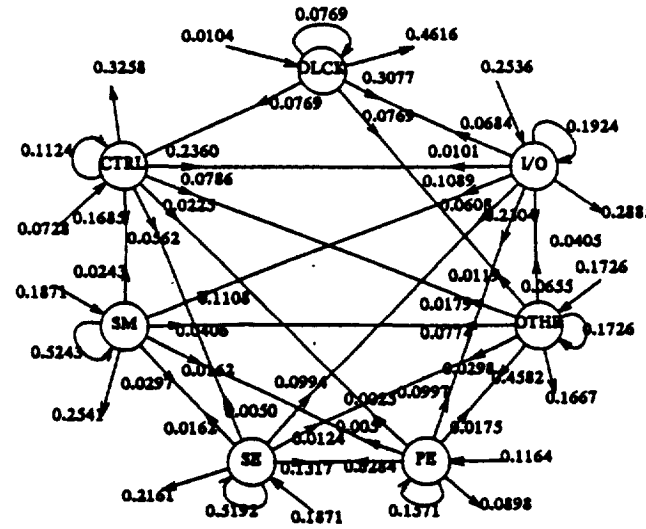


Figure 1. State-transition diagram for a multiple error

followed by another error of the same type (52% for storage management and also for storage exception). If we only focus on those transitions with significant probabilities (i.e., higher than 0.1), the number of states in Figure 1 can be reduced to five. The state-transition diagram for these active states is illustrated in Figure 2. Notice that a cyclic path is formed by the I/O and data management (I/O) along with the two different types of exception states (program exception and storage exception).

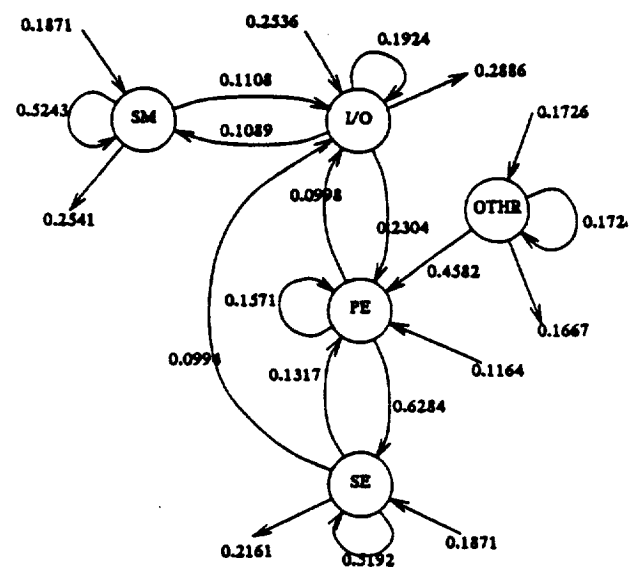


Figure 2. Reduced state-transition diagram of a multiple error

ORIGINAL PAGE IS
OF POOR QUALITY

2.3. Recovery Modeling

Recovery in MVS is designed as a means by which the system can prevent a total loss. Whenever a program is abnormally interrupted due to the detection of an error, the Supervisor gets control. If the problem is such that further processing could degrade the system or destroy data, the Supervisor gives control to the Recovery Termination Manager (RTM). If a recovery routine is available for the problem program, RTM gives control to this routine before deciding to terminate the program.

The purpose of a recovery routine is to free the resources kept by the failing program (if any), to locate the error, and to request either a continuation of the termination process or a retry. Recovery routines are generally provided to cover critical MVS functions. It is however, the responsibility of the installation (or of the user) to write a recovery routine for other programs.

More than one recovery routine can be specified for the same program: if the latest recovery routine asks for a termination of the program, the RTM can give control to another recovery (if provided). This process is called "percolation." The percolation process ends if either a routine issues a valid retry request or no more routines are available. In the latter case, the program and its related subtasks are terminated. If a valid retry is requested, a retry routine restores a valid status using the information supplied by the recovery routine(s) and gives control to the program. In order for a retry to be valid, the system should verify that there is no risk of error-recurrence and that the retry address is properly specified. An error may have four possible effects.

(1) Retry — The system successfully recovered and returned control to the problem program.

(2) Task Termination — The program and its related subtasks are terminated, but the system is not affected.

(3) Job Termination — The job in control at the time of the error is aborted.

(4) System Damage — The job or task in control at the time of the error was critical for system continuation. Thus, job/task termination resulted in system failure.

Figure 3 illustrates the steps in the recovery process. It is clear that recovery can be as simple as a retry or more complex requiring several percolations before a retry. The problem can also be such that no retry or percolation is possible. Table 2 shows the percentage for these different types of situations. For example, for storage management errors, approximately 8% of the cases resulted in a direct retry, 84% involved some percolation and over 8% could not be percolated any further (i.e. jobs/task termination). The table shows that only in a small percentage of the cases was the problem un-recoverable (no-percolation).

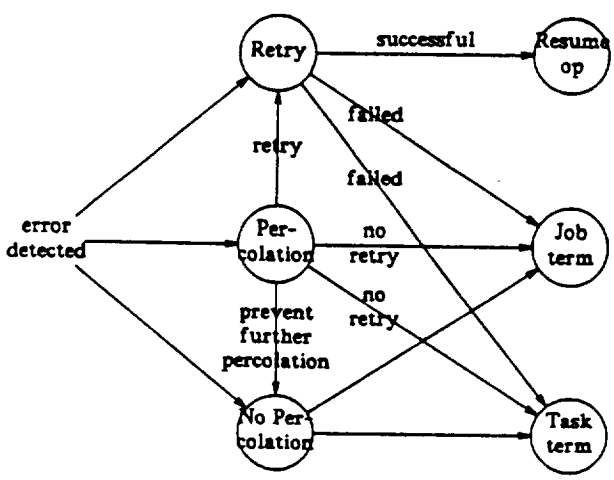


Figure 3. Flow of recovery process

Table 2. Percentages of different recovery attempts for a software error

Type of error	Retry (%)	Percolation (%)	No-Percolation (%)
CTRL	78.38	21.62	0.0
DLCK	2.78	97.22	0.0
I/O	93.49	6.51	0.0
PE	20.09	79.91	0.0
SE	28.09	71.91	0.0
SM	7.77	83.73	8.50
OTHR	14.89	85.11	0.0

3. Software Reliability Model

3.1. Overall Error/Recovery Model

In this section we combine the separate error and recovery models to construct a single overall model shown in Figure 4. Note that a state (normal) represents the normal

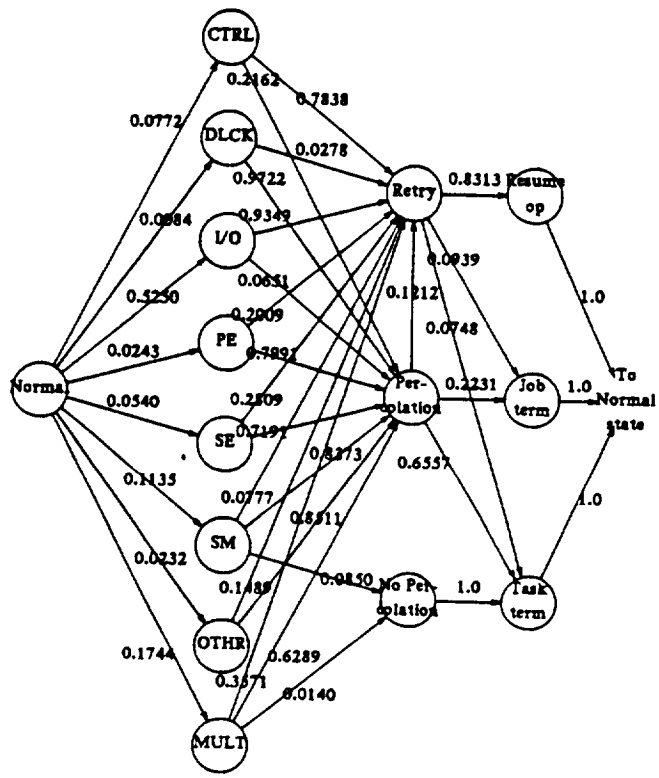


Figure 4. Software error/recovery model

system operation. The results of the recovery process are classified into three different states (resume op. task term and job term) to reflect the severity of errors. The model thus provides a complete overview of software error and recovery from an error occurrence to its recovery.

3.2. Waiting Time Distributions

Table 3 shows the characteristics of both normal and error states in terms of their waiting times[1]. Note that the duration of a single error is generally in the range of 20 — 40 seconds on the average, except for deadlock and "others". The table also shows that the errors not classified are relatively insignificant since their duration is less than 2 seconds. Program exceptions take twice as long as control errors (21 seconds versus 42 seconds). This is possibly due to the extensive software involvement in recovering from program exceptions. Figure 5 shows the density of waiting time in the normal operation state, i.e., the density of the time to error. This density could not be fitted to a simple exponential and was

Table 3. Characteristics of waiting times (seconds) in normal and error states

State	# of obs	Mean waiting time	Standard deviation	Std Error of mean
Normal	2757	10461.33	32735.04	623.44
CTRL	213	21.92	84.21	5.77
DLCK	23	4.72	22.61	4.72
I/O	1448	25.05	77.62	2.04
PE	65	42.23	92.98	11.53
SE	149	36.82	79.59	6.52
SM	313	33.40	95.01	5.37
OTHR	66	1.86	12.98	1.60
MULT	481	175.59	252.79	11.53

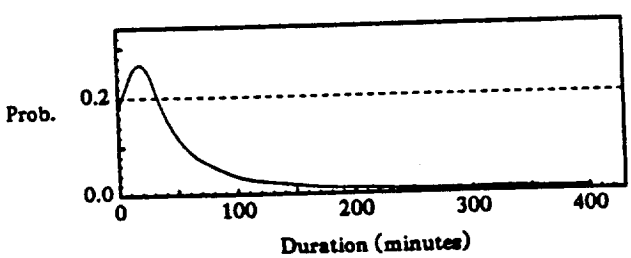


Figure 5. Time to error density

fitted to a multi-stage gamma density function $f(t)$.[2]

$$f(t) = \sum_{i=1}^{n} a_i g(t; \alpha_i, s_i),$$

where $a_i \geq 0$, $\sum_{i=1}^{n} a_i = 1$, and n is the number of stages. The $g(t; \alpha, s)$ is a gamma density function (with s the distance shifting from the origin).

$$g(t; \alpha, s) = \begin{cases} 0 & t < s \\ \frac{1}{\Gamma(\alpha)}(t-s)^{\alpha-1} e^{-(t-s)} & t \geq s, \end{cases}$$

where $\Gamma(\alpha)$ is a gamma function. Hence, the density in Figure 5 so obtained has five stages, given by (tested using the Kolmogorov-Smirnov test [13] at a 0.01 significance level):

$$\begin{aligned} f(t) 0= {} & 0.748\ g(t;\ 2.1,\ -1) + 0.055\ g(t;\ 0.5,\ 0) \\ & + 0.069\ g(t;\ 3.5,\ 3) + 0.030\ g(t;\ 5.0,\ 8) \\ & + 0.098\ g(t;\ 5.0,\ 17). \end{aligned}$$

3.3. Recovery Time Distribution

For the purposes of evaluating the time for recovery, we assumed that each recovery mode takes a constant amount of time. The overall recovery time, i.e., the duration of an error event (or the waiting time in an error state), however was not constant, since an error event can involve more than one recovery attempt or may require more than one recovery routine. The recovery time was then computed as the time difference between the first and the last detected error caused by the same event. The duration of an error event was used to measure the effectiveness of recovery from this event and also the severity of the error.

Figure 6 shows the recovery time densities for three different types of errors: I/O and data management, storage management, and multiple errors. Note that none of these densities could be fitted by simple exponentials at an acceptable level of significance. Thus, they were fitted to phase-type exponential density functions [13].

$$f(t) = \sum_{i=1}^{n} a_i g_i(t),$$

where $a_i \geq 0$, $\sum_{i=1}^{n} a_i = 1$ and n is the number of phases. The g_i function can be an exponential, multi-stage hyperexponential, or muti-stage hypoexponential density function. The densities in Figure 6 were fitted to the following functions (tested using the Kolmogorov-Smirnov test at a 0.01 significance level).

(1) I/O: $f(t) = 0.07825e^{-0.08594t} + 0.000354e^{-0.003958t}$

(2) SM: $f(t) = 0.1642e^{-0.5t} + 0.030424e^{-0.053294t} + 0.0006634e^{-0.006586t}$

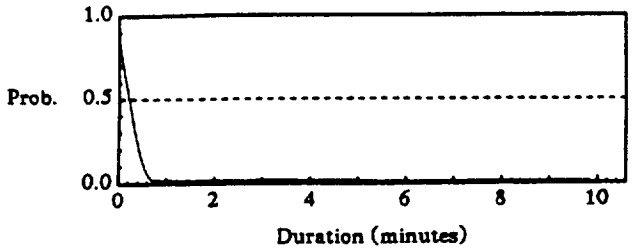
(3) MULT: $f(t) = 0.078e^{-0.4t} + 0.002426e^{-0.005866t} + 0.002163e^{-0.005525t}$

3.4. Summary

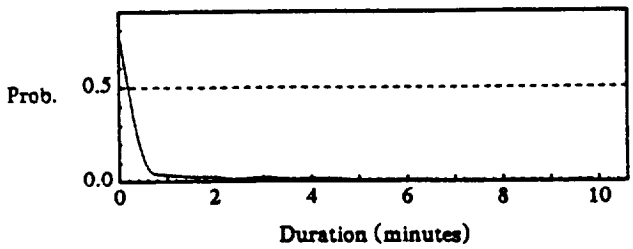
In summary, the model developed explicitly quantifies the error and recovery process in the measured software. We note both the time to error and the recovery time distributions in several key states cannot be modeled as simple exponentials. Hence the overall process is modeled as a semi-Markov process.

[1] The waiting time for state i is the time that the process spends in state i before making a transition [12].

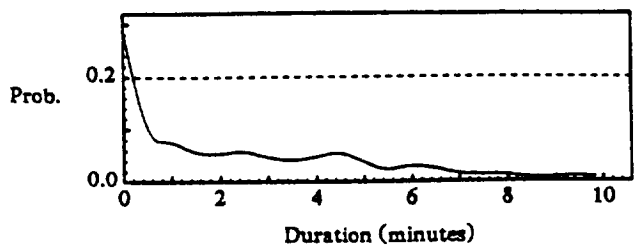
[2] We also tried to fit Figure 5 by using phase-type exponential. However, the fitted phase-type exponential gives us larger error than the multi-stage gamma density.



(a) Density for I/O and Data Management state



(b) Density for Storage Management State



(c) Density for multiple error state

Figure 6. Recovery time (error duration) densities

Further, the semi-Markov process is irreducible with the resume op (resume operation) state, the job term (job termination) state, and the task term (task termination) state being recurrent.

In the next section, we analyze the overall model to determine key software error characteristics. The mean time between different types of errors is evaluated along with model characteristics such as the occupancy probability of key error states.

4. Model Behavior

4.1. General Characteristics

By solving the semi-Markov model, we discover that the measured software system made a transition, on the average, every 43 minutes and 22 seconds. Table 4 lists the mean time between different software errors (i.e., mean time between errors) and Table 5 shows the mean recurrence time for recovery processes. By examining the mean recurrence time for I/O and MULT errors from Table 4 and comparing them with the mean waiting times in Table 3, we find that although the I/O errors occur about 3 times as often as the multiple errors, the system spends nearly 6 times longer in recovering from a multiple error (25 seconds for I/O errors versus 175.6 seconds for multiple errors). This is because recovery from a multiple error involves several different types of recovery attempts. In addition, 63% of the multiple errors invoke percolation compared with the fact that 94% of the I/O errors recovered through retry (see Figure 4).

Table 4. Mean time between errors

Error state	Frequency count	Percent (%)	MTBE (hour)
CTRL	213	7.72	37.83
DLCK	23	0.84	351.58
I/O	1448	52.50	5.56
PE	65	2.43	120.15
SE	149	5.40	54.08
SM	313	11.35	25.73
OTHR	66	2.32	125.84
MULT	481	17.44	16.75

Table 5. Mean recurrence time of recovery

Type of recovery	MTBR (hour)
Retry	4.25
Percolation	8.55
No-Percolation	241.43

4.2. Model Probabilities

Given the irreducible semi-Markov model of Figure 4, the following steady state probabilities were evaluated. The derivations of these measures are given in [12].

(1) transition probability (π_j) — given that the process is now making a transition, the probability that the transition is to state j

(2) occupancy probability (Φ_j) — at any instant of time the probability that the process occupies state j

(3) entry probability (e_j) — at any instant, given that the process is entering a state, the probability that the process enters state j

(4) mean recurrence time ($\bar{\Theta}_j$) — mean recurrence time of state j

The model characteristics are summarized in Table 6. A dashed line in this table indicates a negligible value (less than 0.00001 probability). From the occupancy probability, Φ, of the normal state in Table 6(a), we see that in about 99.5% of the time the software system is operating normally, i.e., only 0.5% of the time the system detects errors. This indicates that the reliability of the measured software system can be as high as 0.995. The table also shows that, of all possible transitions made, 24.73% are to an error state (obtained by summing all the π's for all the error states) and another 25.79% are to a recovery state. Since it was seen earlier that a transition occurs every 43 minutes, we estimate that a software error is detected, on the average, every 3 hours. From Table 6(b), we notice that although an error is detected almost every 3 hours, a successful recovery (i.e. results in resume op) only occurs once every five hours, i.e., nearly 43% of the errors result in task/job termination.

Multiple-error events formed a significant category on their own. Since this type of event involves several error types and results in considerable overhead, it is analyzed separately in the next section.

Table 6. Characteristics of software error/recovery model

Measure	Normal state	Error state							
		CTRL	DLCK	I/O	PE	SE	SM	OTHR	MULT
π	0.2474	0.0191	0.0020	0.1299	0.0060	0.0134	0.0281	0.0057	0.0431
Φ	0.9950	0.00016	-	0.00125	0.000098	0.000189	0.00036	-	0.002913

(a)

Measure	Recovery state			Result		
	Retry	Percolation	No-Percolation	Resume op	Task term	Job term
π_r	0.1704	0.0845	0.0030	0.1414	0.0712	0.0348
$\bar{\theta}$	4.25	8.55	241.43	5.11	10.16	20.74

* - in hour

(b)

Table 7. Characteristics of a multiple error

Measure	Normal state	Error state						
		CTRL	DLCK	I/O	PE	SE	SM	OTHR
π	0.1767	0.0327	0.0048	0.1451	0.1473	0.2957	0.1360	0.0617
Φ	0	0.0648	0.0130	0.3004	0.0837	0.2202	0.2717	0.0462
π_e	0.00568	0.00105	0.00015	0.00466	0.00473	0.00950	0.00437	0.00198
$\bar{\theta}$	0.0489	0.2647	1.8126	0.0596	0.0587	0.0292	0.0636	0.1401

* - in hour

4.3. Characteristics of A Multiple Error

In Section 2 we pointed out that about 17% of software errors were multiple errors. We also noticed that the multiple errors mostly consist of I/O, storage, or program errors. A strong connection between program and storage exception was seen in the occurrence of a multiple error. Table 7 lists the characteristics for a multiple error and was obtained by solving the semi-Markov model described in Figure 1 with a zero waiting time in the normal state (i.e., given a multiple error occurs). From Table 7 we see (from π, transition probability) that nearly 30% of the transitions are made to the storage exception state when the process enters a multiple error mode. Once in a multiple error mode, a storage exception error occurs every 1 minute and 45 seconds ($\bar{\theta}$ = 0.0292 hours in Table 7), while the average duration of multiple errors is about 2 minutes and 56 seconds [3] ($\bar{\theta}$ = 0.0489 hours, the recurrence time of the normal state). As soon as an entry into a multiple error is made, consecutive errors are detected almost every 31 seconds (by taking the reciprocal of the sum of all entry probabilities e in Table 7).

There are several interesting characteristics of multiple errors which can be derived from the model of Figure 1. For example, if we want to know the probability of a storage exception error given an I/O error, we can evaluate it by the multi-step transition probability to the SE state from the I/O state. This turns out to be very small, only 0.0076. However, we find that the probability of an I/O occurring given a SE occurs at any time instant, is as high as 0.668.[4]

5. Conclusion

In this study, we have developed a semi-Markov model to describe the error and recovery processes in the MVS system. The model is based on real error data collected during normal system operation. The semi-Markov model developed provides a quantification of system error characteristics and the interaction between different types of errors. As an example, we provide a detailed model and analysis of multiple errors, which constitute approximately 17% of all software errors and result in considerable overhead. It is suggested that other systems be similarly analyzed and modeled so that a wide range of realistic models of software reliability in an operating environment are available.

Acknowledgment

The authors would like to thank John Gerardi and the members of the Reliability, Availability and Serviceability Group at IBM-Poughkeepsie for access to the data and for many useful comments and suggestions. Thanks are also due to Dr. W.C. Carter for valuable discussions during the initial phase of this work. We also thank L.T. Young and P. Duba for their careful proofreading of the draft of this manuscript.

[3] Note that the average duration of a multiple error predicted here from the model is very close to the mean duration of a multiple error that is obtained from the real data listed in Table 3.

[4] This is partly due to the fact that for a semi-Markov process the transition probability at any time instant, $r_{ij} = \frac{\pi_i p_{ij} \bar{\tau}_{ij}}{\bar{\tau}}$, where π_i is the transition probability to state i at the instant when the process is in the steady state, p_{ij} is the conditional transition probability from state i to state j, $\bar{\tau}_{ij}$ is the mean holding time from state i to state j, and $\bar{\tau}$ is the mean holding time of the process.

References

[1] Goel, A.L., "Software Reliability Models: Assumptions and Applicability," *IEEE Transactions on Software Engineering*, vol. SE-11, pp. 1411-1423, December 1985.

[2] Yamada, S. and Osaki, S., "Software Reliability Growth Modeling," *IEEE Transactions on Software Engineering*, vol. SE-11, pp. 1431-1437, December 1985.

[3] Musa, J., "The Measurement and Management of Sofware Reliability," *IEEE Proceedings*, vol. 68, pp. 1131-1143, September 1980.

[4] Littlewood, B. "Theories of Software Reliability: How good are they and how can they be improved?," *IEEE Transactions on Software Engineering*, vol. SE-6, pp. 489-500, September 1980.

[5] Castillo, X. and Siewiorek, D.P., "A Workload Dependent Software Reliability Prediction Model," in *Proceedings of the 12th International Symposium on Fault-Tolerant Computing*, Santa Monica, California, pp. 279-285, June 22-24, 1982.

[6] Iyer, R. K. and Velardi, P., "Hardware-Related Software Errors: Measurement and Analysis," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 2, pp. 223-231, February 1985.

[7] Hsueh, M.C., Iyer, R.K., and Trivedi, K.S., "A Measurement-Based Performability Model for a Multiprocessor System," in *Proceedings of the 2nd International Workshop on Applied Mathematics and Performance/Reliability Models of Computer/Communication Systems*, Rome, Italy, May 25-29, 1987.

[8] Castillo, X., *A Compatible Hardware/Software Reliability Prediction Model*, PhD Thesis, Carnegie-Mellon University, July, 1981.

[9] Iyer, R.K., Rossetti, D.J., and Hsueh, M.C., "Measurement and Modeling of Computer Reliability as Affected by System Activity," *ACM Transactions on Computer Systems*, vol. 4, no. 3, pp. 214-237, August, 1986.

[10] IBM Corperation, *Environmental Record Editing & Printing Program*, International Business Machines Corporation, 1984.

[11] Iyer, R.K., Young, L.T., and Sridhar, V., "Recognition of Error Symptoms in Large Systems," in *Proceedings of the 1986 IEEE-ACM Fall Joint Computer Conference*, Dallas, Texas, pp. 797-806, November 2-6, 1986.

[12] Howard, Ronald A., *Dynamic Probabilistic Systems*, New York: John Wiley & Sons, Inc.,, 1971.

[13] Trivedi, K.S., *Probability & Statistics with Reliability, Queuing, and ComputerScience Applications*, Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1982.

1986 Proceedings Fall Joint Computer Conference
November 2-6, Dallas, TX.

N94-70426

Recognition of Error Symptoms in Large Systems

R.K. Iyer, L.T. Young and V. Sridhar

COMPUTER SYSTEMS GROUP
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign, Urbana, IL 61801

ABSTRACT

In this paper, a methodology for automatically detecting symptoms of frequently occurring errors in large computer systems is developed. The proposed symptom recognition methodology and its validation are based on probabilistic techniques. The technique is shown to work on real failure data from two CYBER systems at the University of Illinois. The methodology allows for the resolution between independent and dependent causes and, also quantifies a measure of the strength of relationship among the errors. Comparison made with failure/repair information obtained from field maintenance engineers shows that in 85% of the cases, the error symptoms recognized by our approach correspond to real system problems. Further, the remaining 15% although not directly supported by field data, were confirmed as valid problems. Some of these were shown to be persistent problems which otherwise would have been considered as minor transients and hence ignored.

Keywords: Error records, persistent errors, failure symptoms, probabilistic techniques, automatic recognition.

1. INTRODUCTION

The diagnosis of the causes of persisting errors in computer systems is difficult because the underlying faults are complex and may affect different parts of the system. The system usually detects the effects of the faults as many isolated errors. An incorrect diagnosis leads to improper recovery management which ultimately affects the integrity of the system. For the diagnosis to be effective, it is imperative that the system be able to relate errors occurring in different parts or different times.

When a service engineer tries to rectify a fault, he or she studies the error log for the period during which the fault occurred. Though the system may record the effect of a fault as many isolated incidents, the service engineer recognizes the different error entries as symptoms of the same error. This recognition of related error records is based on the service engineer's observance of some similarity, e.g., a recurring pattern among the error records.

This paper proposes a methodology for recognizing symptoms of severe errors in large systems. The goal of the methodology is to automate and formalize a process to relate errors occurring in different parts of a system. The method was developed after several years of study of system error logs and through close consultations with maintenance and field engineers from several manufacturers. The approach uses the system error rate to identify error records among which relationships can exist. Probabilistic techniques are then used to validate and quantify the strength of relationships among error records. The approach takes as input the raw error logs containing an entry for each error detected as an isolated event by a computer system, and produces as output symptoms which characterize persistent errors.

The methodology for automatically recognizing the error symptoms involves three steps:

(1) The recognition of error records among which relationships can exist,

(2) The determination of the existence and validity of relationships among these records and

(3) The quantification of the strength of a relationship if one is found.

The method is illustrated on error data collected from two large Cyber systems used on campus at the University of Illinois at Urbana-Champaign. The approach is simple in concept and hence easy to apply and simple to use. Since it is not based on any specialized characteristics of the Cyber systems or the error logs, it is expected to be applicable to other systems as well. Analysis of error data from two IBM systems is now in progress. Comparison made with failure/repair information obtained from field maintenance engineers shows that in 85% of the cases, the error symptoms recognized by our approach correspond to real system problems. Further, the remaining 15% although not directly supported by field data, were confirmed as valid problems by the engineers. Indeed, some of these were shown to be persistent problems which otherwise would have been considered as minor transients and hence ignored.

1.1 Related Research

Two studies which offer insight into the characterization of errors by their associated symptoms are [Velardi 84] and [Iyer 86]. Software errors in a production environment are analyzed in [Velardi 84], and permanent CPU errors and their relationship to workload are discussed in [Iyer 86].

CH2345-7/86/0000/0797$01.00 © 1986 IEEE

PRECEDING PAGE BLANK NOT FILMED

Research closely related to that described in this paper (symptom-based recognition) is that of [Tsao 83a] and [Sanders 85]. Tsao's work is motivated by the fact that a module exhibits a period of (potentially increasing) unreliability before final failure. The approach attempts to analyze trends in errors by first grouping errors occurring within a short period into Tuples. Heuristics, specific to DEC system error logs, are then used to determine if one tuple is similar to another [Tsao 83b]. As the author points out, two Tuples may be found to be identical by one algorithm, not identical by another. The result of the analysis is a small set of tuple types, each of which are expected to contain error records pointing to the same cause. The approach assumes that all error records in a Tuple are related.

In [Sanders 85], a technique which uses both complete and partial recurrences is proposed. The technique is similar in concept to that proposed by Tsao. Instead of using matching algorithms, a software table is used to quantify system organization. The software table is also used to capture both partial and complete recurrences as well as similarities among error groups. It was found that nearly 15% of the groups (similar to Tuples) contained a collection of unrelated error records, i.e., random groups.

An alternative approach to the above is called "structure-based" diagnosis. The development of a scheme based on the description of the structure and function of a system is described in [Davis 82]. Similar diagnostic schemes have also been proposed in [Shubin 82] and [Genesereth 82]. Another approach called "violated expectation" ([deKleer 76] and [Brown 81]) has also been proposed for troubleshooting. This approach looks for mismatches between the values expected from correct operation and those actually obtained. Whenever possible, the approach identifies a specific component that may be faulty. All these diagnostic systems have been shown to work on modules within a computer system. It is not clear how they will work when a whole computer system is taken into account.

In [Iyer 85], the analysis shows that the system is most vulnerable to errors in the hardware/software interface. It is not clear that a structure-based diagnosis can be fully successful under such circumstances. A structure-based diagnostic system may diagnose the hardware fault in a limited domain, but the chances are that complex faults (e.g. hardware-related-software errors) may be detected as more than one error. In this context, it becomes important to study the relationship among errors that occur within a short time.

In our opinion, both the symptom-based and structure-based approaches are still in their infancy. Hence, further research in both areas, particularly practical studies, needs to be done before either of them can generally be implemented.

The approach proposed in this paper is geared toward determining the symptoms of persistent errors in large systems by probabilistically relating the different manifestations of the same problem. The approach is naturally geared toward differentiating between transient and intermittent errors. To the authors' knowledge, there is no methodology that is able to do this at present. Further, the approach can also evaluate a diagnosis by analyzing tl diagnoses made in the past, i.e., the goodness of syste diagnostics. A methodology that quantifies the sympton associated with related errors by determining the relatio ship among detected errors is essential for this purpos The development of such an approach is discussed in tl following sections.

2. SYSTEM CONFIGURATION AND ERROR MEASUREMENT

2.1 System Configuration

The system studied consisted of two Control Da CYBER 170 systems maintained by the Computer Servic Office at the University of Illinois at Urbana-Champaig The two machines, a model 174 and a model 175, are co pled by their disk system. The two machines run indepe dently but share resources such as tape drives, etc. A interlock table is maintained for use by both machines prevent deadlock arising from conflicts over share resources. Details of the system configuration and tl error detection are given in the Appendix.

2.2 Recognition of Error Groups

A cursory analysis of the data showed duplica entries within a short time period describing the san error condition. It was decided to coalesce error recor that report the same conditions (i.e., same type of err and same machine state) to avoid multiple records refe ring to the same error. This process is referred to as *err clustering*. Details of the clustering on the Cyber syster are given in [Sridhar 85]. A visual examination of tl clustered data showed the existence of sets of cluste occurring within a short time interval. The close tin proximity among some clusters means a substanti increase in the system error rate during that period. T high error rate introduces a suspicion that the erro occurring during the high error rate period may be relate i.e., different errors may be due to a single cause, to mul ple but related causes, or to multiple and independe causes.

The high error rate periods referred to above a called error groups, and are formed by grouping all er clusters occurring within a small time interval of ea other. (This interval was chosen to be fifteen minute Analysis showed that the results were largely insensiti to variation in this time interval between 5 minutes anc hour. It will be seen in section 4 that relationships amo error records not captured by this choice of time inter will be captured during subsequent analysis.[1] The prima difference between a cluster and a group is that clust contain only occurrences of the same error (same er type and machine state), whereas groups cont occurrences of different errors (different error type machine state).

[1] The choice of the interval is analogous to choosing a starting point in merical optimization. A poor starting point will mean that the analysis will t a longer time.

As an illustration of the value of grouping, consider the series of observations in Figure 1. Looked upon singly, these would appear to be separate occurrences of different single bit memory errors. Yet, when the errors are combined into an error group, apparently all of them occur within bank 4. Also, the syndrome, or the problem bit, is the same for all the observations. This points to a problem within bank 4, and not to three isolated problems occurring within three different memory locations. Specifically, it refers to a problem that affects one particular bit of the word — which means the problem is even further isolated to the module. The probable causes now are confined to those that affect only one module — such as a faulty driver or path. In general, of course, the relationships will not be so obvious, hence the need for a probabilistic methodology proposed in the next section.

In summary, we start with the error log file and at the end of preprocessing we have reduced isolated errors to sets of clearly demarcated, periods of high error rate. These sets of error records are called error groups. Error groups identify periods of high unreliability in the system. The frequency of error groups demonstrates the need to relate errors occurring in different parts or times, i.e., global diagnosis. The tbf of the system increases considerably when groups are treated as single events. Thus, clearly, if the system were able to relate the errors in these groups it would be much more reliable. The determination of the existence and strength of relationships among error records in a group is described in the next section.

3. RECOGNITION OF ERROR PATTERNS

Errors in computer systems occur at various times and in many different locations. Often, the most severe errors are hard to diagnose because of the varied and seemingly unrelated symptoms associated with them as they occur in different locations and at different times. A successful diagnostic technique should be able to relate these errors in much the same way as a service engineer, through past experience, identifies a set of error records as the symptoms associated with a particular problem. The goal of this research was to automate and formalize a process to relate errors occurring in different parts of a system.

This section describes the key aspects of such a methodology, and the succeeding section gives examples illustrating the use of this approach. The objectives of the symptom recognition strategy are to start with the error log file, consisting of entries for all the errors detected as *isolated* errors by the system, and to produce as output symptoms associated with related errors that are being diagnosed as isolated and unrelated errors.

The symptom recognition methodology involves three steps:

(1) recognizing candidates among which relationships can exist,

(2) checking to see if a relationship actually exists among these candidates, and

(3) then estimating a measure of strength of the evaluated relationship (if a relationship does exist).

The first step, i.e., recognition of potential candidates for further analysis, is the time-based grouping described in the section 2.2. The second and third steps which are now discussed are recursively used at three levels of validation. At the first level, the relationship among the error records within an error group is evaluated; i.e., we determine whether there is a valid probabilistic relationship among the records in a group or whether they are simply a collection of unrelated records. The second level looks for inter-group relationships because the same cause(s) can give rise to multiple error groups within a short period. The third level looks for overall relationships in the entire data. Each level of analysis increases the resolution of the data and adds more certainty to the results. At each level, a measure of strength of the evaluated relationship is quantified.

The next subsection discusses the probabilistic validation procedure; the subsequent sections discuss the use of this procedure at different levels of analysis.

3.1 Probabilistic Validation Criterion

In this subsection, the technique used recursively to validate and measure the strength of relationships among the error records at various levels is described.

This technique examines the different records between which a relationship may exist and determines whether a valid relationship is possible among them on a probabilistic basis, based on the past data. In formal terms consider the probability space of n error records $A_1, A_2, \cdots, A_n$. Let $P(A_1), P(A_2), \cdots P(A_n)$ be their respective individual probabilities calculated from the data, i.e.,

GRPNO	TIME	NPTS	REC-TYPE	ERR	SYN	QUAD	CSU	BANK	CHIP	ERR-TYPE
129	23SEP:14:38:18	3	0003	PARITY	52	1	0	4	2	MEM
129	23SEP:14:38:18	1	0003	PARITY	52	1	0	4	3	MEM
129	23SEP:14:38:[illegible]	1	0003	PARITY	52	1	0	4	0	MEM
129	23SEP:14:38:18	2	0003	PARITY	52	1	0	4	3	MEM
129	23SEP:14:38:18	2	0003	PARITY	52	1	0	4	2	MEM
129	23SEP:14:38:18	1	0003	PARITY	52	1	0	4	0	MEM

Figure 1: Sample of Grouped Data

(template)	(time)	(error description)			(associated device)	(exact location)			
Record Type	Timestamp	Error Type	Error Code	Syndrome	channel	disk	chassis	quadrant	bank

Figure 2: Error Log Entry for Memory Error

$$P(A_i) = \frac{\#\ of\ occur.\ of\ A_i}{\sum_{i=1}^{i=n} \#\ of\ occur.\ of\ A_i}$$

Then, $P(A_1) * P(A_2) * \cdots * P(A_n)$ is the probability that $A_1, A_2, \cdots, A_n$ will occur together, assuming they are independent. Further, let $P(A_1, A_2, \cdots, A_n)$ be the joint probability of occurrence computed from all the joint occurrences of $A_1, A_2, \cdots, A_n$ in the error log, i.e.,

$$P(A_1, \cdots, A_n) = \frac{\#\ of\ joint\ occur.\ of\ (A_1, \cdots, A_n)}{\sum_{i=1}^{i=n} \#\ of\ occur.\ of\ A_i}$$

If

$$P(A_1) * \cdots * P(A_n) < P(A_1, \cdots, A_n) \qquad (3.1)$$

then it is reasonable to assume that the joint occurrence of $A_1, \cdots, A_n$ is not random. A measure of the strength (S) of the evaluated relationship is given by the ratio

$$S = \frac{P(A_1, \cdots, A_n)}{P(A_1) * \cdots * P(A_n)}$$

What $A_1, A_2, \cdots, A_n$ stand for depends on the level of analysis:

(1) When we are attempting to validate the relationship among the error records in an error group, $A_1, \cdots, A_n$ stand for the individual error records within the error group

(2) When we are studying the relationship among the error groups in an event, $A_1, \cdots, A_n$ stand for each individual error group. Since the existence and strength of relationships are evaluated probabilistically, the approach is not constrained to specific architectures or system configurations.

3.2 Validation at Group level : Inter-Record Relationships

Recall that error groups represent high error rate periods during the operation of the system. The high error rate introduces a suspicion that the error records within an error group may be related. It is, of course, possible that the rise in error rate is just a random incident. To distinguish between these two possibilities, error records in an error group are analyzed in three different ways to determine if the records in an error group have a valid relationship.

Complete Analysis

The first is referred to as a *complete analysis.* T probabilistic validation procedure discussed in the previ subsection is applied to all the records in an error gro i.e., the individual and the joint probabilities of all error records in a group are computed from the data. If the error records in an error group are found to be statis cally related, no further analysis is done on that er group.

Subset-Based Analysis

If error records in a group are found not to be rela through a complete analysis described above, then it m be that subsets of the original error group have valid re tionships. Different combinations of error records ir group are then analyzed to determine the existence of va statistical relationships among them. For example, if th are three records (A, B, C) in an error group, the subs (A, B), (A, C) and (B, C) are analyzed and Equation (3 is used to determine the relationship among the er records in these subsets. This approach is referred to a *subset-based* analysis.

Truncated Analysis

If no relationship is found with complete or subs based analysis, a third analysis based on truncated reco is carried out. In this step, a few fields of the error reco are masked from the analysis. This is best illustra using the format in Figure 2, which refers to the log o memory error.

The fields chassis, quadrant, bank and chip descr the location of error. It may be that a statistically va relationship exists at the bank level (say among a f banks) and that the chip identity of the error records m thwart the recognition of the valid bank level relatic ship.[2] Thus, when the chip identity is dropped, we attempting to determine the existence of statistically va relationships at the bank level. Note that when analysis is carried out with truncated records, m records can enter the probability space. Thus, the fi result can be substantially different from the origi result.

[2] When the chip identity is also taken into account, we are essentially ing to determine if errors occurring in different chips within a short perio statistically related or not.

[3] The same masking analysis may be carried out by starting from highest level of resolution, i.e., the last field of the record. The chip identi masked and the analysis is carried out; then the bank identity is masked an on until a relationship is found or until all the fields are masked. Thus algori will require more run time. The higher resolution offered by this analys achieved by subsequent analysis.

The fields to be masked are determined as follows: starting from the lowest level of resolution, i.e., the beginning of error records, the records in a group are scanned until the first field wherein they are dissimilar is found. All the fields beyond this field are masked from the analysis. For example, two records with the format shown in Figure 2 may have identical fields up to the chassis. The quadrant is the first field with non-identical values. Then the fields bank and chip are masked from the analysis.[3]

This three-tier analysis eliminates error groups consisting of records with no relationships, i.e., random errors, from further analysis. The importance of this analysis is underscored by the fact that in our analysis (shown in section 4) over 25% of the groups were rejected as random groups. The validation of subsets eliminates stray records present among related error records in an error group, and the validation of truncated records captures non-obvious relationships among error records. Thus, the original error groups consisting of records among which relationships *can* exist is refined to error groups consisting of records among which relationships *do* exist.

In some cases, the stray record does not invalidate a group. It merely reduces the strength of the evaluated relationship. Consider three records in a group, say A, B and C. A and B may be related and C may be the stray record. The strength of the relationship computed by analyzing A and B may be much higher than that obtained by analyzing A, B and C. It is possible to perform the three-tier analysis in all the cases and accept only the relationship with the highest strength. Such an analysis is, however, very computer-intensive and, therefore, was not performed here. Stray records are ultimately eliminated in subsequent analysis.

Relationships can exist across error groups; i.e., a single cause can give rise to a persistent error and thus foster multiple error groups within a short time. Therefore, it becomes necessary to examine the validated groups for inter-group relationships. The methodology to recognize inter-group relationships is described in the following subsections.

3.3 Validation at Event Level : Inter-Group Relationships

The persistence of a single cause(s) and the possibility of stray error records among related error records in an error group motivate the second level of analysis, i.e., recognition of related error groups and the elimination of stray error records from error groups.[4]

To analyze the relationship among error groups, it becomes necessary to introduce the concepts of events and symptoms, and symptom sets. An event is defined as the collection of error groups occurring within twenty-four hours and having at least two error records in common. A symptom is defined as a collection of statistically related, non-redundant error records that are common to most of the groups in an event. A symptom set associated with an event is defined as a collection of symptoms that best describe the event. Symptom sets are recognized by analyzing the error groups in an event.

[4] Noise is a representative term for stray error records.

When an event has just one error group, the statistically related error records within that error group form the symptom set associated with that event. When there is more than one error group in an event, the symptom set is formed by intersecting the error groups within the event. The error groups are taken N groups at a time, N-1 groups at a time, and so on down to (N/2), or to the next higher integer number of groups. The intersection of such different combinations of error groups yields different symptoms that are found in these groups. It also eliminates any stray records that occur in a few of the error groups in an event.

Once the symptom set associated with an event is derived, Equation (3.1) is used to check for the relationship among the different symptoms. If only one symptoms is extracted from an event, the event is probably due to a single cause. When more than one symptom is extracted from an event, the probabilistic validation procedure is used to determine if the different symptoms are caused by related or independent causes. For example, if two symptoms, say A and B, are extracted from an event, and if the symptoms satisfy Equation (3.1), i.e.,

$$P(A) * P(B) < P(A,B)$$

then the event is due to multiple but related causes, and the ratio

$$P(A,B) / (P(A) * P(B))$$

quantifies the strength of the relationship between A and B. If

$$P(A) * P(B) > P(A,B),$$

the event is caused by multiple and independent causes.

If two independent causes exist and persist simultaneously,[5] the extracted symptoms will consist of confusing sets of error records. This is because all or most of the groups in the event contain records referring to both causes. When such groups are intersected, symptoms associated with both are extracted as though they were related causes. The records due to one cause are strays with respect to the other cause, and some technique is necessary to discriminate between the two. A higher level of analysis, necessary to deal with such occurrences, is described in the following section.

3.4 Validation at Super-Event Level : Inter-Event Relationships

The possibility of the presence of stray records in all of the error groups in an event, and the need to capture relationships missed by the choice of twenty-four hours as the time interval while forming events, motivate the next and the last level of analysis, i.e., determination of inter-event relationships. Further, when the events resulting from the same cause(s) across the entire data are analyzed, the stray records occurring in one particular event become visible. This allows for the resolution between independent and dependent (related) failures.

Three simple rules are used to recognize related events and to form *super events*. Two events are grouped into a super event if they satisfy any one of the following

[5] Such conditions occur rarely, but they do occur.

criteria:

(1) They have at least one symptom in common

(2) A symptom of one event is a sub or super set of at least one symptom of another event

(3) If they are single-group events, then they have at least two records in common.

Importantly, there is no time restriction when forming super events.

Next, a set of super symptoms, which are collections of symptoms found in common among two or more events in a super event, is generated. The procedure is exactly the same as for events except that instead of taking groups, we take events described by the symptom sets. The intersection operation among the events yields symptoms found in common among these events. The non-null result of each intersection operation forms a super symptom for the associated super event.

Some super events have just one super symptom associated with them, while others have more than one associated with them. To determine whether the different super symptoms associated with a super event are related, (i.e., whether the super event has a single cause) we use the probabilistic validation procedure described in Section 3.1. If the different super symptoms are not related, then the super event is due to multiple and independent causes.

When a super event is due to multiple and independent causes, the number of such distinct independent causes may be determined as follows: the super symptoms are considered in different combinations, taken N-1 super symptoms at a time, and so on down to 2 super symptoms. The probabilistic validation procedure is applied to each such combination. This analysis determines the relationship within a particular subset of super symptoms associated with a super event. If the subset is found related, it is counted as one cause. The analysis is continued until all independent causes associated with a super event are found.

Recall that the goal of this paper is to automate the process by which a service engineer relates errors occurring in different parts of a computer system. The super symptoms derived from the error log file not only consist of sets of related error records but also quantify the strength of their relationships. Whereas the service engineer usually scans the error log pertaining to the short period during which the error occurred, recognizes related error records and makes a local judgement, the methodology scans all the errors that occur in a system and is capable of making a global judgement. The application of this technique to real data is discussed in the next section.

3.5 Summary

The methodology described in this section establishes that it is possible to build a system that will relate errors occurring in different parts/times in a computer system or when two unrelated causes occur simultaneously in a system. It has been shown that the proposed methodology allows for resolution between them. Some causes persist for days or even months. While such causes persist, other independent errors can also occur in a system.

Three levels of validation are used in a recu fashion. In the first, all groups (high error rate per are examined to determine if they contain related rec Next, the validated groups are examined for inter-g relations (events). Finally, the events across the e date are examined for inter-event relationship.[6]

The symptom sets obtained using this method imply that if one error within a symptom set occurs, related errors within the symptom set can be expect occur. The strength of the relationship says how like is that related errors will follow.

The number of levels of analysis can be varie higher resolution. That is, if a system is highly error-p and a grouping of errors occurring within fifteen mi leads to coalescing of unrelated errors, then this gro time interval may be reduced to, say, five minutes or The next section illustrates various facets of this app by applying it to determine the error symptoms i CYBER system.

4. EXAMPLES OF ERROR SYMPTOMS

This section illustrates the various stages of analysis involved in the automatic recognition of symptoms from error log data. A summary of the ana is given first. Then three examples are provided, illus ing the three levels of analysis, namely, the group, and super event levels. Some features of the pro methodology not illustrated by the first example are trated in the last two examples.

4.1 Summary of the Analysis

Table 1 sums up the number of groups, events and events derived from the error logs of Cyber 174 and 175 machines. Note that 67.1% of the Cyber-174 g and 27.5% of the Cyber-175 groups were rejected as dom groups. Most of the super events in both the mac were due to single or related causes. In all, there are f distinct causes that result in severe errors on the C 174 and thirty-three causes on the Cyber-175.

TABLE 1:
FREQUENCY OF GROUPS, EVENTS AND SUPER EV

	Cyber 174	Cyber 17
# of groups	85	142
# of groups rejected	57	39
# of events		
due to single cause	19	53
due to multiple causes	0	2
# of super events		
due to single cause	15	33
due to multiple causes	0	2

[6] Note that each level of analysis adds to the information gathered ous levels, i.e. a group validated at the group level cannot be rejected quent analysis but can find other related groups.

ORIGINAL PAGE IS
OF POOR QUALITY

There are 85 error groups in the Cyber-174 data and 142 groups in the Cyber-175 data. These numbers quantify the periods of highly unreliable operation during the operation of the computer system. During these high error rate periods, many of the system resources may have been utilized in recovery measures.

4.2 Cyber-174 Super Event with Multiple Super Symptoms

This section illustrates the various facets of the proposed methodology. This particular example is chosen because it illustrates all the important steps and the major problems involved in the automatic recognition of symptoms.

Figure 3 shows a super event consisting of four events. Here all events are single group events. As an example of analysis at the group level, we consider the group of error records in the second event, marked EVT 2. In this group, we find four types of records, i.e., those records with ERRTYPE/locations corresponding to E/..., S/0-0-7-1, S/1-0-1-2, and S/1-0-5-0. Using the method of Section 3.1, the strength of this group was 76.50. Since this strength is greater than 1.0, the records exhibit a valid relationship within this group, so subset-based and truncated record analyses are unnecessary. The strengths of the groups respective of the order that they appear in Figure 3 are 1.56, 76.50, 1.56, and 4.00. Analysis of symptoms at the event level is trivial in Figure 3, since each event has only one type of error symptom. An example which better illustrates event level analysis is discussed in the next section.

	RECTYPE	TIME	ERRTYPE	CHANNEL	SYNROME	QUADRANT	CHASIS	BANK	CHIP
	3	84May17 08:05:01	S	14	D0	1	0	1	2
EVT 1	3	84May17 08:07:23	S	14	25	1	0	5	0
	3	84May17 08:15:47	S	14	23	0	0	7	1
	3	84May20 11:24:59	E	14	23	.....			
	3	84May20 11:33:21	S	14	23	0	0	7	1
EVT 2	3	84May20 11:40:48	E	14	23	.....			
	3	84May20 11:58:14	S	14	D0	1	0	1	2
	3	84May20 11:58:14	S	14	25	1	0	5	0
	3	84May28 12:30:45	S	14	23	0	0	7	1
EVT 3	3	84May28 12:44:22	S	14	25	1	0	5	0
	3	84May28 12:49:12	S	14	23	0	0	7	1
	3	84May28 12:53:47	S	14	D0	1	0	1	2
	3	84Jun02 03:54:51	S	14	23	0	0	7	1
EVT 4	3	84Jun02 04:00:00	S	14	38	0	1	5	3
	3	84Jun02 04:15:54	E	14	23	.....			

Figure 3:
Example of Multiple Symptom Super Event on CYBER-174

Analysis at the super event level is performed by applying the probabilistic validation procedure to the set of super symptoms extracted from a super event. In Figure 3, the union of event symptoms yields the following super symptom set:

$$(1)\ (E/\cdots, S/0\text{-}0\text{-}7\text{-}1, S/1\text{-}0\text{-}1\text{-}2, S/1\text{-}0\text{-}5\text{-}0)$$

$$(2)\ (E/\cdots, S/0\text{-}0\text{-}7\text{-}1, S/0\text{-}1\text{-}5\text{-}3) \qquad (4.1)$$

By definition of a super event, these super symptoms can occur within only this super event. The strength of the entire super event by Equation (3.1) is 2.0, which indicates that the events are strongly related. Thus, this super event may be considered the result of a single cause. Had these super symptoms failed to satisfy Equation (3.1), we would have inferred that the super event was due to multiple causes.

It should be noted that the error records in this super event span a period of seventeen days. The cause appears to be an intermittent error which recurs, apparently affecting different components. That they are due to a single cause would not be immediately obvious from looking through the entire error log. This example clearly demonstrates the power of the proposed methodology to extract good quality symptoms associated with severe errors in computer systems.

4.3 Cyber-175 Super Event with Multiple Super Symptoms

Figure 4 shows part of a super event. This super event consists of ten events persisting over a period of the ten weeks from July 8 to September 18 of 1985. Three of these events are shown in Figure 4. The second event, marked EVT 2, consists of three groups, spanning a period of less than seven minutes each. In addition to occurring less than an hour apart and having at least two records in common with another group, these three groups had to be validated at the group level to be included in the same event. Using the measure of strength of relationship between records as described in Section 3.1, the value for the first group's locations (1-1-1-0 , 1-1-2-2 , 1-1-0-1 , 1-0-0-1) is 8.444. For the second group, the value for locations (1-1-1-0 , 1-1-2-2 , 0-1-7-2 , 1-1-3-3) is 12.901. And for the third group, the value for locations (1-1-1-0 , 1-0-0-1 , 0-1-7-2 , 1-1-3-3) is 10.411. These strengths clearly indicate the validity of the record relationships within the three groups.

Event Level Analysis

As an example of analysis at the event level, we consider the set of symptoms generated by the groups in EVT 2. Four symptoms are generated:

$$(1)\ (1\text{-}1\text{-}1\text{-}0)$$

$$(2)\ (1\text{-}1\text{-}1\text{-}0, 1\text{-}0\text{-}0\text{-}1)$$

$$(3)\ (1\text{-}1\text{-}1\text{-}0, 1\text{-}1\text{-}2\text{-}2)$$

$$(4)\ (1\text{-}1\text{-}1\text{-}0, 1\text{-}1\text{-}3\text{-}3, 0\text{-}1\text{-}7\text{-}2) \qquad (4.2)$$

By Equation (3.1) the strength of event EVT 2 is 32.16. This indicates a valid relationship between the three error groups in event EVT 2, i.e., they were all caused by the same problem. But this inference is only a local view. To get a global view, we look for related events across the entire data.

Super Event Level Analysis

For the super-event level analysis, other events with at least one symptom set in common are extracted from the data. Here we illustrate the procedure for the events shown in Figure 4. Events EVT 1 and EVT 3 are single-group events. Their symptom sets are simply the groups themselves. The super symptoms obtained are:

(1) (1–0–2–0 . 1–0–2–2 . 1–1–3–3)

(2) (1–1–1–0 . 1–1–3–3 . 0–1–7–2)

(3) (1–1–1–0 . 1–1–2–2 . 1–0–0–1) (4.3)

Note that the first three symptoms of Equation (4.2) have been absorbed into the third super symptom of Equation (4.3). This provides global support of the inference made at a local level that errors occurring in locations 1-1-1-0, 1-1-2-2, and 1-0-0-1 are due to the same cause.

The physical interpretation of these three super symptoms is that the occurrence of one error implies that another error is likely to follow. The strength of the relationship among the errors in a symptom set tells us how likely it is that the related error will follow. The probabilistic validation procedure quantifies a measure of the strength of the relationship among error records based on their joint and individual occurrences.

For example, the values of the measure of relationship strength between error records pertaining to the three super symptoms of Equation (4.3), respectively, are 5.37, 5.26, and 2.17. Thus, when an error occurs in location 1-1-1-0, errors are likely to occur in locations 0-1-7-2, 1-0-0-1, 1-1-2-2 and 1-1-3-3. Given that an error has occurred in location 1-1-1-0, the chances are that an error is more likely to occur in location 1-1-3-3 or 0-1-7-2 than in location 1-1-2-2 or 1-0-0-1. Thus, the symptom sets tell us about the other parts or other errors that are likely to occur, and the measure of strength of the relationship among the records in a symptom set tells us which parts are more likely to be affected from among all the parts that can be affected.

A practical benefit of the proposed methodology is that it makes service/maintenance engineers aware of all the severe or persisting errors in a system. Although most causes of error took the form of single, transient groups of errors, there were quite a few that resulted in numerous recurring groups of errors, spanning months of time. For example, 26.7% of the CYBER-175's super events were composed of multiple groups of errors, as were 36.4% of the CYBER-175's super events. In this context, it is interesting to note that the maintenance engineers were unaware of the persisting errors discussed in this section until we brought it to their attention. This was not with-

	RECTYPE	TIME	ERRTYPE	CHANNEL	ESTS	ESCMSE	SYNRO	CHPPAR	PPAN	QUARTS	DASI	CHANSK	BCHIP
EVT 1	3	84Sep05 04:01:33	S	14	0	5	58	0	0	1	0	2	2
	3	84Sep05 04:01:47	S	14	0	5	89	0	0	1	0	2	0
	3	84Sep05 04:02:15	S	14	0	5	3D	0	0	1	1	3	3
	3	84Sep05 04:02:28	S	14	0	5	58	0	0	1	0	2	2
	3	84Sep05 04:03:32	S	14	0	5	3D	0	0	1	1	3	3
EVT 2	3	84Sep13 06:00:09	S	14	0	5	54	0	0	1	1	2	2
	3	84Sep13 06:00:22	S	14	0	5	E9	0	0	1	1	0	1
	3	84Sep13 06:00:42	S	14	0	5	54	0	0	1	1	2	2
	3	84Sep13 06:01:31	S	14	0	5	23	0	0	1	1	1	0
	3	84Sep13 06:02:40	S	14	0	5	54	0	0	1	1	2	2
	3	84Sep13 06:03:05	S	14	0	5	23	0	0	1	1	1	0
	3	84Sep13 06:03:28	S	14	0	5	54	0	0	1	1	2	2
	3	84Sep13 06:05:11	S	14	0	5	8A	0	0	1	0	0	1
	3	84Sep13 06:05:40	S	14	0	5	54	0	0	1	1	2	2
	3	84Sep13 06:05:52	S	14	0	5	23	0	0	1	1	1	0
	3	84Sep13 06:05:58	S	14	0	5	8A	0	0	1	0	0	1
	3	84Sep13 07:00:02	S	14	0	5	3D	0	0	1	1	3	3
	3	84Sep13 07:01:40	S	14	0	5	54	0	0	1	1	2	2
	3	84Sep13 07:02:03	S	14	0	5	23	0	0	1	1	1	0
	3	84Sep13 07:02:40	S	14	0	5	3D	0	0	1	1	3	3
	3	84Sep13 07:02:44	S	14	0	5	07	0	0	0	1	7	2
	3	84Sep13 07:02:47	S	14	0	5	3D	0	0	1	1	3	3
	3	84Sep13 08:00:33	S	14	0	5	23	0	0	1	1	1	0
	3	84Sep13 08:01:09	S	14	0	5	3D	0	0	1	1	3	3
	3	84Sep13 08:01:15	S	14	0	5	8A	0	0	1	0	0	1
	3	84Sep13 08:04:12	S	14	0	5	23	0	0	1	1	1	0
	3	84Sep13 08:05:32	S	14	0	5	07	0	0	0	1	7	2
	3	84Sep13 08:06:12	S	14	0	5	3D	0	0	1	1	3	3
	3	84Sep13 08:06:12	S	14	0	5	23	0	0	1	1	1	0
	3	84Sep13 08:07:24	S	14	0	5	3D	0	0	1	1	3	3
EVT 3	3	84Sep18 05:01:53	S	14	0	5	8A	0	0	1	0	0	1
	3	84Sep18 05:03:26	S	14	0	5	23	0	0	1	1	1	0
	3	84Sep18 05:03:30	S	14	0	5	8A	0	0	1	0	0	1
	3	84Sep18 05:03:59	S	14	0	5	54	0	0	1	1	2	2
	3	84Sep18 05:05:06	S	14	0	5	23	0	0	1	1	1	0
	3	84Sep18 05:05:20	S	14	0	5	54	0	0	1	1	2	2

Figure 4:
Example of Multiple Symptom Super Event on CYBER-175

standing the fact that the same maintenance engineers went through the error log on alternate days of the week during the six-month period when these errors persisted. They, however, failed to notice them.

4.4 Super Event with Unrelated Super Symptoms

This section illustrates an example of a super event whose records appear to be strongly related at both the group and event levels but fail to show significant relationships at the super event level.

Figure 5 shows a super event constructed from CYBER-175 error records and consisting of two events. In the first event, marked EVT 1, there are two groups. Analysis of the records in each group yields strengths of

1.42 and 19.27 for the smaller and larger, respectively. In the second event, marked EVT 2, there are three groups. Their strengths, in chronological order, are 19.27, 5.23, and 1.42. Clearly, the records in all five groups are strongly related.

This strong relationship is preserved at the event level as well. The groups of EVT 1 are represented by a single symptom: (1-1-1-0, 1-0-2-0). EVT 2 had three symptoms and an event strength of 4.54:

(1) (1–0–2–0 , 0–0–2–1 , 1–1–7–2)

(2) (1–0–2–0 , 1–1–1–0)

(3) (1–0–2–0) (4.4)

This strength again indicates that the error records in EVT 2 are related to a single cause.

When analyzed at the super event level, however, we find that there is no longer enough statistical support to indicate that the super event of Figure 5 was due to a single cause. The super symptom set is found by taking the union of the symptoms generated by EVT 1 and EVT 2 and is simply the first two symptoms of Equation (4.4). The first super symptom appears independently four different times among all the events generated from the CYBER-175 data. The second super symptom appears eight times independently. By Equation (3.1), the strength of this super event is only 0.38. This low strength indicates that the super event consists of errors due to unrelated causes. The super symptoms occur independently so frequently that their joint appearance in the super event of Figure 5 cannot be considered significant.

Two specific procedure of the proposed methodology, namely, the subset based and truncated analyses, have not been illustrated in the examples discussed so far. Examples of these procedures can be found in a related technical report [Sridhar 85].

5. PERFORMANCE OF THE METHODOLOGY

As a final check on the performance of the methodology, we obtained independent corroboration of our results from the log of failures and repair maintained by the field engineers. Although this recording was not always complete, it did allow us to perform an independent check on our evaluation. The results of this check are summarized below.

In nearly 85% of the cases the engineers were directly able to confirm that our validated superevents corresponded to real system problems. The evaluation was made both on the basis of their experience and from their field maintenance logs. The confirm...d cases corresponded to single event superevents.

For the remaining 15% of the cases, although the engineers could not confirm "noticing" a valid problem, they agreed that (with the benefit of hindsight) that a problem had existed. However, the manifestation (an event) was not severe enough to be noticed by their analysis. One important case was the superevent consisting of ten events, discussed in section 4.2. This superevent

	RECTYPE	TIME	ERRTYPE	CHANNEL	SYNROME	CHPAR	PPAR	QUADRANT	CHASSIS	BANK	CHIP
	3	84May18 03:12:59	S	14	23	0	0	1	1	1	0
	3	84May18 03:14:01	S	14	89	0	0	1	0	2	0
	3	84May18 03:15:12	S	14	23	0	0	1	1	1	0
E	3	84May18 04:00:06	S	14	92	0	0	0	0	2	1
V	3	84May18 04:00:17	S	14	23	0	0	1	1	1	0
T	3	84May18 04:00:31	S	14	92	0	0	0	0	2	1
	3	84May18 04:01:09	S	14	A1	0	0	1	1	7	2
1	3	84May18 04:01:13	S	14	89	0	0	1	0	2	0
	3	84May18 04:01:24	S	14	92	0	0	0	0	2	1
	3	84May18 04:01:55	S	14	23	0	0	1	1	1	0
	3	84May18 04:02:30	S	14	92	0	0	0	0	2	1
	3	84May18 04:03:19	S	14	89	0	0	1	0	2	0
	3	84May19 05:00:21	S	14	92	0	0	0	0	2	1
	3	84May19 05:01:13	S	14	89	0	0	1	0	2	0
	3	84May19 05:01:53	S	14	A1	0	0	1	1	7	2
	3	84May19 05:02:31	S	14	92	0	0	0	0	2	1
	3	84May19 05:02:54	S	14	89	0	0	1	0	2	0
	3	84May19 05:02:58	S	14	92	0	0	0	0	2	1
	3	84May19 05:03:40	S	14	A1	0	0	1	1	7	2
	3	84May19 05:03:59	S	14	23	0	0	1	1	1	0
E											
V	3	84May19 06:00:09	S	14	92	0	0	0	0	2	1
T	3	84May19 06:00:10	S	14	A1	0	0	1	1	7	2
	3	84May19 06:00:11	S	14	89	0	0	1	0	2	0
2											
	3	84May19 07:01:04	S	14	23	0	0	1	1	1	0
	3	84May19 07:04:15	S	14	89	0	0	1	0	2	0
	3	84May19 07:04:30	S	14	23	0	0	1	1	1	0
	3	84May19 07:04:31	S	14	89	0	0	1	0	2	0
	3	84May19 07:04:52	S	14	23	0	0	1	1	1	0
	3	84May19 07:04:55	S	14	89	0	0	1	0	2	0
	3	84May19 07:07:11	S	14	23	0	0	1	1	1	0
	3	84May19 07:07:27	S	14	89	0	0	1	0	2	0
	3	84May19 07:07:48	S	14	23	0	0	1	1	1	0

Figure 5:
Example of Super Event Due to Unrelated Causes

spans ten weeks. Although, each event by itself was not severe enough to show up, when taken as a whole it was clear that a persistent problem had existed. It appears that as a result of our information corrective action was taken because this particular superevent does not exist in the data which has been gathered since then. In summary, we find that not only do our results agree with field findings but also we are able to relate problems over a long period of time which would otherwise go undetected.

The proposed methodology, thus, can be used to evaluate the goodness of any diagnostic system, whether primitive or sophisticated. This can be done by analyzing the relationship among the errors the diagnostic system detects as isolated. A very simple measure for the goodness of diagnosis of a system could be given by

$$1 - \frac{\textit{number of error entries coalescing into related errors}}{\textit{total number of error entries in the error log}}$$

6. CONCLUSION

This paper has developed a methodology for automatically detecting symptoms associated with persistent errors in computer systems. The methodology was shown to work on two large computer systems in a multisystem configuration. The recognition process is based on probabilistic techniques and is implemented in three levels. Each level adds confidence to the results. The power of the methodology to allow for the resolution between single and multiple but independent causes was also demonstrated. With multiple but overlapping symptoms, the methodology determines the number of distinct independent symptoms. The methodology quantifies the strength of the relationship among related errors. Research is now in progress to develop a suitable diagnostic system that will use the extracted error symptoms and automatically carry out diagnosis and recovery. This would not only facilitate better diagnosis and recovery management, but will also improve the overall reliability and performance of a system.

ACKNOWLEDGEMENTS

The authors thank George Badger, Director CSO, Warner Brigham, Sandy Moy and Milt Cloud at CSO for providing insight into the measured systems and access to the data. This work was supported in part by the Joint Services Electronics Program (U.S. Army, U.S. Navy, and U.S. Air Force) under contract number N00014-84-C-0149 and in part by IBM corporation.

APPENDIX
SYSTEM DESCRIPTION

This analysis uses detailed error data automatically collected over a six-month period by the operating system. As with other large systems, the CYBER machines maintain a log of a variety of normal and abnormal events — called the "system dayfile." The abnormal events are errors automatically detected by the system. Errors can originate in the peripheral processors, the central processor, main memory, or the disk subsystem. The errors are captured by hardware that checks the integrity of information being transmitted between physical elements in the system. When an error is detected, a system routine collects pertinent information from the hardware concerning the state of the machine at the time of the error and stores it in the dayfile. A decoded sample of the information logged is shown Figure A.1.

The fields of information include:

(1) record type: specifies the format of the rest of the record (RECTYPE)

(2) error code: specifies the nature of the error (ERR)

(3) time and date of error

(4) channel in use at the time of error (CHAN)

(5) equipment number of the device involved (EST)

The three major record types are:

(1) deadstarts (RECTYPE = 0001): these are logged for all scheduled and unscheduled system restarts. The two common deadstart levels are (a) deadstart level zero which denotes a complete halt of the machine with no job recovery possible, and (b) deadstart level three which implies that the integrity of the machine state is maintained and recovery is possible for most jobs.

(2) disk subsystem errors (RECTYPE = 0024): these included disk checkword errors, channel parity errors, and device contention errors.

(3) parity-related errors (RECTYPE = 0003): these include memory errors (single error correction, double error detection), peripheral processor errors, channel errors, extended core parity and central memory controller errors.

REFERENCES

[Brown 81] J. S. Brown, et. al., "Pedagogical and knowledge engr. techs. in the SOPHIE systems," Xerox CIS-14,1981.

[Davis 82] R. Davis, *et al*, "Diagnosis based on structure and function," AAAI-82, Pittsburgh, 1982, pp. 137-142.

[deKleer 76] "Local methods for localizing faults in electronic circuits," Vol. PAMI-4, No. 3, May 1982.

[Genesereth 82] M. Genesereth, "Diagnosis using hierarchical design models," AAAI-82, Pittsburg, 1982.

[Iyer 85] R. K. Iyer, P. Velardi, "Hardware-related software errors," *IEEE Tr. SWE*, vol. SE-11, Feb 1985.

[Iyer 86] R. K. Iyer, et. al., "Measurement and modeling of comp. rel. as affected by syst. activity," *ACM TOCS*, Aug 1986.

[Sanders 85] D. Sanders, M.S. thesis, ECE-UIUC, 1985.

[Shubin 82] H. Shubin and J. W. Ulrich, "IDT: An intelligent diagnostic tool," AAAI-82, Pittsburg, 1982.

[Sridhar 85] V. Sridhar, et. al., "Recognition of error symptoms in large systems," CSG-46, CSL, UIUC, 1985.

[Tsao 83a] M. M. Tsao, "Trend analysis and fault prediction," Tech. Rep. CMU-CS-83-130, Carnegie-Mellon University, 1983.

[Tsao 83b] M. M. Tsao and D. P. Siewiorek "Trend analysis on system error files" FTCS-13, 1983.

[Velardi 84] P. Velardi, R. K. Iyer, "A study of software failures and recovery in MVS," *IEEE Tr. Comp.*, vol. C-33, 1984.

OBS	TIME	REC-TYPE	ERR	CHAN	EST	SYN	POSIT	QUAD	CSU	BANK	CHIP	ERR-TYPE
1	19MAY84:2:06:12	0024	CHKWD	5	10	.	266051	.	.	.	.	.
2	19MAY84:5:00:21	0003	PARITY	.	.	92	.	0	0	2	1	MEM
3	19MAY84:5:00:57	0003	PARITY	.	.	92	.	0	0	2	1	MEM
4	19MAY84:5:01:13	0003	PARITY	.	.	89	.	1	0	2	0	MEM
5	19MAY84:5:01:53	0003	PARITY	.	.	A1	.	1	1	7	2	MEM
6	19MAY84:5:02:31	0003	PARITY	.	.	92	.	0	0	2	1	MEM
7	19MAY84:5:02:54	0003	PARITY	.	.	89	.	1	0	2	0	MEM
8	19MAY84:5:02:58	0003	PARITY	.	.	92	.	0	0	2	1	MEM
9	19MAY84:5:03:40	0003	PARITY	.	.	A1	.	1	1	7	2	MEM

Figure A.1: Sample From Error Log

Reconfigurable Tree Architectures Using Subtree Oriented Fault Tolerance

MATTHEW B. LOWRIE AND W. KENT FUCHS, MEMBER, IEEE

CMIT PREV. ANN. SA 126213

Abstract—**An approach to the design of reconfigurable tree architectures is presented in which spare processors are allocated at the leaves. The approach is unique in that spares are associated with subtrees and sharing of spares between these subtrees can occur. The Subtree Oriented Fault Tolerance (SOFT) approach is more reliable than previous approaches capable of tolerating link and switch failures for both single-chip and multichip tree implementations while reducing redundancy in terms of both spare processors and links. VLSI layout is $O(n)$ for binary trees and is directly extensible to N-ary trees and fault tolerance through performance degradation.**

Index Terms—**Fault tolerance, reconfiguration, reliability, spare processors, tree architectures.**

I. Introduction

A continually rising demand for high performance computation has created a need for highly concurrent computer architectures. One architecture which has received significant attention is the tree topology [1]–[3]. Tree architectures have an inherent ability to compute concurrently with typical communication times between the n processors being $O(\log n)$. However, as the number of processor nodes and communication links increases, the probability of single or multiple failures within structured concurrent architectures becomes unacceptably large. Consequently, recent interest has arisen in designing the ability to reconfigure concurrent architectures with one or more faults. Reconfigurability, which is one aspect of fault tolerance, is especially significant in tightly coupled tree architectures where the failure of a single link or processor can result in the subsequent loss of all communication with processors in the subtree below the faulty element.

One of the initial reconfigurable binary tree proposals was made by Hayes, who developed a procedure for constructing "optimal" 1-fault tolerant trees [4], which has since been extended by Kwan and Toida [5]. Raghavendra, Avižienis, and Ercegovac (RAE) [6], improved on these proposals by adding sufficient redundant lines in order to tolerate multiple failures. Link redundancy for binary trees in the RAE approach is as high as 200 percent, and VLSI layout may require $O(n \log n)$ area [7]. Hassan and Agarwal [7], recently presented a modular technique which allocates one spare to multilevel groups of processors. This scheme is conceptually similar to the RAE approach in that it dedicates every spare to one specific group of processors, but has the advantage of $O(n)$ layout and modularity for multichip architectures. Another proposal for reconfiguration which is applicable to tree architectures has been proposed by Rosenberg [8]. The approach requires a collinear layout with each node requiring access to a log n bus. Redundancy in terms of switching transistors is $O(\log n)$ for each node. The switching structure provides for efficient utilization of spare processors. However, fault tolerance for the communication lines and the switching transistors is not considered.

One of the important objectives in designing for reconfiguration is efficient utilization of spares. If the architecture has k spare processors, then the objective is to be able to tolerate any combination of k processor failures through reconfiguration. This should be accomplished with a reasonable increase in interconnect, manageable layout complexity for large numbers of processors, and a bounded number of pins per chip. In addition, failures in interconnect and switching structures should also be tolerable through reconfiguration.

A strategy for satisfying these objectives for binary tree architectures is presented in this paper. The approach places spare processors at the leaves of the tree and provides for considerable flexibility in reconfiguration through sharing of spares between adjacent subtrees. This strategy, which is referred to as Subtree Oriented Fault Tolerance (SOFT), utilizes a virtual displacement technique to reconfigure a spare processor into the tree. The capability of sharing spare processors between subtrees provides the SOFT approach with significantly higher reliability than previous techniques allowing for switch and link fault failures, where reliability is the probability that the tree is functional at a time t, given that it was fault free at time 0. In contrast to other proposals, SOFT is able to tolerate link and switch failures while reducing the number of redundant links between processors. For binary trees, the approach is shown to yield $O(n)$ layout. The architecture can be partitioned on separate chips for arbitrarily large trees, while providing fault tolerance for both on and off-chip connections. Fault tolerance through performance degradation is also possible with a SOFT design, as well as application to N-ary trees.

In Section II of this paper, the SOFT architecture is presented for binary trees. Considerations for implementing a SOFT binary tree are discussed, including the placement of

Manuscript received July 31, 1985; revised March 28, 1986 and January 21, 1987. This research was supported in part by the Microelectronics and Computer Technology Corporation (MCC) under a VLSI/CAD grant, and by the National Aeronautics and Space Administration (NASA) under Contract NASA NAG 1-613.

The authors are with the Computer Systems Group, Coordinated Science Laboratory, University of Illinois, Urbana, IL 61801.

IEEE Log Number 8715432.

0018-9340/87/1000-1172$01.00 © 1987 IEEE

PRECEDING PAGE BLANK NOT FILMED

spare processors and communication links. Section III provides a formal analysis of reconfigurability in SOFT binary trees in the presence of processor, switch, and link failures. In Section IV, comparisons between the reliability of SOFT and past reconfigurable designs are presented. Finally, Section V extends the SOFT concept to N-ary trees.

II. The Soft Design for Binary Trees

The SOFT approach to reconfigurable tree architectures employs spare processors at the leaves of the tree and additional links between processors to maintain a complete tree topology in the presence of multiple faulty processors and links. Faulty processors and links are bypassed by utilizing a simple switching structure, which allows a logical displacement of processors and links to occur from the failure to a spare at a leaf. At levels high in the tree, failure of a node results in bypassing the node, thereby allowing communication directly between the faulty node's father and one of its sons. Thus, the faulty node's son assumes the tasks allocated to its faulty father. Since the son is performing the tasks of its father, another processor must be found to assume the son's tasks. In a similar fashion, one of the son's sons assumes its responsibilities. This "logical displacement" continues until a spare is configured in at the leaves. A detailed discussion of SOFT reconfiguration is presented in Section III.

A. Terminology for Binary Trees

All trees are said to have $i + 1$ levels, with the root at level 0 and the leaves on level i. The term "upper levels" refers to levels 0 through $i - 1$. The root is labeled 1, the left son of any node (processor) n is labeled $2n$ and the right son is $2n + 1$. Two nodes are *adjacent* if they are connected by a nonredundant or redundant communication link. The *father* of a node n on level k, f_n, is the adjacent node on level $k - 1$. The *grandfather* of n, g_n, is the father of the father of n. Similarly, the son of a node n is son_n. f_l and son_l represent the father and son nodes of link l, respectively. The *brother* of a node n, b_n, is the single node having the same f_n. b_l refers to either node connected to a redundant link l. A *subtree* is defined as a tree with $< i$ levels, which is contained in the tree architecture such that the leaves of the subtree correspond to leaves in the complete tree. The *leftmost* (*rightmost*) *descendent* of a subtree is the node which can be found by following only left (right) descendents of the root of the subtree. The *cousin* of a node n, cous_n, is the leftmost (rightmost) descendent of the root on the same level if n is the rightmost (leftmost) descendent of the root. For all other n, cous_n is $n - (+)1$ if n is a left (right) son of its father. The *ancestor* of a node n on a level q, A_n^q, is the single node on level q which contains n in its subtree.

B. Allocation of Redundant Nodes and Links

The number of spare processors supported by the SOFT architecture is 2^c, where c is an integer: $0 \leqslant c \leqslant i - 1$. Algorithm 1 is an algorithm for positioning these spares. The redundant links required by the SOFT architecture are allocated as described by Algorithm 2. The SOFT binary tree of Fig. 1 was generated by Algorithms 1 and 2 with $i = 4$ and $c = 2$. A subtree with leaves $x + k2^{i_{SST}}$ to $x + (k + 1)2^{i_{SST}} -$

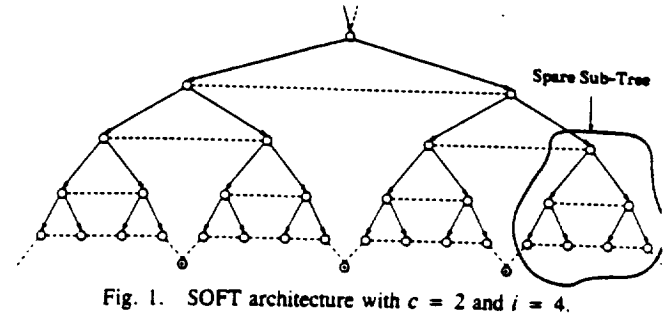


Fig. 1. SOFT architecture with $c = 2$ and $i = 4$.

1, where x is the leftmost leaf of the root and $0 \leqslant k \leqslant 2^c -$ is referred to as a Spare Subtree, or SST. Each SST has *associated spare* which is adjacent to its rightmost leaf. T spare adjacent to an SST's leftmost leaf is referred to as *nonassociated spare*. In contrast to X-tree or Hyperti structures [9]–[11], the SOFT topology is not a half-ri structure in which each level contains cousin connectic instead of the n to b_n connections utilized by SOFT.

Algorithm 1: Placement of Spares:

```
Begin
    i_SST := i - c;          {height of spare subtree}
    for k := 1 to 2^(i - i_SST) do begin
      x := leftmost leaf of root; {leftmost descendant c
        level i}
      n := x + k2^(i_SST) - 1
      add spare and connect as right son to n;
          {associated spare of SST k
          which has leaves n - 2^(i_SST) + 1 to n}
      connect spare to cous_n;
      end
    end.
```

Algorithm 2: Placement of Redundant Links:

```
Procedure Brother_connect (n :node)
    begin
        connect n to b_n;
        if level(n) < i      {n is not a leaf}
          then begin
            Brother_connect(left son of n);
            Brother_connect(left son of b_n);
            end
          else
            if not(adjacent_to_spare(n))
                then connect n to cous_n;
        end
begin
    Brother_connect(left son of root);
    end.
```

C. Redundancy Calculations

The calculations for percentage of spare redundancy are straightforward:

$$\text{percent node redundancy} = \frac{2^c}{2^{i+1} - 1}.$$

The number of redundant links is

$$\text{redundant links} = \left(\sum_{g=0}^{g=i-2} 2^g \right) + 2^i + 2^c + 1.$$

TABLE I
SOFT REDUNDANCY

% Node Redundancy	% Link Redundancy
≈25.0%	≈100%
≈12.5%	≈87.5%
≈6.25%	≈81.25%
$\frac{1}{2^{i+1}-1}$	≈75%

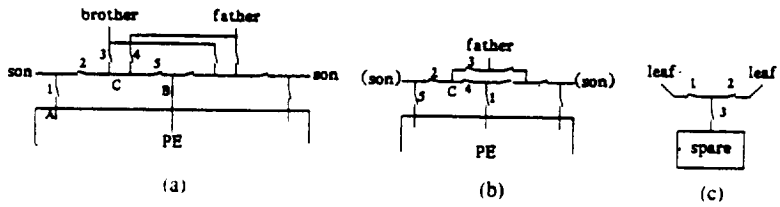


Fig. 2. SOFT switching in binary trees. (a) Upper level switching. (b) Leaf switching. (c) Spare switching.

Thus, the percentage of redundancy for links is

$$\text{percent link redundancy} = \frac{2^{i-1} + 2^{i} + 2^{c}}{2^{i-1} - 1}.$$

For large i, this is approximately 0.5 + 0.25 + (percent redundancy of spares). Table I enumerates the possible percentage node redundancies for large i and the corresponding percentage of link redundancy.

D. Implementing the SOFT Architecture

Implementing the SOFT strategy requires all processors to be identical. Most well-known algorithms for tree architectures utilize identical processors [1]–[3]. It is also assumed that I/O through the leaves is not required. This is not a necessary restriction, but simplifies presentation of the architecture. Most tree architectures also do not use I/O through the leaves [1]–[3]. In fact, the classical H-tree layout cannot accommodate I/O through the leaves for large trees.

1) Switching Scheme: The virtual processor displacement concept of SOFT reconfiguration is implemented with the switch structures of Fig. 2. Utilizing the switch structures, design of the processing elements is independent of the reconfiguration scheme and each processor can maintain a standard three input–output port design.

2) Multichip Trees: If an entire tree cannot fit onto a single chip or wafer, then the tree must be partitioned for chip allocation. The major partitioning consideration is imposed by pin limitations. From Fig. 1, it can be seen that at most one additional link per chip is required for chips containing no leaf processors, and fewer pins per chip for chips containing leaf processors.

3) VLSI Layout of SOFT Trees: To efficiently lay out a SOFT binary tree in VLSI, an adjustment in the general architecture is made in order to employ a variation of the classical $O(n)$ H-tree layout. The VLSI layout for a tree of $i \leq 4$ follows the layout of Fig. 3(a). The location of spares depends upon the percentage of spares allocated to the tree. As in Algorithm 1, the spares are located on the nonbrother redundant links, between the leaves. For trees with $i > 4$ the layout algorithm presented below results in area of $O(n)$. The result of Algorithm 3 with $i = 7$ is depicted in Fig. 3(b). The ellipses represent the five-level subtrees constructed in the first "for loop" of the algorithm.

Algorithm 3: VLSI Layout of Trees with $i > 4$.

```
begin
    for k := 1 to 2^(i-4) do
      z(k) := a five-level SOFT tree constructed by Algorithms 1 and 2;
    for k := 1 to 2^(i-5) do
      connect root(z(2*k)) to root(2*k - 1));
    construct main tree (m) as an i - 5 level nonredundant binary tree;
    apply Algorithm 2 to m omitting all cousin connections;
    for k := 1 to 2^(i-5)
      x := leftmost leaf of root;
      move to x + k - 1
      connect z(2*k) as left son;
      connect z(2*k - 1) as right son;
      end
    end.
```

Theorem 1: The SOFT layout, as described in Algorithm 3, for a binary tree of n leaves, has $O(n)$ area.

Proof: It is well known that the area of an H-tree is $O(n)$ [12]. The layout can be thought of as having $O(\sqrt{n})$ rows and columns. The layout produced by Algorithm 3 (as in Fig. 3) has at most one redundant link (additional row or column) parallel to each row or column of the $O(n)$ H-tree layout. Since each spare can be thought of as lying in a redundant link, the edge of the square corresponding to the layout of a SOFT tree is at most $O(k\sqrt{n})$, where k is a constant which, in the worst case is between 2 and 3. Since this is still $O(\sqrt{n})$, the area of the SOFT layout is $O(n)$. □

The modification to the general SOFT approach necessary for VLSI layout has the following implications.

1) While the maximum number of spares is ≈25 percent, the minimum is increased from an arbitrarily small percentage (1 per tree) to ≈3 percent (1 per five-level subtree).

2) Since the five-level subtrees do not share any connections among spares, SST's cannot "borrow" spares from neighbors not within that subtree. This does not, however, affect the reliability analysis presented in Section IV, which shows significant reliability enhancement is gained with the SOFT strategy.

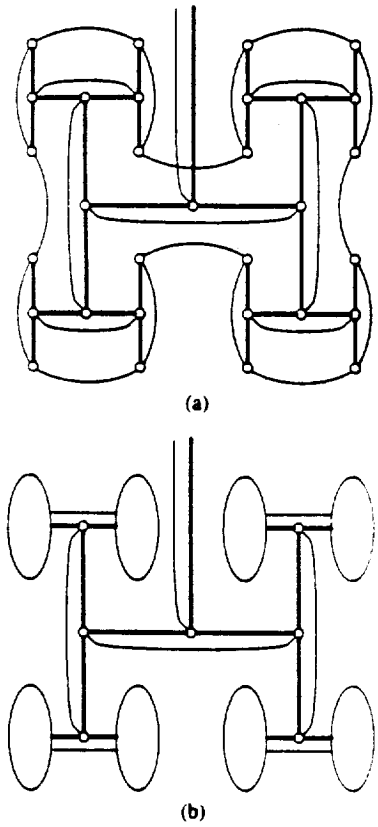


Fig. 3. VLSI layout of SOFT architecture.

III. Fault Tolerance in SOFT

Necessary and sufficient conditions are presented for reconfigurability in SOFT architectures. Reconfiguration for faulty processors is considered first, followed by an analysis of reconfiguration for link failures and switch failures. The fault model for PE's and links is functional in nature and includes any fault affecting the correct operation of the processor or the link under consideration. The fault model for switches consists of stuck-open or stuck-closed faults.

A. Tolerance of Processor Failures

In Section III-A-1, a concise description of SOFT reconfiguration is presented. The second section presents some basic properties of SOFT reconfiguration. Based on these properties, necessary and sufficient conditions for reconfigurability are derived in Section III-A-3.

Definition 1: Displacement is the logical movement of a node n to the physical position corresponding to f_n in order to replace f_n due to either the failure or displacement of f_n. At the leaves, displacement includes the logical movement of a leaf to the left or right in order to replace b_n or $cous_n$. Displacement through node n refers to the act of displacing n. □

Definition 2: Double displacement refers to an attempt to displace a node twice. For example, if n is displaced, it is assuming f_n's tasks. If displacement were to occur through n again, n would assume g_n's tasks. □

In analyzing the reliability of a reconfiguration scheme, it is necessary to determine both what fixed fault subsets (a set of processors in the tree designated as having failed) are reconfigurable (i.e., static reconfiguration), and for an existing machine with failures, whether the failure of an additio specific processor can be tolerated (i.e., dynamic reconfigu tion). For SOFT dynamic reconfiguration, the order in wh faults occur does not affect the ability of the architecture reconfigure. As a result, if it is known exactly what sets faults can be reconfigured, then a fault which occurs at tim can be reconfigured if and only if the new fault subset reconfigurable. Similarly, if it is known exactly what failu can be reconfigured given a set of faults, then all sets reconfigurable faults can be inductively determined. Con quently, the reconfigurability analysis of this section applicable to both scenarios.

1) Reconfiguration: As discussed in Section II, failure o node in an upper level results in a series of displacements ur a spare is configured in. If a leaf node n fails and it is adjac to its SST's spare, n is simply bypassed and replaced by t spare. If n is replacing f_n, n takes b_n as one son and the spa as its other son. If n is not adjacent to the SST's associat spare, it must be replaced by or take as its second son t nonbrother leaf adjacent to it, a_n. If a son of f_{a_n} is adjacent tc spare, f_{a_n} may use that spare as a second son, otherwise takes a leaf adjacent to one of its sons. This displaceme continues along level i until a spare is configured in. I convention, if a failure occurs in an SST, it displaces towa its associated spare if possible. If there are two failures in t SST, it must configure in the nonassociated spare. If t nonassociated spare had been configured in to its associat SST, then this SST must take its nonassociated spare. Th continues until an unused spare is configured in. In Fig. 4, . example of reconfiguration with four faults and four spares presented.

Complete algorithms have been developed which guarant proper reconfiguration of a SOFT tree in the presence of a set of multiple faults in which a reconfiguration exists [13]. concise summary of those algorithms is presented below.

Failure of spares: Bypass the spare. The result is equiv lent to taking the two SST's adjacent to the faulty spares a combining them into one larger SST.

Failure of SST nodes: If there are no faults within the SS and the associated spare is not configured in as the son of nonassociated SST, then reconfiguration proceeds toward t associated spare, else reconfiguration proceeds toward t nonassociated spare.

Failure of a node above the SST level: Reconfiguratic proceeds from an upper level to an SST. Once the displac ment has reached the SST level, it continues as previous described. Three conditions determine the SST toward whi reconfiguration proceeds: a) the root of the SST is descendant of the faulty node, b) all ancestors of the root of t SST on levels between and including the root of the SST a the level below the faulty node are fault free, c) the associat spare of the SST is not being used. If there exists an SS satisfying a), b), and c) then reconfiguration proceeds towa the leftmost such SST, else reconfiguration proceeds towa the leftmost SST satisfying a) and b). If there is no SST f which a) and b) is true then the tree is not reconfigurable.

2) Properties of SOFT Reconfiguration: The followir Lemmas present basic aspects of reconfigurability whic

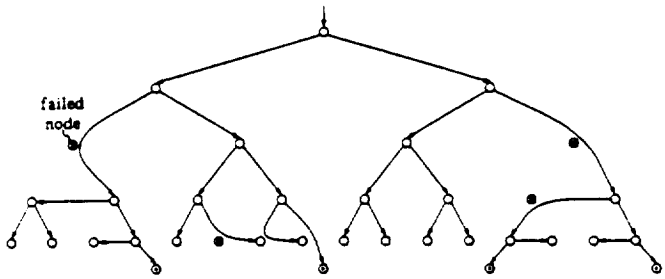


Fig. 4. Reconfiguration in an $i = 4$, $c = 2$ tree with four faults.

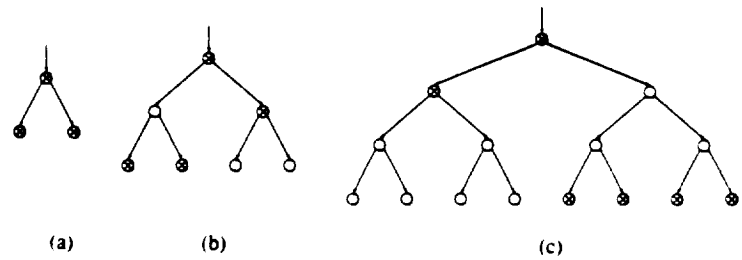


Fig. 5. Unreconfigurable faults at upper levels.

enable derivation of the necessary and sufficient conditions presented in Section III-A-C.

Lemma 1: Double displacement occurs at upper levels if and only if displacement is attempted through a faulty node.

Proof: Displacement only occurs in descendants of faulty nodes. The lemma is not concerned with displacement of leaves so no double displacement occurs due to unavailability of spares. For two displacements to intersect at one node (double displacement, by Definition 2), the displacement due to a failure higher in the tree must intersect a previously displaced node. But it is necessary to pass through a father in order to reach one of its descendants. □

Lemma 2: SOFT trees cannot reconfigure using double displacement.

Proof: By Definition 3, at upper levels some node n is assuming g_n's tasks. But n is not adjacent to b_{f_n}. Thus, communication with b_{f_n} would be lost, and it would not be possible to maintain the rigid tree topology if double displacement occurred. At level i, the father of the double displaced node would be adjacent to only one leaf processor. □

Lemma 3: At most one of n and b_n can be displaced.

Proof: Displacement of both would imply that either f_n has been displaced twice, which is not possible by Lemma 2, or there is a displacement through f_n and f_n has failed, which is also impossible by Lemmas 1 and 2. □

Lemma 4: Two failures within an SST, or displacement of the root of the SST, and a failure within the SST, are reconfigurable if and only if both spares adjacent to the SST can be reconfigured in as sons of nodes in the SST.

Proof: Displacement of the root of the SST is the same as failure of the root of the SST in terms of reconfiguration below that level. Consequently, only failures within the SST need be considered. If only one adjacent spare can be configured into the SST, and there are two failures in the SST, then reconfiguration is not possible by Lemma 2. If both adjacent spares can be used, then reconfiguration through fault-free nodes is clearly possible, i.e., a fault subset such as Fig. 5(a) cannot exist with only two failures. □

3) Analysis of Reconfigurability: In previous approaches, unreconfigurable multiple failures correspond to more than one fault in a group of nodes which have been allocated a single spare [6], [7]. Unreconfigurable multiple failures in the SOFT approach, regardless of the number and location of spares, correspond to faults which force double displacement. For example, consider the failure of a node n. Displacement must occur through a son_n. Consequently, the presence of a fault subset as depicted in Fig. 5(a) is not reconfigurable. If left son_n has failed, then displacement must occur through right son_n which implies that one of right son_n's must be fault free for reconfiguration to occur. The failure of left son_n can be thought of as forcing the reconfiguration into the subtree with right son_n as its root. Thus, Fig. 5(b) and (c) is not reconfigurable since there is no path from the highest faulty node to the leaves through only good nodes.

In order to determine what fault scenarios are reconfigurable, spare sufficiency (SS) is defined. SS is a Boolean value associated with each node of the tree.

Definition 3: The SS of a spare s is 1 if and only if s is fault free and not configured into the array, or, the nonassociated spare of the SST has SS of 1 and is not configured into the array as a son of s's associated SST, or s is faulty, and the SST for which s is not associated has SS of 1. The SS of a node within an SST is equal to the SS value of its associated spare. The SS of a node n in a level above the SST's is 1 if and only if SS(left son_n) or SS(right son_n) is 1 and n is fault free, where SS(n) denotes the spare sufficiency of n. □

In addition, the *shiftability* of a displaced node n, denoted as $s(n)$, is formally defined as

$$s(n) = \text{SS}(b_n) + \sum_{q=\text{level}(n)-1}^{q=1 \text{ by } -1} \text{SS}(b_{A_n^q}) \left[\prod_{l=\text{level}(n)-1}^{l=q \text{ by } -1} \text{good}(A_n^l) \right] \tag{1}$$

where good(n) is 1 only if node n is fault free.

Theorem 2 describes reconfigurability at the SST level. Corollaries 1-3 provide necessary and sufficient conditions for reconfiguration of a failure anywhere in a SOFT tree. Theorem 3 describes the class of fault subsets which are reconfigurable in SOFT architectures.

Theorem 2: The failure of any node n within an SST whose root is not displaced is reconfigurable if and only if SS(n) is 1.

Proof: A detailed proof can be found in [13]. An outline of the proof follows. From Definition 3, one of three conditions holds for a node in SST S to have SS of 1: a) its associated spare is available, or b) there exists an SST whose spare is available such that all SST's left of S and right of the available spare have either faulty spares or spares configured in to their associated SST's, or c) the associated spare of S is failed and its nonassociated SST has SS of 1. Any one of these three conditions is sufficient for reconfiguration. If none of these three conditions holds, reconfiguration is not possible. □

Corollary 1: Failure of node n, above the SST level, is

reconfigurable if and only if the node is not displaced and SS(n) is 1, or, the node is displaced and $s(n)$ is 1.

Proof: A detailed proof is contained in [13]. The proof distinguishes the cases of n not displaced and n displaced. If n is not displaced then Corollary 1 follows directly from Definition 3 and Theorem 2. If n is displaced, then the failure of n is reconfigurable if and only if n's displacement can be shifted into another subtree. In this case, Corollary 1 follows from the definition of $s(n)$ and the argument above. □

Corollary 2: Failure of a nonredundant node n within SST S can be tolerated if and only if either SS(n) is 1, or the root of S, r, is displaced and $s(r)$ is 1.

Proof: When a failure in an SST is detected, there are three possibilities for reconfiguration. If and only if SS(n) is 1, reconfiguration can proceed toward either the associated or the nonassociated spare, from Theorem 2. If r has been displaced, then the third alternative is to shift r's displacement into another SST, and use S's associated spare to reconfigure for n's failure. From Corollary 1, this is possible if and only if $s(r)$ is 1. □

Corollary 3: The failure of a spare n is tolerable if and only if the failure of a nonredundant node in either the associated or unassociated SST is tolerable.

Proof: If the spare is not configured into the array, then its failure is tolerable. A failure in either SST adjacent to the SST is also tolerable. If the spare has been configured into the array, then there are two possibilities for reconfiguration. The first is to undo the displacement causing the spare to be configured into the array. The second is to bypass the spare and use the neighboring SST's other adjacent spare. To do the first, it is necessary and sufficient that the spare had been configured into its associated SST (if the spare is configured into its unassociated SST, then the displacement cannot be shifted out of that SST) and a failure within the SST is tolerable (this follows from Lemma 4 and Corollary 2). If the spare is to be bypassed, then it is necessary and sufficient that a failure in the neighboring SST be tolerable, also by Lemma 4 and Corollary 2. □

Theorem 3: A SOFT tree is properly reconfigured if and only if each faulty node above the SST level has a path through fault-free nodes to the ith level (i.e., no subsets such as in Fig. 5), and there are no more than $x + 1$ faults or displacements of the root of an SST in any x adjacent SST's, and there are at least $x + 1$ spares in the tree.

Proof: If there is no reconfiguration forced into a subtree whose children are all faulty, then reconfiguration at upper levels can proceed through fault-free nodes into the roots of the SST's (from Corollary 1). The question of reconfigurability then concerns only the availability of spares at the SST level. If there are $x + 1$ faults in x SST's, there are exactly $x + 1$ spares available. At most two failures or a failure and a displacement of the root can occur within an SST, otherwise the condition of the Corollary is violated for $x = 1$. Lemma 4 indicates that any two failures within an SST can be tolerated as long as the nonassociated adjacent spare can be reconfigured in. Since there are $x + 1$ spares available, the faulty PE's are reconfigurable. □

Given a SOFT architecture and a specific fault subset, an algorithm has been implemented in Pascal for determining proper reconfiguration exists, and which specifies the confi ration. The algorithm runs in time directly proportional to number of nodes above the Spare Subtree level. Since the s of the tree above the SST's equals the number of spares l one, the algorithm is $O(n)$ for a fixed percentage of spar where n is the number of nodes in the tree. For a fixed numl of spares, the algorithm runs in order constant time.

B. Tolerance of Switch and Link Failures

In this section, a SOFT architecture is shown to be capal of tolerating functional failures in communication links a stuck-open and closed faults in the switches of Fig. 2. T following abbreviations are used in describing each node: PF denotes the processor of a node as being fault free, SF represents the switches associated with a node as being fau free, and LFF indicates the links connected to a node are fau free.

1) Failure of Links:

Definition 4: A node of a tree is *replaceable* if and only if is PFF, and, in the given fault scenario, the fault of tha processor is reconfigurable. □

The difference between replaceability and SS is tha replaceability assumes that there are no link/switch failures t prevent reconfiguration. Similarly, replaceability is substi tuted for spare sufficiency in the definition of shiftability, $s(n)$

Definition 5: A *nonredundant link* l (link between n anc f_n) is isolated if: a) f_l, *left* son_{f_l}, and *right* son_{f_l} are SFF anc LFF (except for the failed link), and b) b_{son_l} is replaceable. □

Lemma 5: If a faulty nonredundant link is isolated, then the tree can be reconfigured around that link.

Proof: The switching scheme of Fig. 2 allows the faulty link to be removed from the tree if b_{son_l} can be displaced. The restriction of Definition 5 b) guarantees that b_{son_l} can be displaced. Part a) of Definition 5 guarantees that the switching around the failed link will allow this reconfiguration to occur. □

If no ancestor has failed, then the link failure is considered as a failure of f_l. If an ancestor has failed, then its displacement is shifted through f_l thereby freeing a spare in another SST for use in case of another failure.

Definition 6: A redundant link l, in an upper level, is isolated if: a) f_{b_l} is PFF, SFF, and LFF, and both brothers are SFF, and b) $s(f_{b_l})$ is 1. □

Lemma 6: If a redundant link, at upper levels, is isolated then its failure is reconfigurable.

Proof: The restrictions of the definition and Corollary 2, guarantee that the displacement which necessitates the use of the redundant link can be shifted into another subtree. Thus, the faulty redundant link is no longer configured into the tree. □

Definition 7: A redundant link l in level i is isolated if either: a) the displacement is to the left (right) and the left (right) brother is replaceable, or b) the displacement is from above and f_{b_l} is PFF, SFF, and LFF and $s(f_{b_l})$ is 1. □

Consequently, if f_{b_l} has failed then the displacement is not shiftable.

Lemma 7: If any redundant link is isolated then its failure is reconfigurable.

Proof: At upper levels this is true by Lemma 6. At level i, Definition 7 applies. If the displacement is to the left or right [Definition 7 a)] and if that leaf is replaceable, the displacement which caused the use of this redundant link can be reversed to configure in a different spare. Thus, the redundant link once again becomes isolated from the tree. If the reconfiguration involving the redundant link is a result of a displacement of the father, then the displacement of the father must be shifted into another subtree, *and* reconfiguration in level i must be in the direction away from the faulty link, otherwise the faulty link will not be configured out of the tree [b)]. This is possible under the condition of part b) of Definition 7 by Corollary 2. □

Theorem 4: Isolated link failures can be tolerated in SOFT architectures.

Proof: From Lemmas 5 and 7, all interprocessor isolated link failures can be tolerated. A fault in a link connected directly to a processor [i.e., links A and B in Fig. 2(a)] can be modeled as a failure of the processor itself. In this case, the link fault is reconfigurable if the processor is replaceable. Failure of link C in Fig. 2(a) only prevents displacement to the left if the PE has failed (displacement to the right is possible), or displacement to the right if the PE has not failed (displacement to the left is still possible). In Fig. 2(b), failure of link C prevents displacement of the node to the left or right if the node is not faulty, and to the left only if the node has failed. □

2) Failure of Switches:

Definition 8: Any *switch in a nonspare node* is defined to be *isolated* if it is in a node which is PFF, SFF (except for the faulty switch), LFF, and the node is replaceable. Any *switch in a spare node* is defined to be *isolated* if the spare is SFF (except for the faulty switch), LFF, and the replaceability of the SST's adjacent to the spare are 1. □

Theorem 5: If a switch is isolated, then its failure is reconfigurable.

Proof: By symmetry, the only switches which need to be considered are switches 1–5 in Fig. 2(a) and (b) and the switches of 2(c). Since the node is PFF, SFF, and LFF prior to the switch failure, the node can support any reconfiguration for which either the faulty switch is not necessary (stuck-open fault), or the switch should be closed, or the closing of the switch has no effect. The fact that the node is replaceable indicates that the failure of this node is tolerable. As a result, if the node is displaced it can be returned to the undisplaced state. This can be done at upper levels by shifting the displacement into another subtree. At the leaves, the reconfiguration is shiftable since the failure of this leaf is reconfigurable. Failure of the switches of Fig. 2(a) and (b) is tolerable by the following analysis.

1) *Switch 1:* A stuck-open fault is reconfigurable as a failure of the processor. Reconfiguration of stuck-closed is not required since the node is PFF.

2) *Switches 2–5:* A stuck-open fault is reconfigurable by placing the node in the undisplaced state. Stuck-closed can also be properly configured in the undisplaced state.

In spare switching [Fig. 2(c)], for the case of stuck-closed faults, the spare is not configured in unless it is needed. Consequently, stuck-closed faults have no effect. For stuck-open faults, if the spare is not configured in, then the replaceability in both SST's is 1, and failure of any switch is tolerable. If the spare is configured in, then a stuck-open fault is tolerable since either SST can reconfigure a different spare in, which means that there is a configuration which does not employ these switches. □

IV. Reliability Analysis

In this section it is shown that the reliability of a SOFT binary tree, even with the restrictions imposed by VLSI layout, is always superior to a tree implemented using the class of approaches employed by [6] and [7]. This is demonstrated by first establishing an upper bound on the reliability of the previous approaches, independent of the actual implementation, i.e., their optimal reliability, and comparing it to a lower bound for SOFT trees. Exact reliability calculations of some specific SOFT implementations are also derived and compared.

A. Reliability of Other Approaches

It is assumed that the $i + 1$ level tree is allocated k spares. A Modular Sparing Approach (MSA) to fault tolerance in binary trees is any approach to reconfigurable design which partitions a tree into k groups of processors and allocates each group of processors one spare to be used exclusively by that group. The work of Raghavendra *et al.* (RAE) and Hassaan and Agarwal (M trees) can both be classified as MSA. Significantly, SOFT trees are not included in this category. It should be noted that the strategy of Rosenberg [8] is not MSA. However, Rosenberg's strategy does not allow for interconnect or switch failure. With an MSA, each module must be functioning in order for the tree to be functioning. Thus, the reliability can be expressed as the product of the reliability of all of the modules. In general this is

$$R_{\text{sys}} = \sum_{m=1}^{m=k} R_m \tag{2}$$

where R_m is the reliability of the mth module. Although some MSA schemes may tolerate interconnect failure, the following reliability analysis considers only processor failures, for sake of simplicity. Since the spare can be configured into the module in case of any single failure, the reliability of each module can be expressed as

$$R_{\text{module}} = R^q + qR^{q-1}(1-R)$$

where R is the reliability of each individual processor and q is the number of processors in the module, including the spare. The reliability for all processors is assumed to be equal and exponentially distributed (i.e., $R = e^{-\lambda t}$). Although this assumption is not accurate for many environments, it does provide an initial point of comparison and is a common assumption in reliability analysis [6], [7], [14]. For simplicity, the failure rate of spares, μ, is assumed to be equal to the failure rate of nonredundant processors λ. In the following

discussion, it is assumed that the trees can be divided evenly into modules of size q, although the theorem does not rely on this assumption.

Theorem 6: Optimal reliability in an MSA tree corresponds to when the tree is divided into modules of equal size: $q = (2^{i+1} - 1 + k)/k$.

Proof: Consider a tree with equal module size. The following inequality indicates that every time a single node is moved from a module of size q to another module of size q, thereby creating modules of size $q + 1$ and $q - 1$, R_{sys} decreases.

$$[R^q + qR^{q-1}(1-R)]^2 > [R^{q+1} + (q+1)R^q(1-R)] \cdot [R^{q-1} + (q-1)R^{q-2}(1-R)].$$

Additionally, the following inequality indicates that moving nodes from modules of smaller size into modules of larger size will decrease reliability.

$$[R^{q+c} + (q+c)R^{q+c-1}(1-R)] \cdot [R^{q-k} + (q-k)R^{q-k-1}(1-R)] > [R^{q+c+1} + (q+c+1) \cdot R^{q+c}(1-R)][R^{q-k-1} + (q-k-1)R^{q-k-2}(1-R)]$$

for any $0 \leqslant c$, and $0 \leqslant k < q$. □

It has been shown that the reliability versus spares curve of M-trees is greater than that of RAE [7]. The reason for this is that the RAE scheme allocates an entire spare to the root, whereas M-trees can "spread" the spare out into level 1 (or more) nodes. At lower levels, however, the number of nonredundant nodes per spare is the same. Consequently, as i and/or k increases, the reliabilities of both schemes converge. In contrast, the next section demonstrates that the SOFT approach *always* results in superior reliability over MSA designs.

B. Reliability of SOFT Trees

The reliability of a redundant system composed of nodes with equal reliability R, which is not subject to degraded performance, can be thought of as a polynomial of degree $(2^{i+1} - 1 + k)$ where there are $(2^{i+1} - 1 + k)$ nodes in the system. The polynomial can be expressed as

$$R_{sys} = \alpha_0 R^{(2^{i+1}-1+k)} + \alpha_1 R^{(2^{i+1}-1+k)-1}(1-R) + \cdots + \alpha_j R^{(2^{i+1}-1+k)-j}(1-R)^j + \cdots + \alpha_{(2^{i+1}-1+k)}(1-R)^{(2^{i+1}-1+k)}$$

where R is the reliability of individual components, and α_j is the number of ways in which j reconfigurable faults can occur in the tree ($\alpha_j = 0$ for $j > k$). For comparison of α, an α_j from a SOFT reliability equation is denoted as α_{j_s}, whereas, α_{j_m} is associated with the optimal MSA reliability.

In order to analyze SOFT reliability, it is necessary to calculate the number of possible processor locations for the jth faulty processor to occur given that $j - 1$ faults have already been successfully reconfigured. This is dependent on the specific SOFT implementation; however, using the following identity, a lower bound on the number of possible locations for the jth fault is derived in Theorem 7.

$$\text{Number of nodes per SST (including the spare)} = \frac{2^{i+1}}{k}.$$

If a given SOFT or MSA tree contains $j - 1$ specific fault and the tree has not failed, then the location of the faul processors (fault subset) is referred to as an α_{j-1_s} or α_{j-1} scenario, respectively. In SOFT, if $j - 1$ faults have bee reconfigured, this implies that the associated spares of $j -$ SST's are configured in (or failed), and $k - j + 1$ SST's hav associated spares which are not used.

Theorem 7: $(k - j + 1)((2^{i+1}/k) + 1)$ is a lower boun on the number of reconfigurable faults from an α_{j-1_s} scenario

Proof: In each of the $k - j + 1$ SST's which has its associated spare unused, at least $(2^{i+1}/k)$ positions for the jth fault exist, which can be reconfigured. If in α_{j-1_s} the father of one of the $k - j + 1$ SST's has not yet failed, then the failure of the father can be tolerated and 1 can be added to $2^{i+1}/k$ as the number of locations for $k - j + 1$ of the SST's. If the father has failed, however, then it must have been reconfigured into a neighboring SST, which means that a fault in that neighboring SST can be tolerated and the f_{SST}'s reconfiguration shifted into the SST with the free spare. The lower bound, therefore, remains valid. If two of the $k - j + 1$ SST's are adjacent to the same father, then the father has been counted twice in the lower bound. If the father f has not yet failed, however, then the failure of f_f is also tolerable, and by the same analysis, the lower bound remains valid. □

Theorem 8: The reliability of a SOFT tree is always greater than the reliability of an equivalent MSA tree for k (the number of spares) > 1.

Proof: If it can be shown that for at least one j, $\alpha_{j_s} > \alpha_{j_m}$, and $\alpha_{j_s} \geqslant \alpha_{j_m}$ for all other j, then $R_{sys_{SOFT}} > R_{sys_{MSA}}$ (assuming that the reliability of individual processors is the same for both architectures). Since $\alpha_{0_m} = \alpha_{0_s} = 1$ and $\alpha_{1_m} = \alpha_{1_s} = (2^{i+1} - 1 + k)$, R_{sys} with $k = 0$ or $k = 1$ is identical for both schemes. Consider, however, α_2. MSA can tolerate only one failure per module. As a result

$$\alpha_{2_m} = \binom{(2^{i+1} - 1 + k)}{2} - k\binom{(2^{i+1} - 1 + k)/k}{2}$$

where

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

and

$$\alpha_{2_s} \geqslant \binom{(2^{i+1} - 1 + k)}{2} - k\binom{2^{i+1}/k}{2}.$$

Now consider α_j. Since $\alpha_{2_s} > \alpha_{2_m}$, it will be shown by contradiction that $\alpha_{j_s} > \alpha_{j_m}$ given that $\alpha_{j-1_s} > \alpha_{j-1_m}$. From the definition of α, it is evident that if $\alpha_{j_m} \geqslant \alpha_{j_s}$, at least one fault configuration corresponding to a successful reconfiguration of $j - 1$ faults in MSA (denoted as a fault configuration in α_{j-1_m}) must have more possible locations for successful reconfiguration of the jth fault *than* the possible locations for the jth fault in an α_{j-1_s} configuration.

Derivation of the number of "choices" for the jth fault to

occur for each α_{j-1} in optimal MSA trees is straightforward and is less than the lower bound of Theorem 7.

$$\text{places for } j \text{ per } \alpha_{j-1_m} = (k-j+1)\left(\frac{(2^{i+1}-1+k)}{k}\right). \quad \square$$

. Due to the analytical complexity of a global reconfiguration strategy and the variety of possible SOFT implementations, a closed form expression for reliability has not been found for arbitrary size trees with arbitrary numbers of spares. However, the analysis presented in Section III is sufficient to determine the reliability of any specific SOFT tree. The following section presents some exact reliability calculations for example SOFT implementations.

C. Reliability Examples

The reliabilities of four-level trees, implemented by M-tree, RAE, and SOFT as a function of the number of spares, for $R = e^{-0.1t}$, $t = 0.5$, and $t = 1.0$, are shown in Table II. It should be noted that the SOFT reliability is superior even when the modular schemes are allocated more spares. The data for $t = 1.0$ are graphically displayed in Fig. 6. In Table III and Fig. 7, the reliabilities versus time curves of a tree with no redundancy, duplicated four-level trees, and four-level trees with four spares employing an optimal MSA, a SOFT approach, and a tree with optimal reliability are presented. A scheme with optimal reliability guarantees reconfiguration for any number of faults less than or equal to the number of spares. Table III includes calculations for an RAE tree with four spares and an M-tree with five spares. The M-tree approach was allocated five spares due to its inability to support four.

D. Increasing SOFT Reliability

There are several possibilities for enhancing the reliability of a SOFT architecture. If a designer is not concerned with VLSI layout issues or is willing to pay $O(n \log n)$ area, the SOFT tree can be implemented such that the leaves are fully connected and the full sharing of inter-SST spares is practical. As an alternative, sharing of spares between the $i = 4$ leaf subtrees is possible using the procedure of Horowitz and Zorat [9]. A second option is the addition of redundant lines between leaves which are adjacent to the same spare. This allows reconfiguration of SST's with numerous faults, assuming that the spares are available in neighboring SST's. The cost of adding these lines is an additional k lines (where k is the number of spares). Finally, for applications such as yield enhancement, where processor yield may be quite low, ≈ 50 percent sparing is possible. In SOFT implementations with 50 percent sparing, spare and link placement is the same as in Algorithms 1 and 2, with the exception that each spare is associated with a single leaf and there are no cousin connections between leaves.

E. SOFT Performance Degradation

The SOFT approach, as with the RAE technique, allows for graceful degradation in performance. No redundant spares are allocated for graceful degradation; however, redundant lines

TABLE II
RELIABILITY FOR A FOUR-LEVEL TREE BY NUMBER OF SPARES

t	number of spares	R_{RAE}	$R_{M\text{-}tree}$	$R_{opt\text{-}MSA}$	R_{SOFT}
0.50	8	0.9504	——	0.9512	——
	5	0.9281	0.9350	0.9350	——
	4	0.9033	——	0.9248	0.9974
	2	——	0.8499	0.8811	0.9528
	1	——	0.8179	0.8179	0.8179
1.00	8	0.8274	——	0.8298	——
	5	0.7655	0.7832	0.7832	——
	4	0.7073	——	0.7563	0.9665
	2	——	0.6194	0.6553	0.7841
	1	——	0.5416	0.5416	0.5416

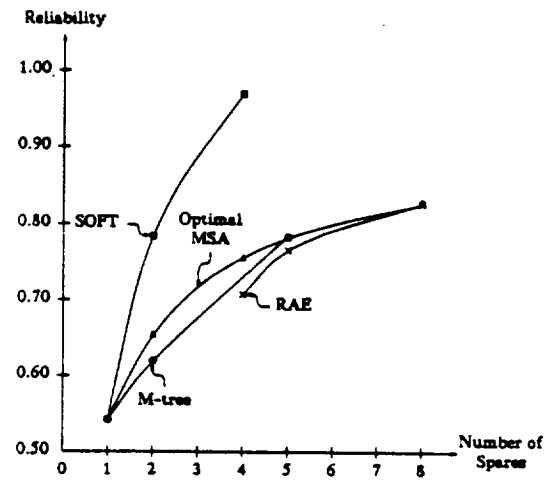


Fig. 6. Reliabilities of four-level trees at $t = 1.0$.

TABLE III
RELIABILITY FOR $\lambda = 0.1$

t	$R_{no\ red.}$ 0 spares	R_{Dup}	R_{RAE} 4 spares	$R_{M\text{-}tree}$ 5 spares	$R_{MSA\text{-}opt}$ 4 spares	R_{SOFT} 4 spares	R_{OPT} 4 spares
0.00	1.000	1.000	1.000	1.000	1.000	1.000	1.000
0.25	0.687	0.902	0.972	0.982	0.979	1.000	1.000
0.50	0.472	0.721	0.903	0.935	0.925	0.997	0.998
0.75	0.325	0.544	0.810	0.866	0.847	0.987	0.990
1.00	0.223	0.396	0.707	0.783	0.756	0.965	0.971
1.25	0.153	0.283	0.604	0.694	0.661	0.927	0.936
1.50	0.105	0.200	0.505	0.604	0.566	0.873	0.886
1.75	0.072	0.140	0.417	0.517	0.447	0.807	0.822
2.00	0.050	0.097	0.339	0.437	0.396	0.731	0.747

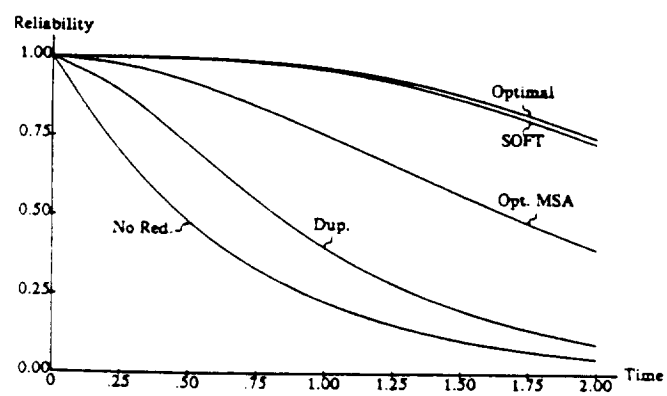


Fig. 7. Reliability versus time for a four-level tree and four spares.

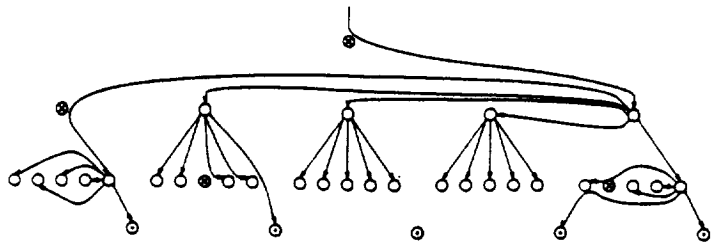


Fig. 8. Reconfiguration in a 5-ary tree.

are located as described by Algorithm 2. For SOFT architectures a spare at the root is unecessary, in contrast to previous graceful degradation approaches [6]. Since failures are passed to the leaves, the only nodes which must assume the functions of their brothers are the leaves. Also, as noted before, the redundancy in terms of links is reduced and the reliability is enhanced. The only failures which disable the tree are long runs of failures along the leaves and a failure of a node for which no path of good nodes into the leaves exists, i.e., the class of failures depicted in Fig. 5.

V. SOFT N-Ary Trees

N-ary trees, in which each nonleaf node has N sons, are more suitable for certain tasks than classical binary trees. For example, 4-ary (quad) tree architectures have been proposed for implementing several classes of artificial intelligence related algorithms [15]. The following brief discussion summarizes how the SOFT approach is applicable to N-ary trees. A more complete description and analysis can be found in [13].

A. Construction of Reconfigurable N-ary Trees

1) Location of Redundant Lines: A link allocation approach which is applicable to arbitrarily large N is to restrict reconfiguration to displacement of the "outside" two children of each nonleaf node. Redundant links to each of a node's brothers are added only to the two outside brothers. The link redundancy at upper levels is approximately $(2N - 3)/N$ percent.

2) Location of Redundant Processors: Allocation of up to one associated spare per group of N brothers is allowed. The spares are placed between SST's with connections from a spare to both its associated SST and a neighboring SST. Redundancy is, therefore,

$$\text{percent link redundancy} \approx \frac{2N-3}{N} + \left(\frac{N^{i-1} + N^c}{\sum_{g=0}^{g=i} N^g} \right)$$

$$\text{percent node redundancy} = \frac{N^c}{\sum_{g=0}^{g=i} N^g}$$

where N^c is the number of spares allocated to the tree, with $1 \leqslant c \leqslant i - 1$.

B. Reconfiguration

Reconfiguration is fundamentally the same as in bi: trees. A sample reconfiguration for four failures in a 5-ary is illustrated in Fig. 8.

VI. Conclusions

A unique approach to the design of reconfigurable architectures has been presented. The design allocates sp: at the leaves of trees and allows sharing of spares betw subtrees. The architecture has $O(n)$ VLSI layout for bin trees and is directly extensible to N-ary trees. A lower bo on reliability for a SOFT tree was shown to be more relia than all modular sparing approaches, with significantly l redundancy. The SOFT architecture provides for reconfigu ble trees which tolerate both link and switch failures. 7 virtual displacement concept with sharing of spares betw clusters of processors is also applicable to other concurr architectures.

Acknowledgment

The authors wish to express their thanks to M.-F. Chang f his assistance in preparing this paper.

References

[1] C. A. Mead and L. A. Conway, *Introduction to VLSI Systen* Reading, MA: Addison-Wesley, 1980.

[2] S. A. Browning, "A tree machine," *Lambda (VLSI Design)*, pp. 3 36, Second Quarter 1980.

[3] M. J. Attallah and S. R. Kosaraju, "A generalized dictionary machi for VLSI," *IEEE Trans. Comput.*, pp. 151-155, Feb. 1985.

[4] J. P. Hayes, "A graph model for fault tolerant computing systems. *IEEE Trans. Comput.*, pp. 875-883, Sept. 1976.

[5] C. L. Kwan and S. Toida, "An optimal 2-fault tolerant realization symmetric hierarchical tree systems," *Networks*, vol. 12, pp. 231 239, 1982.

[6] C. S. Raghavendra, A. Avižienis, and M. Ercegovac, "Fault toleran in binary tree architectures," *IEEE Trans. Comput.*, pp. 568-52 June 1984.

[7] A. S. M. Hassan and V. K. Agarwal, "A fault-tolerant modul: architecture for binary trees architectures," in *Proc. Fifteenth Fau Tolerant Comput. Symp.*, June 1985, pp. 344-349.

[8] A. L. Rosenberg, "The Diogenes approach to testable fault tolerar arrays of processors," *IEEE Trans. Comput.*, pp. 902-910, Oc 1983.

[9] E. Horowitz and A. Zorat, "The binary tree as an interconnectio network: Applications to multiprocessor systems and VLSI," *IEE Trans. Comput.*, pp. 247-253, Apr. 1981.

[10] J. R. Goodman and C. H. Sequin, "Hypertree: A multiprocesso interconnection topology," *IEEE Trans. Comput.*, pp. 923-933 Dec. 1981.

[11] A. M. Despain and D. A. Patterson, "X-tree: A tree structure multiprocessor computer architecture," in *Proc. Fifth Compute Architecture Symp.*, April 1978, pp. 144-150.

[12] J. B. Ullman, *Computational Aspects of VLSI*. Rockville, MD Computer Science Press, 1984.

[13] M. B. Lowrie, "Reconfigurable tree architectures," M.S. Thesis CSG Rep. 65, Coordinated Sci. Lab., Univ. Illinois, Urbana, 1987.
[14] R. S. Swarz and D. P. Siewiorek, *The Theory and Practice of Reliable System Design*. Bedford, MA: Digital, 1982.
[15] A. Rosenfeld, "Quadtrees and pyramids for pattern recognition and image processing," in *Proc. Fifth Int. Conf. Pattern Recognition*, 1981, pp. 802-806.

Matthew B. Lowrie received the B.S. degree (highest honors) in computer engineering and the M.S. degree in electrical engineering from the University of Illinois, Urbana, in 1985 and 1986, respectively.

He is currently a Ph.D. degree candidate in electrical engineering at the University of Illinois. His research interests include all aspects of high speed parallel processing with emphasis on fault-tolerant computing. He is supported by a fellowship with AT&T Bell Laboratories.

W. Kent Fuchs (S'80–M'83) received the B.S.E. degree in electrical engineering from Duke University, Durham, NC, in 1977, the M.S. degree in electrical engineering from the University of Illinois, Urbana, in 1982, the M.Div. degree from Trinity Evangelical Divinity School, Deerfield, IL, in 1984, and the Ph.D. degree in electrical engineering from the University of Illinois, in 1985.

He is currently an Assistant Professor in the Departments of Electrical and Computer Engineering, Computer Science, and the Coordinated Science Laboratory, University of Illinois. His research interests include all aspects of VLSI system design with emphasis on fault-tolerant computing. Current funding agencies for this research include the Semiconductor Research Corporation, the Office of Naval Research, Digital Equipment Corporation, the National Aeronautics and Space Administration, the Joint Services Electronics Program, the Microelectronics and Computer Technology Corporation, Texas Instruments, and the National Science Foundation.

Dr. Fuchs recently received the Digital Equipment Corporation Incentives for Excellence Award, 1986, the Best Paper Award, ACM/IEEE Design Automation Conference, in the simulation and test category, 1986, and the Xerox Faculty Research Award, College of Engineering, University of Illinois, 1987.

SUBMITTED: IEEE Transactions on Computers

LOCAL CONCURRENT ERROR DETECTION AND CORRECTION IN DATA STRUCTURES USING VIRTUAL BACKPOINTERS

C. C. Li, P. P. Chen, W. K. Fuchs

Computer Systems Group
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1101 W. Springfield Ave.
Urbana, IL 61801

ABSTRACT

A new technique, based on virtual backpointers, for local concurrent error detection and correction in linked data structures is presented in this paper. Two new data structures, the Virtual Double-Linked List, and the B-Tree with Virtual Backpointers, are described. For these structures, double errors can be detected in O(1) time and errors detected during forward moves can be corrected in O(1) time. The application of a concurrent auditor process to data structure error detection and correction is analyzed, and an implementation is described, to determine the effect on the mean time to failure of a multi-user shared-database system. The implementation utilizes a Sequent shared-memory multiprocessor system operating on a shared database of Virtual Double-Linked Lists.

Index Terms- concurrent error detection, data structures, concurrent structure checking

---

This research was supported in part by the National Aeronautics and Space Administration (NASA) under Contract NASA NAG 1-602, and in part by the SDIO/IST and managed by the Office of Naval Research under contract N00014-86-K-0519.

Address of Correspondent

W. Kent Fuchs
Computer Systems Group
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1101 W. Springfield Ave.
Urbana, IL 61801

(217) 333-9731

I. INTRODUCTION

Linked data structures form an integral part of many software and database systems. Performing error detection and correction to preserve the correctness of data structures is important in achieving overall system reliability. To reduce the performance degradation incurred through their use, detection and correction should ideally be executed concurrently with normal processing, and every invocation of these procedures should be completed in O(1) time. If any global checking information (e.g., a global count) is used in detection or correction, then O(n) nodes must be accessed, where n is the number of nodes in the structure, and those procedures cannot run in O(1) time. In addition, since node access time is the major contributing factor to the cost of error detection, the number of nodes accessed should be minimized. The Checking Window concept is introduced in this paper as a method of formalizing these ideas, and as a method of describing local concurrent error detectability as a function of the number of nodes to be checked. To preserve the structural integrity of linked data structures, a new approach to detecting and correcting structural errors, called the virtual backpointer, is also introduced in this paper. The technique is used to construct two new data structures: the Virtual Double-Linked List and the B-Tree with Virtual Backpointers. The Virtual Double-Linked List uses the same amount of storage as the double-linked list from which it is derived. The B-Tree with Virtual Backpointers, derived from the B-tree of order m, requires m+4 more fields in each node. It is shown that O(1) local concurrent error detection can be performed for both structures, and that O(1) correction is possible for those errors detected during forward moves through the structures. Correction for those errors detected during backward moves through the structures is in worst case O(n).

The foundation work concerning robust data structures was performed by Taylor, Morgan, and Black [1]. Several techniques have since been presented to achieve robust data structures; however, most achieve error detection in O(n) time. A global count, as used by Taylor, Morgan and Black in the modified(k) double-linked list, the chained and threaded binary tree, and the robust B-tree [1-3], by Munro and Poblete in their isomorphic binary tree [4], by Sampaio and Sauvé in

their robust binary tree [5], and by Seth and Muralidhar in their mod(2) chained and threaded binary tree [6], necessitates, for some errors, a traversal of all the nodes of the structure for error detection. The three pointer tree, as explained by Yoshihara *et al.* [7] requires $O(n)$ time to detect double errors, since a preorder traversal of all the nodes of the tree is performed. Though not indicated in their paper, error detection can be performed in $O(1)$ time using the D-loops within the structure, but only single errors can be detected. Kuspert's work with the separately-chained hash table [8], which is an application of double-linked lists, achieves detection in $O(1)$ time; however, five extra fields must be stored in each node.

A general theory of local detectability and local correctability has been introduced and formalized by Black and Taylor [9], and has been successfully applied to several different types of data structures, including: the spiral(k) list [9], the LB-tree [9-10], the mod(k) list [11], the helix(k) list [12], and the AVL tree [13]. The intention of their work is to be able to correct an arbitrary number of errors in a data structure, provided the errors are sufficiently separated from each other. However, the complexities of the correction algorithms (which include error detection) are typically not $O(1)$.

The organization of this paper is as follows. Section II presents an analysis of local concurrent error detection, giving formal definitions for Checking Windows and local concurrent error detectability. In Section III, the virtual backpointer concept is described and is used to construct two new data structures: the Virtual Double-Linked List and the B-Tree with Virtual Backpointers. The local concurrent error detectability and correctability of each structure is analyzed. Section IV describes a concurrent auditor process as applied to data structure error detection, analyzes its effectiveness in increasing the mean time to failure of a system, and presents the results of an implementation. Finally, Section V summarizes the results.

II. LOCAL CONCURRENT ERROR DETECTION AND CORRECTION

Local concurrent error detection (LCED) is an on-line technique for detecting structural errors in a locality of a currently accessed node in a linked data structure. If the size of the locality is constant and the degree of each node is fixed, then an LCED procedure will run in O(1) time. *Local concurrent error correction* (LCEC) can correct errors detected by an LCED procedure, using another locality of the currently accessed node (not necessarily the same as that used by the LCED procedure). If the size of the locality is again constant, then an LCEC procedure will run in O(1) time. Error detection and correction typically degrade system performance. The degradation is a function of the number of nodes accessed, the number of nodes stored, and the computation required, for detection and correction. For the LCED procedures analyzed here, no extra node accesses are required (except in the initialization phase). Hence, the storage and computation requirements dominate the cost of error detection and correction.

Linked data structures may be modeled as directed graphs. A graph $G = (N, E)$ consists of a finite set of nodes $N = \{N_1, N_2, \cdots, N_n\}$ and a finite set of edges $E = \{E_1, E_2, \cdots, E_m\}$. Each edge $E_i = <N_j, N_k>$ links a pair of ordered nodes in this directed graph (digraph). In the digraph representation of a linked data structure, the nodes represent the data records, and the edges represent the pointers between the records. If all the nodes consist of the same fields, then the data structure is said to be *uniform*. A *move* from a node N_j to a node N_k is possible if there exists an edge E_i between them, and is represented as $N_j \rightarrow N_k$. Then N_k is *reached* from N_j by *following* E_i. A *traversal* is a series of moves starting at a root node or header of a structure that accesses part or all of the data structure.

An LCED procedure is invoked to detect structural errors whenever a move attempts to follow a pointer, which may be a forward pointer, a backward pointer, or a virtual backpointer (Section III). That is, the LCED procedure attempts to *verify* the move. Thus, it is on-line, or concurrent with normal structure access.

The errors considered in this paper are those that affect the structural information of the data structure (e.g., pointer values, structural checking information). The probability of an erroneous pointer to a random location remaining undetected by the techniques presented in this paper is proportional to 2^{-bd}, where b is the number of bits used to represent a pointer, and d is the number of erroneous pointers required for masking. Since this probability is very low, the error detection analysis concentrates on the case where erroneous pointers point to other nodes of the same type. This kind of error may occur in partially or incorrectly updated data structures, or as a result of software errors or hardware failures. These erroneous pointers may or may not coincide with logical pointer boundaries; however, the routine that accesses nodes from slow memory can detect these boundary errors and supply this information to the LCED procedure.

Memory subsystems are commonly configured hierarchically, and the ratio of the access time of slower memory (used to store the data structure, e.g., MOS RAM, disk) to that of faster memory (used to buffer the currently accessed nodes, e.g., cache, register file) is usually very large. Hence it is desirable to have all the nodes in the LCED or LCEC localities stored in the fastest memory. In the remainder of this paper, A_i will represent the address of a node N_i in a linked data structure. N_i may have many pointers to other nodes, and a desired move MV from N_i will be represented as $N_i \rightarrow N_{MV}$.

DEFINITION 1: R_c is a fast memory of capacity c nodes, which holds the c most recently accessed nodes, including the node reached by the current move MV. Since a move is performed between two nodes, c must be at least two to verify the move. That is, for a move MV $N_i \rightarrow N_{MV}$, R_c holds both N_i and N_{MV}. If $c = 1$ then only N_{MV} could be stored, and the information of the source node N_i (e.g., address, pointer value) would be lost. Thus, an erroneous move would be indistinguishable from a correct move. □

The LCED procedure requires a set of c nodes to verify the move MV. This set of nodes is called a *Checking Window*. The cost of a Checking Window is proportional to c, since it involves storing the required nodes in the fast memory (storage cost) and performing checks on those nodes

(computation cost). The nodes in the Checking Window need not be re-accessed from slow memory, since they are already stored in R_c.

DEFINITION 2: Let a set of Checking Windows of size c, W^c, be defined recursively as: $W^c = \{W_j^{c-1} \cup N_k\}$ where W_j^{c-1} is the jth *Checking Window* of W^{c-1} $(1 \leqslant j \leqslant |W^{c-1}|)$ and $N_k \notin W_j^{c-1}$ is adjacent to one of the nodes in W_j^{c-1}. The base case is $W^2 = \{\{N_i, N_{MV}\}\}$. □

W_m^c, for some m, is constructed by adding one more node N_k to the smaller Checking Window W_j^{c-1}, such that N_k can be reached from W_j^{c-1} in one move. All such W_m^c form a set of sets, W^c. It will be shown that Checking Windows of the same size do not necessarily achieve the same detectability. When the context is clear, we may use W^c to represent one particular W_j^c.

EXAMPLE 1: Consider a forward move $N_i \rightarrow N_{i+1}$ in a normal double-linked list (Figure 1):

$$W_1^2 = \{N_i, N_{i+1}\}$$

$$W^2 = \{W_1^2\} = \{\{N_i, N_{i+1}\}\}$$

$$W_1^3 = \{N_i, N_{i+1}, N_{i+2}\}$$

$$W_2^3 = \{N_{i-1}, N_i, N_{i+1}\}$$

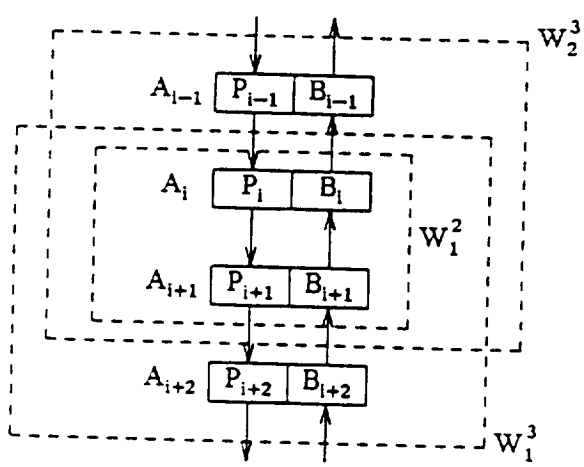


Figure 1. Checking Windows for a Double-Linked List.

$$W^3 = \{W_1^3, W_2^3\} = \{\{N_i, N_{i+1}, N_{i+2}\}, \{N_{i-1}, N_i, N_{i+1}\}\}$$

etc. □

The Lock and Key concept is now introduced as a generalization of structural checking information that is distributed throughout the nodes of linked data structures (distributed checks). In the simplest case, nodes in the structure will have associated with them a Key. When performing a move from a node to its child, the node's Key becomes an argument to the child's Lock function, which either returns "True," signaling a valid move, or "False," signaling error detection. In its most general form, the Lock and Key concept allows for multiple-Key Locks and Keys distributed over potentially many nodes.

DEFINITION 3: A *Key* is information associated with a node (e.g., its address, a pointer, or distributed check) that is used by a checking function to verify a move. □

DEFINITION 4: A *Lock*, $Lock_{MV}$, is a checking function that verifies a move, such that $Lock_{MV}(Key_1, \cdots, Key_k)$ = "True" if all its Key_i arguments are present and correct, "False" if all its Key_i arguments are present and not all are correct, or "X" (don't care) if not all its Key_i are present. A Lock whose Key arguments are all present is called a *checkable* Lock, otherwise the Lock is an *uncheckable* Lock. □

The computational overhead to evaluate the checkable Locks is O(1) if all $Lock_{MV}$ are defined on Keys that can be contained in a fixed-size Checking Window W_j^c. No storage overhead is necessary because Locks are functions and are not stored, and Keys can be information that is already present in the node, e.g., pointers.

DEFINITION 5: A *Circular Lock*, $CLock_{N_i \sim N_k}$, is a Lock function whose Keys are addresses of nodes:

$$Keys = \langle A_i, A_k \rangle$$

$$CLock_{N_i \sim N_k}(x, y) = (x \ ?= g(y))$$

where $\sim$ is a pointer (e.g., a forward pointer, a backward pointer, a virtual backpointer) of N_i to

N_k. g is a function that generates x using a series of pointers, and ?= represents a comparison that returns either "True" or "False" for a checkable CLock. □

Circular Locks possess the property that for all starting nodes N_i, any single pointer error encountered in the moves of g causes the Lock to evaluate to "False." The following two examples show that the double-linked list and a binary tree with signatured access paths employ Locks and Keys. The double-linked list uses a Circular Lock checking function, while the tree with signatured access paths uses a Lock defined on O(*height-of-tree*) Keys.

EXAMPLE 2: Let $N_0, N_1, \cdots, N_n$ be the nodes of a double-linked list. Let a node N_i have a forward pointer P_i and a backpointer B_i. For a forward move $N_i \rightarrow N_{i+1}$:

$$\text{Keys} = \langle A_i, A_{i+1} \rangle$$

$$\text{CLock}_{N_i \rightarrow N_{i+1}}(x, y) = (x \ ?= g(y)) = (x \ ?= y.B).$$

The backpointers are the distributed checks, and the g function in the Circular Lock retrieves the backpointer B from the node at y. This structure achieves O(1) single pointer error detection in Checking Window W_1^2 (*cf.* Example 1). □

EXAMPLE 3: In the signatured access path technique, signatures defined over the nodes of valid traversal paths are embedded at path termination points, where a traversal path starts at a header and ends at a leaf, for a binary tree [14]. Error detection is achieved by comparing signatures generated at traversal time with the embedded signatures. A simple signature is the logical exclusive-or function ($\oplus$) of all the pointers in the valid traversal path.

$$\text{Keys} = \langle \text{ordered set of pointers in a valid traversal path, signature} \rangle$$

$$\text{Lock}_{\text{forward}}(p_1, \cdots, p_k, \text{signature}) = (p_1 \oplus \cdots \oplus p_k \oplus \text{signature} \ ?= 0).$$

The nodes' pointers are the distributed checks. This structure cannot guarantee O(1) detection time as O(*height-of-tree*) nodes may be accessed in the traversal path. □

We now determine the minimum number of errors that are required to cause the checkable Locks used by the LCED procedure to evaluate to "True" in a particular Checking Window. This is similar to the *changes* used by Taylor, Morgan and Black [15] to determine the distance between two data structure instances. The difference here is that the distance is measured within a Checking Window. Hence this new distance is termed *local distance*, from which the definition of local concurrent error detectability follows directly. Let $Lock_{MV}$ be defined, for every possible move MV in a specific data structure, over Keys distributed in nodes contained in a fixed-size Checking Window.

DEFINITION 6: The *local distance*, $d_j^c(MV)$, within a Checking Window of size c is defined as the minimum number of pointer errors in all W_j^c that can mask a move to an incorrect node, due to a pointer error, where MV is the move to the correct node. Errors are not detectable if all checkable $Lock_{MV}$ evaluate to "True." □

DEFINITION 7: The *local concurrent error detectability*, $D^c(MV)$, for a specified move MV and Checking Window of size c is given by:

$$D^c(MV) = \max(d_j^c(MV)) - 1,\ 1 \leqslant j \leqslant |W^c|. \qquad \square$$

The max function is used because, for a specified move, it is always possible to find a Checking Window W_j^c which can detect at least D^c simultaneous errors (including the pointer from N_i to N_{MV} that is erroneous). When the context is clear, we may omit the parameter MV in $d_j^c(MV)$ or $D^c(MV)$.

The following theorem will be used to prove that the local concurrent error detectability of data structures employing the virtual backpointer is the same for both forward and backward moves.

THEOREM 1: In a uniform data structure, if for every pointer of the form $N_i \rightarrow N_k$ there exists a $\sim$ pointer to reach N_i from N_k in one move, and the Lock functions are Circular Locks, then using an LCED procedure, $D^c(N_i \rightarrow N_k) = D^c(N_k \sim N_i) = D^c$.

PROOF: Since the data structure is uniform, $N_i \rightarrow N_k$ and $N_k \sim N_i$ represent all possible forward and backward moves, respectively. Notice that $W_1^2 = \{N_i, N_k\}$. Thus, all W_j^c are also the same for both moves as W_j^c is defined on W_1^2. If $N_i \rightarrow N_k$ is erroneously changed to $N_i \rightarrow N_{k'}$, it is isomorphic to the case $N_k \sim N_i$ being changed to $N_k \sim N_{i'}$, because the pointers used in the g function of the Circular Lock are not changed by the isomorphism. In both cases, the Locks evaluate to the same value because the accessible nodes in W_j^c are the same. By Definition 6, $d_j^c(N_i \rightarrow N_k) = d_j^c(N_k \sim N_i)$. Hence $D^c(N_i \rightarrow N_k) = D^c(N_k \sim N_i) = D^c$.

□

Theorem 2 will be used in determining the upper bounds of local concurrent error detectability for the Virtual Double-Linked List and B-Tree with Virtual Backpointers.

THEOREM 2: Local concurrent error detectability is a monotonically increasing function of window size c. That is, $D^{c-1} \leqslant D^c \leqslant D^n$ for $3 \leqslant c \leqslant n$, where n is the total number of nodes in the data structure.

PROOF: Every W_m^c is constructed by adding one adjacent node N_k to a Checking Window of size c-1: $W_m^c = W_j^{c-1} \cup N_k$. If each checkable Lock in W_j^{c-1} evaluates to "True" in W_j^{c-1} then it will remain "True" in W_m^c because the Keys of the Lock are contained in both W_j^{c-1} and W_m^c. If the addition of N_k causes an uncheckable Lock in W_j^{c-1} to evaluate to "True" or "X" in W_m^c, this results in $d_m^c = d_j^{c-1}$. However, if the uncheckable Lock evaluates to "False," then $d_m^c > d_j^{c-1}$, since at least one other error would be required to mask the detected error. Hence, $d_m^c \geqslant d_j^{c-1}$. Then $\max(d_m^c) \geqslant \max(d_j^{c-1})$, and $D^c \geqslant D^{c-1}$ follows from Definition 7. The upper limit of detectability is trivially D^n, since the entire structure is then included in the Checking Window.

□

If the Checking Window includes all the nodes of the structure, LCED procedure degenerates into a global error detection procedure, which requires $O(n)$ execution time. Therefore, to achieve maximum local concurrent error detectability, it is sufficient to use a W_j^c with minimum size c for which $D^c = D^n$.

The LCED procedures mentioned throughout this section were unspecified because the actual procedure used depends on the particular data structure to be checked. The general LCED

technique is as follows. First, determine the appropriate Checking Window W_j^c that achieves the desired local concurrent error detectability. For each possible move from each node, identify the Lock functions and associated Key arguments that are used to perform the checking. The LCED procedure can be constructed as follows: for each move made, access the nodes defined by the Checking Window, and evaluate all the checkable Lock functions. If all Locks return "True," then either no error has occurred or undetectable errors have occurred; if any Lock returns "False," then at least one error has been detected. Once an error has been detected by an LCED procedure, LCEC may be performed. The upper limit of correctability is $\left\lfloor \frac{D^c}{2} \right\rfloor$. However, the actual correctability depends upon the data structure.

Since errors are detected and corrected based only on information from nodes in the Checking Window, many other detectable errors may exist simultaneously throughout the data structure. Although the local concurrent error detectability and correctability may only be one or two in the window, the actual number of detectable and correctable errors may be much larger.

III. VIRTUAL BACKPOINTERS

The *virtual backpointer* is a distributed checking symbol that can be used to achieve O(1) LCED and O(1) LCEC during a forward move, and O(1) LCED and O(n) LCEC during a backward move in many linked data structures. In addition, it can be used to generate a backpointer from a node N_i to its parent N_{parent}. In the general case, a virtual backpointer may point to an ancestor $N_{ancestor}$ of a node N_i, where $N_{ancestor}$ is an *ancestor* of N_i if there exists a series of moves from $N_{ancestor}$ to N_i.

DEFINITION 8: In a linked data structure, let $N_{ancestor}$ be an ancestor of N_i, and Q_i be the set of all pointers in N_i. The *virtual backpointer* $V_i = f(Q_i, A_{ancestor})$, where f is a function such that $A_{ancestor} = f^*(Q_i, V_i) = f^*(Q_i, f(Q_i, A_{ancestor}))$, and f^* is a companion function determined by f. In

general, there may be vectors of virtual backpointers, $\vec{V}_i = \vec{f}(Q_i, \vec{A})$, which, after suitable transformation by $\vec{f}^*$, point to vectors of nodes $\vec{A}$.

□

The virtual backpointer has the following properties. 1) For a forward move $N_i \rightarrow N_{i+1}$, V_{i+1} provides checking information. 2) For a backward move $N_{i+1} \sim N_i$, V_{i+1} provides the backpointer after transformation by f^*, and $Q_{ancestor}$ is used as checking information. Two example data structures employing the virtual backpointer are presented in the following subsections: the Virtual Double-Linked List, which is derived from the double-linked list, and the B-Tree with Virtual Backpointers, which is derived from the B-tree.

A. Virtual Double-Linked List

The *Virtual Double-Linked List* (VDLL) is a data structure that employs the virtual backpointer and possesses local concurrent error detectability and correctability. Errors are detected in O(1) time with an LCED procedure. For a forward move, detected errors may be corrected using LCEC in O(1) time; for a backward move, detected errors may be corrected using LCEC in $O(n)$ time. The VDLL requires no more storage space than the double-linked list (DLL), and retains the simplicity of the DLL, in that it is possible to move directly from a node to its parent, using the virtual backpointer. This is not possible, for example, in the modified(k) DLL [1], for $k \geqslant 2$, which must access other ancestors of a node in order to reach the node's parent.

DEFINITION 9: A *Virtual Double-Linked List* is described as follows (Figure 2). In a linked list data structure, let N_{i-1} be the parent of N_i, and P_i be the forward pointer of the N_i, therefore $Q_i = \{P_i\}$. Let $f(\{x\}, y) = f^*(\{x\}, y) = x \oplus y$, then $V_i = P_i \oplus A_{i-1} = A_{i+1} \oplus A_{i-1}$, and $A_{i-1} = P_i \oplus V_i$, where $\oplus$ denotes the logical exclusive-or function. Also, c header nodes $N_0, N_{-1}, \cdots, N_{-c+1}$ are added, where c is the size of the Checking Window. These header nodes are assumed to be always accessible by the LCED procedure. Note that $N_{-c+1} = N_\infty$.

□

The VDLL is created from the DLL by replacing its backpointers with virtual backpointers. The same operation can be applied to the modified(k) DLL family [1], resulting in the modified(k)

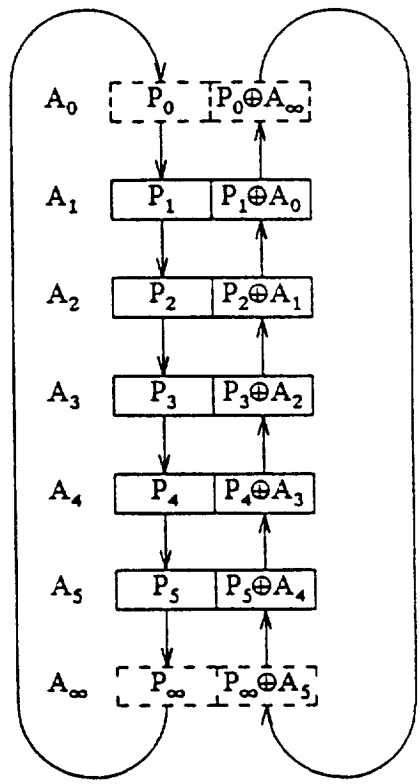


Figure 2. Virtual Double-Linked List (VDLL) of 5 nodes.

VDLL structures. It will be shown that each modified(k) VDLL achieves greater local concurrent error detectability than the corresponding modified(k) DLL.

DEFINITION 10: A *modified(k) Virtual Double-Linked List* is described as follows. In a linked list data structure, let N_{i-k} be the kth ancestor of N_i, and P_i be the forward pointer of the N_i, therefore $Q_i = \{P_i\}$. Let $f(x, y) = f^*(\{x\}, y) = x \oplus y$, then $V_i = P_i \oplus A_{i-k} = A_{i+1} \oplus A_{i-k}$, and $A_{i-k} = P_i \oplus V_i$. Also, max(k+1, c) header nodes, are added. □

The possible Locks and Keys of the VDLL can be identified as follows (Figure 2). For a forward move $N_i \rightarrow N_{i+1}$ following P_i,

$$\text{Keys} = \langle A_i, P_{i+1} \oplus V_{i+1} \rangle$$

$$\text{CLock}_{N_i \rightarrow N_{i+1}}(x, y) = (x \ ?= g(y)) = (x \ ?= y),$$

where g is the identity function. For the backward move $N_{i+1} \sim N_i$ following $V_{i+1} \oplus P_{i+1}$.

$$\text{Keys} = \langle A_{i+1}, A_i \rangle$$

$$\text{CLock}_{N_{i+1} \sim N_i}(x, y) = (x \ ?= g(y)) = (x \ ?= y.P),$$

where g retrieves the pointer P from the node at y. Locks and Keys for the modified(k) VDLL can be identified similarly. Using the results of the analysis of LCED, we now determine the local concurrent error detectability of the VDLL.

THEOREM 3: Using an LCED procedure, the local concurrent error detectability of the VDLL is $D^2(\text{forward}) = D^2(\text{backward}) = D^2 = 1$, and $D^c(\text{forward}) = D^c(\text{backward}) = D^c = D^3 = 2$, $\forall\, c \geqslant 3$.

PROOF: Since the VDLL uses virtual backpointers and Circular Locks, by Theorem 1, $D^c(\text{forward}) = D^c(\text{backward})$. Consider a forward move MV, $N_i \rightarrow N_{i+1}$, following P_i. The LCED procedure attempts to verify this move. A pointer that does not point to a logical node boundary can easily be detected by the node access routine. Therefore consider only erroneous pointers that lead to valid logical node addresses. Suppose that P_i is erroneous and leads to N_{j+1} instead of N_{i+1}. In $W_1^2 = \{N_i, N_{j+1}\}$, $d_1^2 = 2$: either V_{j+1} or P_{j+1} must be erroneous to mask the error in P_i. Assume that V_{j+1} is erroneous (Figure 3a). In $W_1^3 = \{N_i, N_{j+1}, N_{j+2}\}$, $d_1^3 = 2$. However, in $W_2^3 = \{N_{i-1}, N_i, N_{j+1}\}$, V_i will lead to the detection of the error in P_i, because following the backpointer given by $V_i \oplus P_i$ will lead to a node N_{k-1} instead of N_{i-1}, and $P_{k-1} \neq N_i$. Therefore, V_i must be changed into the value $A_{j+1} \oplus A_{i-1}$ to mask the error in P_i. Thus $d_2^3 = 3$.

Assume now that V_{j+1} is not erroneous, so P_{j+1} must be erroneous (Figure 3b). Consider $W_1^3 = \{N_i, N_{j+1}, N_{k+2}\}$. The LCED procedure will not detect the error in P_i if P_{j+1} has been changed to $A_{k+2} = A_i \oplus V_{j+1}$, and $V_{k+2} \oplus P_{k+2}$ has been changed (via a change in either V_{k+2} or P_{k+2}) to A_{j+1}. The remainder of the analysis is similar to the case above, and gives $d_1^2 = 2$, $d_1^3 = 3$, and $d_2^3 = 3$. According to Definition 7, $D^2 = 1$ and $D^3 = 2$. Since the VDLL can be changed to another correct VDLL by three pointer errors (node deletion), $D^n = 2$, where n is the number of nodes in the struc-

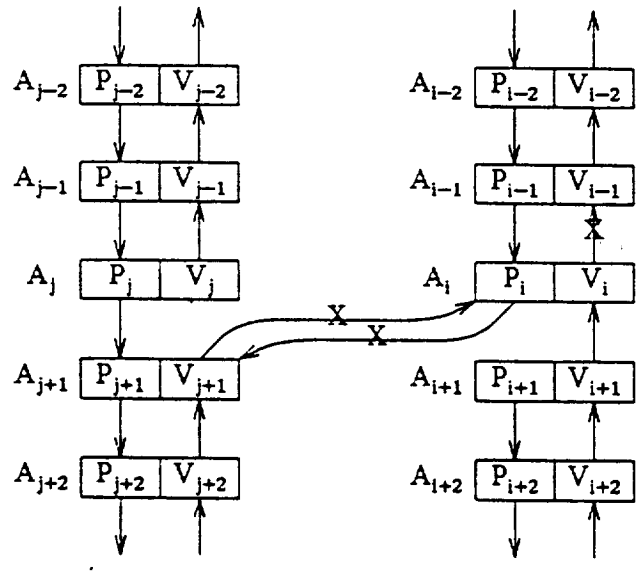


Figure 3a. Analysis of VDLL: Errors in P_i, V_i, and V_{j+1}.

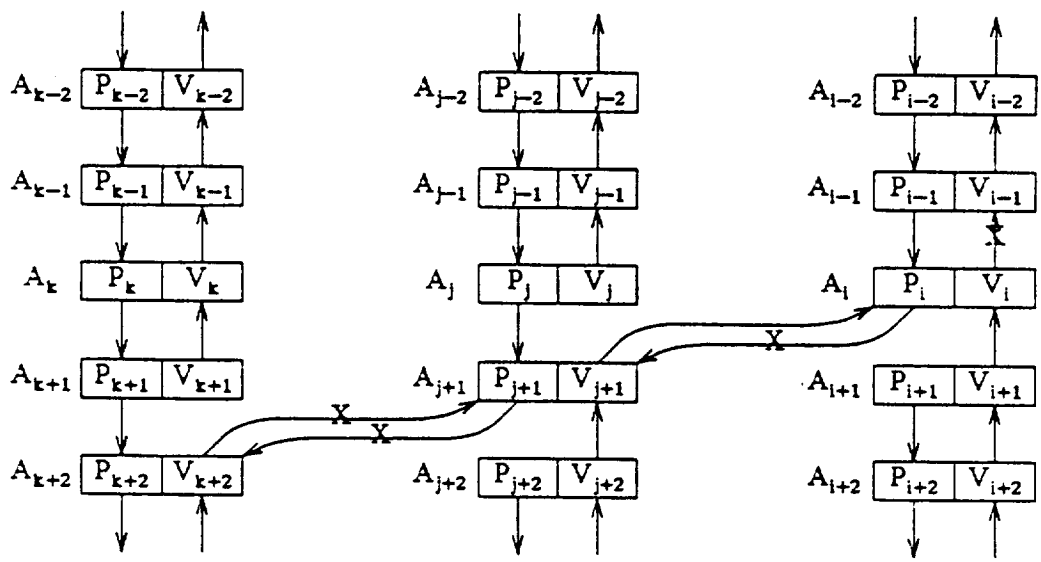


Figure 3b. Analysis of VDLL: Errors in P_i, V_i, P_{j+1}, and V_{k+2}.

ture. By Theorem 2, $D^c = 2, \forall c \geqslant 3$.

□

The above proof suggests that when moving forward $N_i \rightarrow N_{MV}$ following P_i, use $W^3 = \{N_{prev}, N_i, N_{MV}\}$ as the Checking Window, where N_{prev} corresponds to N_{i-1} in the proof; and when moving backward $N_i \sim N_{MV}$ following $P_i \oplus V_i$, use $W^3 = \{N_i, N_{MV}, N_{next}\}$ as the Checking Window, where N_{next} is the node reached by following $P_{MV} \oplus V_{MV}$. By using these windows, double pointer errors can be detected, or single pointer errors corrected (described below). The LCED procedure using this Checking Window evaluates four locks when moving either forward or backward. For a forward move, the locks are: L1: $A_{prev} ?= P_i \oplus V_i$, L2: $A_i ?= P_{MV} \oplus V_{MV}$, L3: $A_i ?= P_{prev}$ and L4: $A_{MV} ?= P_i$. For a backward move, the locks are: L1: $A_{next} ?= P_{MV} \oplus V_{MV}$, L2: $A_{MV} ?= P_i \oplus V_i$, L3: $A_{MV} ?= P_{next}$ and L4: $A_i ?= P_{MV}$. (In the W^2 Checking Window, only two locks are evaluated, namely $A_i ?= P_{MV} \oplus V_{MV}$ and $A_{MV} ?= P_i$ for the forward move, and $A_{MV} ?= P_i \oplus V_i$ and $A_i ?= P_{MV}$ for the backward move.) A comparison of local concurrent error detectability is given in Table 1 for the VDLL, modified(2) VDLL, modified(3) VDLL, DLL without a global count, and modified(2) and modified(3) DLL without global counts [1], for various sized Checking Windows. The local detectability of the modified(2) and modified(3) VDLL can be obtained using

Table 1. Local Concurrent Error Detectability of Several Linked List Data Structures.

Local Detectability	VDLL	mod(2) VDLL	mod(3) VDLL	DLL	mod(2) DLL	mod(3) DLL
D^2	1	0	0	1	0	0
D^3	2	1	0	1	1	0
D^4	2	2	1	1	2	1
D^5	2	3	2	1	2	2
D^6	2	3	3	1	2	3
D^7	2	3	4	1	2	3

the same analysis technique as that applied to VDLL. Any modified(k) VDLL achieves greater local concurrent error detectability than the corresponding modified(k) DLL. For $k > 3$, no further improvement in detectability can be made for either of the two families.

THEOREM 4: Any single pointer error detected by a forward move in $W^3 = \{N_{prev}, N_i, N_{MV}\}$ in a VDLL can be corrected with an O(1) LCEC procedure requiring at most one extra node access for both diagnosis and correction. Any single pointer error detected by a backward move in $W^3 = \{N_{next}, N_{MV}, N_i\}$ in a VDLL can be corrected with an $O(n)$ LCEC procedure requiring at most one extra node access for diagnosis.

PROOF: Since the local concurrent error detectability for this structure using W^3 is $D^3 = 2$, the upper limit of correctability is 1. Assume that a single error has been detected during a forward move. The LCED procedure supplies the values of the four detection locks (Table 2a), and three error indication values generated by a node access routine, NA_{prev}, NA_i, NA_{MV}, that indicate out-of-bounds pointers or pointers that do not point to logical node boundaries, when used to access N_{prev}, N_i and N_{MV}, respectively. There are eight possible errors: 1) A_{prev} error, 2) P_{prev} error, 3) A_i error, 4) P_i error, 5) V_i error, 6) A_{MV} error, 7) P_{MV} error and 8) V_{MV} error. To distinguish the eight errors, the seven-tuple syndrome {L1, L2, L3, L4, NA_{prev}, NA_i, NA_{MV}} is constructed (Table 2b). For the error-free case, the syndrome will be {True, True, True, True, True, True, True}. There are two cases of identical syndromes for different errors. In each case one extra node is accessed to completely diagnose the error. N_X is accessed by following P_{MV} to distinguish a P_{MV} error from a V_{MV} error. N_Y is accessed by following $P_i \oplus V_i$ to distinguish an A_{prev} error from a V_i error. Once the error has been diagnosed, correction proceeds as follows:

1) A_{prev} error: correct value is $P_i \oplus V_i$.

2) P_{prev} error: correct value is A_i.

3) A_i error: correct value is P_{prev}.

4) P_i error: correct value is A_{MV}.

Table 2a. Detection and Diagnosis Locks for Forward Moves in the VDLL using W^3.

Detection Locks		
L1	A_{prev} ?= $P_i \oplus V_i$	
L2	A_i ?= $P_{MV} \oplus V_{MV}$	
L3	A_i ?= P_{prev}	
L4	A_{MV} ?= P_i	
Diagnosis Locks		
L5	A_{MV} ?= $P_X \oplus V_X$	Access N_X via P_{MV}
L6	A_i ?= P_Y	Access N_Y via $P_i \oplus V_i$

Table 2b. Error Detection and Diagnosis Syndromes for Errors Detected by Forward Moves in the VDLL using W^3.

error	L1	L2	L3	L4	NA_{prev}	NA_i	NA_{MV}	L5	L6
A_{prev}	F	T	T	T	T	T	T	-	T
P_{prev}	T	T	F	T	T	T	T	-	-
A_i	T	F	F	T	T	T	T	-	-
P_i	F	T	T	T	T	T	F	-	-
	F	F	T	T	T	T	T	-	-
	F	F	T	T	T	T	F	-	-
V_i	F	T	T	T	T	T	T	-	F
A_{MV}	T	T	T	F	T	T	T	-	-
P_{MV}	T	F	T	T	T	T	T	F	-
V_{MV}	T	F	T	T	T	T	T	T	-

5) V_i error: correct value is $A_{prev} \oplus P_i$.

6) A_{MV} error: correct value is P_i.

7) P_{MV} error: correct value is $A_i \oplus V_{MV}$.

8) V_{MV} error: correct value is $A_i \oplus P_{MV}$.

Assume now that a single error has been detected during a backward move. The LCED procedure supplies the values of the four detection locks (Table 3a), and three error indication values

generated by a node access routine, NA_{next}, NA_{MV}, NA_i, that indicate out-of-bounds pointers or pointers that do not point to logical node boundaries, when used to access N_{next}, N_{MV} and N_i, respectively. There are eight possible errors: 1) A_{next} error, 2) P_{next} error, 3) A_{MV} error, 4) P_{MV} error, 5) V_{MV} error, 6) A_i error, 7) P_i error and 8) V_i error. To distinguish the eight errors, the seven-tuple syndrome {L1, L2, L3, L4, NA_{next}, NA_{MV}, NA_i} is constructed (Table 3b). For the error-free case, the syndrome will be {True, True, True, True, True, True, True}. There are two cases of identical syndromes for different errors. In each case one extra node is accessed to

Table 3a. Detection and Diagnosis Locks for Backward Moves in the VDLL using W^3.

Detection Locks		
L1	A_{next} ?= $P_{MV} \oplus V_{MV}$	
L2	A_{MV} ?= $P_i \oplus V_i$	
L3	A_{MV} ?= P_{next}	
L4	A_i ?= P_{MV}	
Diagnosis Locks		
L5	A_{next} ?= $P_X \oplus V_X$	Access N_X via P_{next}
L6	A_i ?= $P_Y \oplus V_Y$	Access N_Y via P_i

Table 3b. Error Detection and Diagnosis Syndromes for Errors Detected by Backward Moves in the VDLL using W^3.

error	L1	L2	L3	L4	NA_{next}	NA_{MV}	NA_i	L5	L6
A_{next}	F	T	T	T	T	T	T	-	-
P_{next}	T	T	F	T	T	T	T	F	-
A_{MV}	T	F	F	T	T	T	T	-	-
P_{MV}	T	T	T	F	F	T	T	-	-
	T	T	F	F	T	T	T	-	-
	T	T	F	F	F	T	T	-	-
V_{MV}	T	T	T	T	F	T	T	T	-
	T	T	F	T	T	T	T	-	-
	T	T	F	T	F	T	T	-	-
A_i	T	T	T	F	T	T	T	-	-
P_i	T	F	T	T	T	T	T	-	F
V_i	T	F	T	T	T	T	T	-	T

completely diagnose the error. N_X is accessed by following P_{next} to distinguish a P_{next} error from a V_{MV} error. N_Y is accessed by following P_i to distinguish a P_i error from a V_i error. Once the error has been diagnosed, correction proceeds as follows:

1) A_{next} error: correct value is $P_{MV} \oplus V_{MV}$.

2) P_{next} error: correct value is A_{MV}.

3) A_{MV} error: correct value is P_{next}.

4) P_{MV} error: correct value is A_i.

5) V_{MV} error: To correct the error in V_{MV}, first access the headers of the structure. Next, move forward, accessing nodes $N_0, N_1, \cdots, N_k$, performing W^3 LCED and correcting single errors with O(1) LCEC, until $P_k = A_{MV}$. Then the correct value of $V_{MV} = A_k \oplus P_{MV}$.

6) A_i error: correct value is P_{MV}.

7) P_i error: correct value is $A_{MV} \oplus V_i$.

8) V_i error: correct value is $A_{MV} \oplus P_i$. □

Note that for a forward move, both diagnosis and correction are O(1) time, and require one extra node access. For a backward move, diagnosis is O(1) time (one extra node access) but correction requires $O(n)$ extra node accesses in the worst case. Thus, O(1) LCEC is possible for an error detected by a forward move, while $O(n)$ LCEC is possible for an error detected by a backward move. The proof assumed that W^3 LCED was used; if W^2 is used instead, then diagnosis for both the forward and backward moves is still O(1), but correction for both moves requires $O(n)$ LCEC.

B. B-Tree with Virtual Backpointers

The *B-Tree with Virtual Backpointers* (VBT) of order m is a data structure that possesses local concurrent error detectability and correctability. Errors are detected in O(1) time if the time complexity is measured as a function of the number of nodes in the tree, i.e., n. For a forward move,

detected errors can be corrected using O(1) LCEC; for a backward move, detected errors can be corrected using $O(\log_{2m} n)$ LCEC. The VBT requires m+4 extra fields in each node, and has the additional feature that backward traversal can be performed without a stack, using the virtual backpointer.

The underlying structure of the VBT is the B-tree of order m [16], which finds application in the construction and maintenance of large-scale search trees. The B-tree has the following characteristics:

1) Every node contains at most $2m$ keys, and every node except the root contains at least m keys. The root contains at least one key.

2) Every node is either a leaf node, with no pointers to other nodes, or an internal node, with pointers to other internal nodes or to leaf nodes.

3) All leaf nodes appear at the same level.

4) An internal node with k keys will have k+1 pointers to subtrees. The k keys will be arranged in strictly increasing order, and keys in the ith subtree will be less than the ith key, while keys in the $i+1^{th}$ subtree will be greater than the ith key.

Let $P_{i,j}$ be the jth pointer in node N_i. Assume that each pointer requires one word of memory. Therefore, each pointer is uniquely addressable by $A_{i,j}$ (Figure 4a). The VBT is modified from the B-tree in the following ways to achieve local concurrent error detectability.

1) A header node N_0 is created with $P_{0,j} = A_{1,j}$ for $0 \leqslant j \leqslant 2m$.

2) V_i, the virtual backpointer of N_i, is defined as $V_i = P_{i,0} \oplus P_{i,1} \oplus \cdots \oplus P_{i,2m} \oplus A_{parent,j}$ where the jth pointer in N_{parent} points to N_i. For the special case of the virtual backpointer from the root to the header, V_1 is defined on $A_{0,0}$, even though all $P_{0,j}$ point to N_1.

3) The keys of N_i (i.e., $K_{i,1}, K_{i,2}, \cdots, K_{i,2m}$) are arranged in a matrix (Figure 4b) and the *key check symbols* $X_{i,j}$ and $Y_{i,j}$ are generated using a product code [17] as follows:

$$X_{i,j} = K_{i,(j-1)m+1} \oplus K_{i,(j-1)m+2} \oplus \cdots \oplus K_{i,(j-1)m+m} \quad , 1 \leqslant j \leqslant 2$$

$$Y_{i,j} = K_{i,j} \oplus K_{i,m+j} \quad , 1 \leqslant j \leqslant m.$$

$K_{i,j}$ is used to determine $X_{i,\mathrm{int}((j-1)/m)+1}$ and $Y_{i,(j-1) \bmod m + 1}$, called its *corresponding* X and Y check symbols, respectively.

The number of key fields used in N_i is called $count_i$, which is added for performance enhancement. A VBT of order 2 is illustrated in Figure 4c. The possible Locks and Keys of the VBT can be identified as follows. Assuming the jth pointer of N_i points to N_k, for a forward move $N_i \rightarrow N_k$ following $P_{i,j}$,

$$\text{Keys} = \langle A_{i,j}, (P_{k,0} \oplus P_{k,1} \oplus \cdots \oplus P_{k,2m} \oplus V_k) \rangle$$

$$\text{CLock}_{N_i \rightarrow N_k}(x, y) = (x \ ?= g(y)) = (x \ ?= y),$$

where g is the identity function. For the backward move $N_k \sim N_i$ following $(P_{k,0} \oplus P_{k,1} \oplus \cdots \oplus P_{k,2m} \oplus V_k)$,

$$\text{Keys} = \langle A_k, A_{i,j} \rangle$$

$$\text{CLock}_{N_k \sim N_i}(x, y) = (x \ ?= g(y)) = (x \ ?= y.P_j),$$

where g retrieves the jth pointer $P_{i,j}$ from the node at y.

We now determine the local concurrent error detectability of the VBT, employing the results of the analysis of LCED. Using Theorem 2, Table 4 presents the possible key and pointer errors that can occur in the VBT (errors in the *count* field are covered by the fifth and sixth rows of the table), and the number of errors required to mask them, assuming an LCED procedure is used.

THEOREM 5: Using an LCED procedure, the local concurrent error detectability of the VBT is $D^2 = 1$ and $D^3 = D^c = 2, \forall c \geqslant 3$.

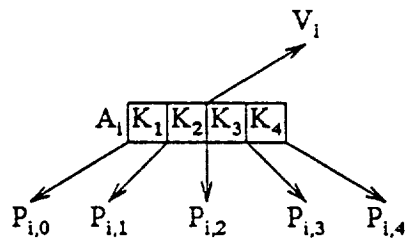


Figure 4a. Node Representation in Order-2 B-Tree with Virtual Backpointers.

$K_{i,1}$	$K_{i,2}$	...	$K_{i,m}$	$X_{i,1}$
$K_{i,m}+1$	$K_{i,m}+2$	...	$K_{i,2m}$	$X_{i,2}$
$Y_{i,1}$	$Y_{i,2}$	...	$Y_{i,m}$	

$V_i = P_{i,0} \oplus P_{i,1} \oplus \cdots \oplus P_{i,2m} \oplus A_{parent,j}$
$count_i =$ number of key fields used in N_i

Figure 4b. Virtual Backpointer and Key Check Symbols in a VBT Node.

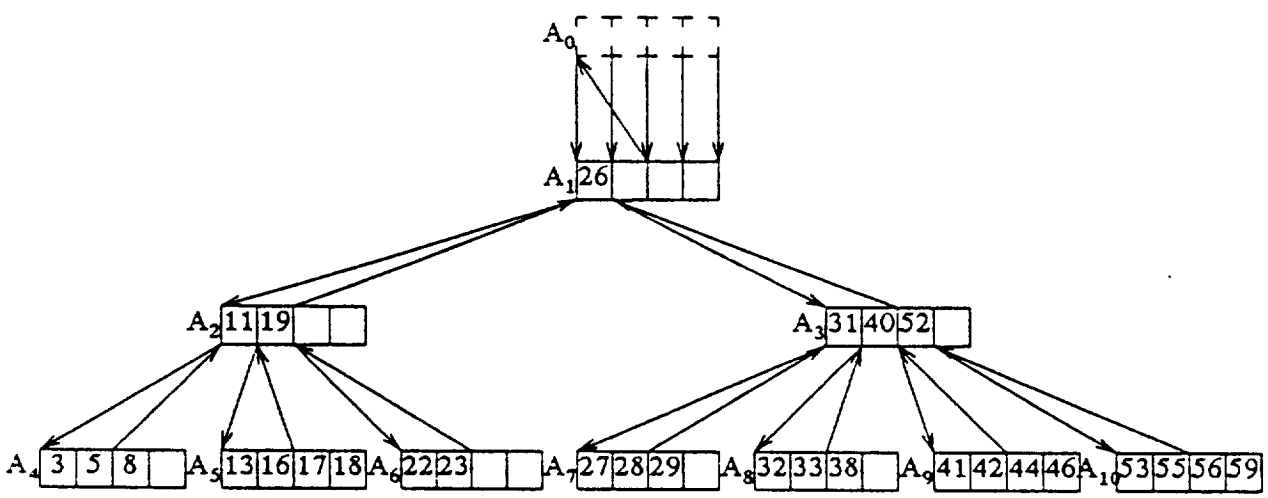


Figure 4c. Order-2 B-Tree with Virtual Backpointers (VBT).

PROOF: From Table 4, the minimum $d_j^2 = 2$ and the minimum $d_j^c = 3$, $\forall\ c \geqslant 3$. From Definition 7, it follows that $D^2 = 1$ and $D^c = 2$, $\forall\ c \geqslant 3$. □

From Table 4 it can be seen that no increase in the local concurrent error detectability can be gained by using W^c for $c \geqslant 3$. It can be shown that when moving forward $N_i \rightarrow N_{MV}$ following $P_{i,j}$, or when moving backward $N_i \sim N_{MV}$ following $(P_{MV,0} \oplus P_{MV,1} \oplus \cdots \oplus P_{MV,2m} \oplus V_{MV})$, use $W^3 = \{N_{prev}, N_i, N_{MV}\}$ and $W^3 = \{N_i, N_{MV}, N_{next}\}$ respectively, to achieve detection of double pointer errors, or correction of single pointer errors (described below). In the window for the forward move, N_{prev} is the parent of N_i, and in that for the backward move, N_{next} is the parent of N_{MV}. The LCED procedure using this window evaluates four locks. For a forward move, the locks are: L1: $A_{prev,r} ?= P_{i,0} \oplus P_{i,1} \oplus \cdots \oplus P_{i,2m} \oplus V_i$, L2: $A_{i,j} ?= P_{MV,0} \oplus P_{MV,1} \oplus \cdots \oplus P_{MV,2m} \oplus V_{MV}$, L3: $A_i ?= P_{prev,r}$ and L4: $A_{MV} ?= P_{i,j}$. For a backward move, the locks are: L1: $A_{next,s} ?= P_{MV,0} \oplus P_{MV,1} \oplus \cdots \oplus P_{MV,2m} \oplus V_{MV}$, L2: $A_{MV,t} ?= P_{i,0} \oplus P_{i,1} \oplus \cdots \oplus P_{i,2m} \oplus V_i$, L3: $A_{MV} ?= P_{next,s}$ and L4: $A_i ?= P_{MV,t}$. (In the W^2 Checking Window, only two locks are evaluated, namely $A_{i,j} ?= P_{MV,0} \oplus P_{MV,1} \oplus \cdots \oplus P_{MV,2m} \oplus V_{MV}$ and $A_{MV} ?= P_{i,j}$ for the forward move, and $A_{MV,t} ?= P_{i,0} \oplus P_{i,1} \oplus \cdots \oplus P_{i,2m} \oplus V_i$ and $A_i ?= P_{MV,t}$ for the backward move).

Table 4. Analysis of Errors in the VBT.

Error Condition	$\max(d_j^2)$	$\max(d_j^3)$	$\max(d_j^c)$ $\forall\ c \geqslant 4$
Non-empty VBT becomes empty	$2m+1$	$2m+1$	$2m+1$
Empty VBT becomes non-empty	$2m+2$	$2m+2$	$2m+2$
Key, X or Y becomes erroneous	3	3	3
Internal node's non-null pointer points to incorrect node	2	3	3
Internal node's non-null pointer becomes null	6	6	6
Internal node's null pointer becomes non-null	6	7	7
Two of internal node's pointers exchanged	2	4	4
Internal node becomes a leaf node	3	3	3
Leaf node becomes an internal node	3	4	5

THEOREM 6: Any single pointer error detected by a forward move in $W^3 = \{N_{prev}, N_i, N_{MV}\}$ can be corrected with at most $2m+1$ extra node accesses in O(1) time. Any single pointer error detected by a backward move in $W^3 = \{N_i, N_{MV}, N_{next}\}$ can be corrected in $O(\log_{2m} n)$ time if it is detected during a backward move.

PROOF: Since the local concurrent error detectability of this structure in W^3 is $D^3 = 2$, the upper limit of correctability is 1. Assume that the error detected is a single error. The error may be a key, a key check symbol, a *count* or a pointer. For the key or key check symbol error, diagnosis and correction are performed using the procedures for product codes [17]. For a *count* error, all the keys and key check symbols will be correct, hence counting the non-null keys will regenerate the *count*.

For the pointer error, if the erroneous pointer is located at the header node, it can be corrected by simple comparison because there are $2m+1 \geqslant 3$ identical pointers in the header. Otherwise, there are two cases: detection by a forward move and detection by a backward move. Assume that the error has been detected during the forward move from N_i to N_{MV} following $P_{i,j}$. The LCED procedure supplies the values of the four detection locks (Table 5a), and three error indication values generated by a node access routine, NA_{prev}, NA_i, NA_{MV}, that indicate out-of-bounds pointers or pointers that do not point to logical pointer boundaries, when used to access N_{prev}, N_i and N_{MV}, respectively. There are nine possible errors: 1) A_{prev} error, 2) $P_{prev,r}$ error where $P_{prev,r}$ is the pointer from N_{prev} to N_i, 3) A_i error, 4) $P_{i,j}$ error, 5) $P_{i,s}$ error for $0 \leqslant s \leqslant 2m$ and $s \neq j$, 6) V_i error, 7) A_{MV} error, 8) $P_{MV,t}$ error for $0 \leqslant t \leqslant 2m$, and 9) V_{MV} error. To distinguish the nine errors, the seven-tuple syndrome $\{L1, L2, L3, L4, NA_{prev}, NA_i, NA_{MV}\}$ is constructed (Table 5b). For the error-free case, the syndrome will be {True, True, True, True, True, True, True}. There are two cases of identical syndromes for different errors. In each case extra nodes are accessed to completely diagnose the error. The nodes N_X^t are accessed by following all the pointers $P_{MV,t}$ from N_{MV} to distinguish a $P_{MV,t}$ error from a V_{MV} error. N_Y is accessed by following $P_{i,0} \oplus P_{i,1} \oplus \cdots \oplus P_{i,2m} \oplus V_i$ to distinguish an A_{prev} error from a V_i error or a $P_{i,s}$ error. The latter two errors are distinguished

Table 5a. Detection and Diagnosis Locks for Forward Moves in the VBT using W^3.

	Detection Locks	
L1	$A_{prev,r} ?= P_{i,0} \oplus \cdots \oplus P_{i,2m} \oplus V_i$	
L2	$A_{i,j} ?= P_{MV,0} \oplus \cdots \oplus P_{MV,2m} \oplus V_{MV}$	
L3	$A_i ?= P_{prev,r}$	
L4	$A_{MV} ?= P_{i,j}$	
	Diagnosis Locks	
L5	$\prod_{t=0}^{count_{MV}} (A_{MV,t} ?= P^t_{X,0} \oplus \cdots \oplus P^t_{X,2m} \oplus V^t_X)$	Access N^t_X via $P_{MV,t}$ for $0 \leqslant t \leqslant 2m$
L6	$A_i ?= P_{Y,r}$	Access N_Y via $P_{i,0} \oplus \cdots \oplus P_{i,2m} \oplus V_i$
L7	$\prod_{s=0}^{count_i} (A_{i,j} ?= P^s_{Z,0} \oplus \cdots \oplus P^s_{Z,2m} \oplus V^s_Z)$	Access N^s_Z via $P_{i,s}$ for $0 \leqslant s \leqslant 2m$ and $s \neq j$

Table 5b. Error Detection and Diagnosis Syndromes for Errors Detected by Forward Moves in the VBT using W^3.

error	L1	L2	L3	L4	NA_{prev}	NA_i	NA_{MV}	L5	L6	L7
A_{prev}	F	T	T	T	T	T	T	-	T	-
$P_{prev,r}$	T	T	F	T	T	T	T	-	-	-
A_i	T	F	F	T	T	T	T	-	-	-
$P_{i,j}$	F	T	T	T	T	T	F	-	-	-
	F	F	T	T	T	T	T	-	-	-
	F	F	T	T	T	T	F	-	-	-
$P_{i,s}$ $(s \neq j)$	F	T	T	T	T	T	T	-	F	F
V_i	F	T	T	T	T	T	T	-	F	T
A_{MV}	T	T	T	F	T	T	T	-	-	-
$P_{MV,t}$	T	F	T	T	T	T	T	F	-	-
V_{MV}	T	F	T	T	T	T	T	T	-	-

by accessing the nodes N^s_Z by following all the pointers $P_{i,s}$ from N_i. Once the error has been diagnosed, correction proceeds as follows:

1) A_{prev} error: compute $A_{prev,r}$ from $P_{i,0} \oplus P_{i,1} \oplus \cdots \oplus P_{i,2m} \oplus V_i$, from which A_{prev} can be calculated.

2) $P_{prev,r}$ error: correct value is A_i.

3) A_i error: correct value is $P_{prev,r}$.

4) $P_{i,j}$ error: correct value is A_{MV}.

5) $P_{i,s}$ error: correct value is $A_{prev,r} \oplus P_{i,0} \oplus \cdots \oplus P_{i,s-1} \oplus P_{i,s+1} \oplus \cdots \oplus P_{i,2m} \oplus V_i$.

6) V_i error: correct value is $A_{prev,r} \oplus P_{i,0} \oplus P_{i,1} \oplus \cdots \oplus P_{i,2m}$.

7) A_{MV} error: correct value is $P_{i,j}$.

8) $P_{MV,t}$ error: correct value is $A_{i,j} \oplus P_{MV,0} \oplus \cdots \oplus P_{MV,t-1} \oplus P_{MV,t+1} \oplus \cdots \oplus P_{MV,2m} \oplus V_{MV}$.

9) V_{MV} error: correct value is $A_{i,j} \oplus P_{MV,0} \oplus P_{MV,1} \oplus \cdots \oplus P_{MV,2m}$.

Assume now that the error has been detected during a backward move from N_i to N_{MV} following $P_{i,0} \oplus P_{i,1} \oplus \cdots \oplus P_{i,2m} \oplus V_i$. The LCED procedure supplies the values of the four detection locks (Table 6a), and three error indication values generated by a node access routine, NA_{next}, NA_{MV}, NA_i, that indicate out-of-bounds pointers or pointers that do not point to logical pointer boundaries, when used to access N_{next}, N_{MV} and N_i, respectively. There are eight possible errors: 1) A_{next} error, 2) $P_{next,s}$ error where $P_{next,s}$ is the pointer from N_{next} to N_{MV}, 3) A_{MV} error, 4) $P_{MV,t}$ error for $0 \leqslant t \leqslant 2m$, 5) V_{MV} error, 6) A_i error, 7) $P_{i,j}$ error for $0 \leqslant j \leqslant 2m$, and 8) V_i error. To distinguish the eight errors, the seven-tuple syndrome {L1, L2, L3, L4, NA_{next}, NA_{MV}, NA_i} is constructed (Table 6b). For the error-free case, the syndrome will be {True, True, True, True, True, True, True}. There are two cases of identical syndromes for different errors. In each case extra nodes are accessed to completely diagnose the error. The nodes N_X^j are accessed by following all the pointers $P_{i,j}$ from N_i to distinguish a $P_{i,j}$ error from a V_i error. N_Y is accessed by following $P_{next,0} \oplus P_{next,1} \oplus \cdots \oplus P_{next,2m} \oplus V_{next}$ to distinguish a $P_{next,s}$ error from a V_{MV} error or a $P_{MV,t}$ error. The latter two errors are distinguished by accessing the nodes N_Z^t by following all the pointers $P_{MV,t}$ from N_{MV}. Once the error has been diagnosed, correction proceeds as follows:

1) A_{next} error: compute $A_{next,s}$ from $P_{MV,0} \oplus P_{MV,1} \oplus \cdots \oplus P_{MV,2m} \oplus V_{MV}$, from which A_{next} can be calculated.

2) $P_{next,s}$ error: correct value is A_{MV}.

Table 6a. Detection and Diagnosis Locks for Backward Moves in the VBT using W^3.

	Detection Locks	
L1	$A_{next,s} ?= P_{MV,0} \oplus \cdots \oplus P_{MV,2m} \oplus V_{MV}$	
L2	$A_{MV,t} ?= P_{i,0} \oplus \cdots \oplus P_{i,2m} \oplus V_i$	
L3	$A_{MV} ?= P_{next,s}$	
L4	$A_i ?= P_{MV,t}$	
	Diagnosis Locks	
L5	$\prod_{j=0}^{count_i} (A_{i,j} ?= P^j_{X,0} \oplus \cdots \oplus P^j_{X,2m} \oplus V^j_X)$	Access N^j_X via $P_{i,j}$ for $0 \leqslant j \leqslant 2m$
L6	$A_{next} ?= P_{Y,r}$	Access N_Y via $P_{next,0} \oplus \cdots \oplus P_{next,2m} \oplus V_{next} = A_{Y,r}$
L7	$\prod_{t=0}^{count_{MV}} (A_{MV,t} ?= P^t_{Z,0} \oplus \cdots \oplus P^t_{Z,2m} \oplus V^t_Z)$	Access N^t_Z via $P_{MV,t}$ for $0 \leqslant t \leqslant 2m$

Table 6b. Error Detection and Diagnosis Syndromes for Errors Detected by Backward Moves in the VBT using W^3.

error	L1	L2	L3	L4	NA_{next}	NA_{MV}	NA_i	L5	L6	L7
A_{next}	F	T	T	T	T	T	T	-	-	-
$P_{next,s}$	T	T	F	T	T	T	T	-	F	-
A_{MV}	T	F	F	T	T	T	T	-	-	-
$P_{MV,t}$	T	T	F	T	T	T	T	-	T	F
	T	T	F	T	F	T	T	-	T	F
V_{MV}	T	T	F	T	T	T	T	-	T	T
	T	T	F	T	F	T	T	-	T	T
A_i	T	T	T	F	T	T	T	-	-	-
$P_{i,j}$	T	F	T	T	T	T	T	F	-	-
V_i	T	F	T	T	T	T	T	T	-	-

3) A_{MV} error: correct value is $P_{next,s}$.

4) $P_{MV,t}$ error: To correct the error in $P_{MV,t}$, first access the headers of the structure. Next, move forward, accessing nodes $N_0, N_1, \cdots, N_k$, performing W^3 LCED and correcting single errors with O(1) LCEC, until $P_{k,s} = A_{MV}$. Then the correct value of $P_{MV,t}$ is $A_{k,s} \oplus P_{MV,0} \oplus \cdots \oplus P_{MV,t-1} \oplus P_{MV,t+1} \oplus \cdots \oplus P_{MV,2m} \oplus V_{MV}$.

5) V_{MV} error: To correct the error in V_{MV}, first access the headers of the structure. Next, move forward, accessing nodes $N_0, N_1, \cdots, N_k$, performing W^3 LCED and correcting single errors with O(1) LCEC, until $P_{k,s} = A_{MV}$. Then the correct value of V_{MV} is $A_{k,s} \oplus P_{MV,0} \oplus P_{MV,1} \oplus \cdots \oplus P_{MV,2m}$.

6) A_i error: correct value is $P_{MV,t}$.

7) $P_{i,j}$ error: correct value is $A_{MV,t} \oplus P_{i,0} \oplus \cdots \oplus P_{i,j-1} \oplus P_{i,j+1} \oplus \cdots \oplus P_{i,2m} \oplus V_i$.

8) V_i error: correct value is $A_{MV,t} \oplus P_{i,0} \oplus P_{i,1} \oplus \cdots \oplus P_{i,2m}$. □

The robust B-tree [3] presented by Black, Taylor and Morgan performs double error detection or single error correction in $O(n)$ time, and requires $2m+3$ extra fields in each node of an order-m B-tree. Taylor and Black have also developed the LB-Tree [10] which is locally correctable, in that it can correct many single errors if they occur in separate substructures. However, in order to verify a pointer, one level of nodes must be traversed, and to correct a pointer, all the levels above the current level must be traversed. Hence, double error detection and single error correction require $O(n)$ time, and $2m+5$ extra fields in each node of an order-m B-tree are required. In comparison, the advantages of the VBT are as follows:

1) Double pointer errors can be detected in the VBT using an O(1) LCED procedure.

2) Single pointer errors can be corrected in the VBT using an O(1) LCEC procedure for an error detected during a forward move, or using an $O(\log_{2m} n)$

LCEC procedure for an error detected during a backward move.

3) The VBT requires only m+4 extra fields in each node.

4) The virtual backpointer facilitates backward traversals of the VBT, which can then be used to enhance performance.

IV. ANALYSIS AND IMPLEMENTATION OF A CONCURRENT AUDITOR PROCESS

The *Concurrent Auditor Process* (CAP) is an on-line process for error detection and correction that runs in parallel with user processes accessing a database. It is used, in this case, to perform data structure error detection and correction for the user processes, and allows concurrent access to structures being checked to reduce the system performance degradation due to error detection. Koved and Waldbaum have developed an auditor program that provides detection of computer subsystem failures [18], based on Waldbaum's concept of the auditor program [19]. Taylor, Morgan and Black have suggested the use of an audit program to periodically perform error detection and correction in data structures [1]. However, little analysis has been performed on the effectiveness of such an audit program. This section presents an analysis of the effectiveness of the CAP and presents measurements of the CAP's effectiveness in a Sequent Balance 8000 multiprocessor implementation using a database of VDLL.

The CAP described here accesses structures more frequently and uniformly than user processes to reduce the latency of error detection. Also, the CAP performs error detection in Checking Windows of higher cost than those used by user processes, to reduce their performance degradation. For example, if the database is composed of VDLL or VBT instances, user processes may perform single pointer error detection in W^2 with less computation cost, while relying on the CAP to detect the less-frequent double pointer errors in W^3 with more computation cost. The effectiveness of the CAP is determined by its increase of the mean time to failure (MTTF) of the

system. Ideally, a large increase is achieved with little degradation of user process performance. Hence, the CAP permits user processes to access structures being checked as long as they do not insert or delete nodes from the CAP's current Checking Window. Expressions are derived to determine the MTTF in a multi-user, n-process system with and without the use of the CAP. This is followed by the results of an implementation of the CAP using a VDLL database.

A. Analysis

In a multi-user, n-process shared-database environment, assume that the CAP performs error detection in W^3 and that user processes perform error detection in W^2. The pointer errors can then be divided into three classes: E_0, E_1 and E_2. E_0 errors are those which can be detected by a user process or by the CAP. E_1 errors can be detected by the CAP but not by a user process. E_2 errors can be detected by neither a user process nor the CAP. Suppose the time for an E_1 error to occur is $T^n_{E_1}$, the time for a user process to encounter that error is T_U, and the time for the CAP to detect an E_1 error is T_A. For the purposes of analysis assume, in a given time interval, both the number of errors that occur and the number of accesses to a particular node are random variables following a Poisson distribution. Then, random variables $T^n_{E_1}$, T_U and T_A follow an exponential distribution with mean time γ_1^n, β and α, respectively.

LEMMA 1: The probability of an E_1 error causing any of the n processes to fail in the presence of the CAP is $1-\left(\frac{\beta}{\alpha+\beta}\right)^n$.

PROOF: For a single process, the probability to fail can be derived using basic probability theory:

$$\mathrm{Prob}(T_A > T_U) = \int_0^\infty \mathrm{Prob}(T_U = x)\mathrm{Prob}(T_A > x)dx = \int_0^\infty \frac{1}{\beta} e^{-x/\beta} e^{-x/\alpha} dx = \frac{\alpha}{\alpha+\beta}.$$

Therefore, the probability of any of the n processes failing is $1-\left(1-\frac{\alpha}{\alpha+\beta}\right)^n = 1-\left(\frac{\beta}{\alpha+\beta}\right)^n$. □

THEOREM 7: Without the use of the CAP, MTTF $= \gamma_1^n + \beta$, and with the use of the CAP,

$$\mathrm{MTTF}_{\mathrm{CAP}} = \min\left(\frac{\gamma_1^{n'}}{1-\left(\frac{\beta}{\alpha+\beta}\right)^n}, \gamma_2^{n'}\right) + \beta.$$

PROOF: If no CAP is used, MTTF $= \min(E(T_{E_1}^n), E(T_{E_2}^n)) + E(T_U) = \min(\gamma_1^n, \gamma_2^n) + \beta = \gamma_1^n + \beta$, where E(X) is the expected value of random variable X.

In the presence of the CAP, the determination of whether an E_1 error will cause a failure can be modeled as a Bernoulli trial with parameter $p = 1-\left(\frac{\beta}{\alpha+\beta}\right)^n$. Hence the $\mathrm{MTTF}_{\mathrm{CAP}}$ follows a geometric distribution with mean $\frac{\gamma_1^{n'}}{p}$, where n' represents the effect of n user processes and the CAP.

□

If E_1 and E_2 errors are formed by the accumulation of E_0 errors, then $T_{E_1}^n$ and $T_{E_2}^n$ are proportional to the access frequency. Thus $\gamma_1^n = n\gamma_1^1$, $\gamma_2^n = n\gamma_2^1$ and $\gamma_2^1 >> \gamma_1^1$. This gives, for the without-CAP case, MTTF $= \gamma_1^n + \beta = n\gamma_1^1 + \beta$. In the with-CAP case, since the CAP is $\frac{\beta}{\alpha}$ times faster in checking the data structure than a user process, $\gamma_1^{n'} = \left(n+\frac{\beta}{\alpha}\right)\gamma_1^1$. E_2 errors will retain an exponential distribution but with different mean $\gamma_2^{n'} = \left(n+\frac{\beta}{\alpha}\right)\gamma_2^1$. For this case the theorem gives

$$\mathrm{MTTF}_{\mathrm{CAP}} = \min\left(\frac{\left(n+\frac{\beta}{\alpha}\right)\gamma_1^1}{p}, \left(n+\frac{\beta}{\alpha}\right)\gamma_2^1\right) + \beta.$$

EXAMPLE 4: Suppose γ_1^1 = 100 hours, γ_2^1 = 10,000 hours, β = 1 minute, and 5 user processes are active on the system. Without the use of the CAP, MTTF $\simeq$ 500 hours. However, by using the CAP, and with α = 10 seconds, MTTF is increased to $MTTF_{CAP}$ =2050 hours. □

If α is small enough (i.e., the CAP is fast enough), the $\dfrac{\left(n+\frac{\beta}{\alpha}\right)\gamma_1^1}{1-\left(\frac{\beta}{\alpha+\beta}\right)^n}$ term can exceed the $\left(n+\frac{\beta}{\alpha}\right)\gamma_2^1$ term. In this case, $MTTF_{CAP} = \left(n+\frac{\beta}{\alpha}\right)\gamma_2^1+\beta$. This effectively eliminates the chances of a user process failure due to E_1 errors, which occur more often than E_2 errors.

B. Implementation

A model database of VDLL was implemented in C and run on a Sequent Balance 8000 shared-memory multiprocessor system with six CPUs. Single random errors and worst-case double errors (called "double cooperative errors," where a second error masks a previous error) were injected into the database one at a time. Error detection was accomplished by one of four user processes, the database manager, or the CAP, each of which performed either W^2 or W^3 checking. The database manager serviced all update requests, and the CAP operated in the idle time of the database manager, to reduce performance degradation. Databases of 50, 100, 500 and 1000 nodes were used in the simulations. Each database consisted of eight VDLL instances: six non-empty instances, one empty instance, and a free list. To model the locality of user process database access, each user process performed approximately 80% of its operations (composed of 75% searches, 12.5% insertions and 12.5% deletions) within one VDLL, and the other 20% in a randomly selected VDLL.

For each single or double error injected, the detection latency and the number of operations completed in that time were measured, for five different combinations of user process LCED/CAP LCED (Table 7). The mean error detection latencies for the five combinations, applied to databases

of 50, 100, 500 and 1000 nodes, are shown in Table 8. Table 9 shows by what factor use of the CAP can decrease the error detection latency. The following observations can be made based on the results of the implementation:

1) Single and double LCED can be performed on the VDLL in O(1) time.

2) The use of the CAP significantly reduces the error detection latency of both single random errors and double cooperative errors.

3) The CAP is more effective in reducing the detection latency of single random errors as the size of the database increases.

Using the analysis results of the previous section, the first observation shows that $\gamma_1^{n'} \geqslant 5\gamma_1^n$. Thus from Theorem 5, the $MTTF_{CAP} > 5 \times MTTF$. This clearly shows the utility of the CAP in increasing the MTTF of the system.

V. SUMMARY

In this paper, we have presented a new technique for local concurrent error detection in linked data structures that can achieve O(1) error detection in a variety of data structures. This technique uses the concept of a Checking Window to define the locality in which local concurrent error detection is performed and also to determine the associated cost of the locality. The virtual backpointer was introduced and used to define two new data structures, the Virtual Double-Linked List, which incurs no storage overhead, and the B-Tree with Virtual Backpointers of order m, which requires m+4 extra fields per node. It was shown that double errors could be detected using a local concurrent error detection procedure in O(1) time for both structures. In addition, those errors detected during forward moves were shown to be correctable using a local concurrent error correction procedure in O(1) time. Correction of those errors detected during backward moves was shown to be, in worst case, O(n). Finally, an analysis and implementation of a concurrent auditor

Table 7. Combinations of User Process LCED and CAP LCED.

Case	User Process LCED	CAP LCED
1	W^2	None
2	W^2	W^2
3	W^2	W^3
4	W^3	None
5	W^3	W^3

Table 8. Mean Error Detection Latencies.

Error Type	Database Size	Number of Samples	Case 1	2	3	4	5
Single Random Error	50	10000	77	8	7	64	7
	100	10000	144	11	10	127	10
	500	1800	4884	147	134	5052	140
	1000	200	29087	372	308	31033	312
Double Cooperative Error	50	10000	72	7	7	39	7
	100	10000	60	13	10	57	11
	500	1800	420	54	48	447	50

Table 9. Detection Latency Reduction Factor Through Use of the CAP.

Error Type	Database Size	Cases Compared 1:2	1:3	4:5
Single Random Error	50	10	11	9
	100	13	14	13
	500	33	37	36
	1000	78	94	99
Double Cooperative Error	50	10	10	6
	100	5	6	5
	500	8	9	9

process in a shared database using the virtual backpointer technique was shown to significantly reduce the error detection latency.

REFERENCES

[1] D. J. Taylor, D. E. Morgan, and J. P. Black, "Redundancy in Data Structures: Improving Software Fault Tolerance," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 6, pp. 585-594, November 1980.

[2] D. J. Taylor, D. E. Morgan, and J. P. Black, "A Compendium of Robust Data Structures," *Proceedings of the 11th Annual International Symposium on Fault Tolerant Computing*, pp. 129-131, June 1981.

[3] J. P. Black, D. J. Taylor, and D. E. Morgan, "A Robust B-Tree Implementation," *Proceedings of the 5th International Conference on Software Engineering*, pp. 63-70, March 1981.

[4] J. I. Munro and P. V. Poblete, "Fault Tolerance and Storage Reduction in Binary Search Trees," *Information and Control*, vol. 62, pp. 210-218, 1984.

[5] M. C. Sampaio and J. P. Sauvé, "Robust Trees," *Proceedings of the 15th Annual International Symposium on Fault Tolerant Computing*, pp. 23-28, June 1985.

[6] S. C. Seth and R. Muralidhar, "Analysis and Design of Robust Data Structures," *Proceedings of the 15th Annual International Symposium on Fault Tolerant Computing*, pp. 14-19, June 1985.

[7] K. Yoshihara, Y. Koga, and T. Ishihara, "A Robust Data Structure Scheme with Checking Loops," *Proceedings of the 13th Annual International Symposium on Fault Tolerant Computing*, pp. 241-248, June 1983.

[8] K. Kuspert, "Efficient Error Detection Techniques for Hash Tables in Database Systems," *Proceedings of the 14th Annual International Symposium on Fault Tolerant Computing*, pp. 198-203, June 1984.

[9] J. P. Black and D. J. Taylor, "Local Correctability in Robust Storage Structures," to appear: *IEEE Transactions on Software Engineering*.

[10] D. J. Taylor and J. P. Black, "A Locally Correctable B-Tree Implementation," *The Computer Journal*, vol. 29, no. 3, pp. 269-276, 1986.

[11] I. J. Davis and D. J. Taylor, "Local Correction of Mod(k) Lists," CS-85-55, Dept. of Computer Science, University of Waterloo, December 1985.

[12] I. J. Davis, "Local Correction of Helix(k) Lists," CS-86-30, Dept. of Computer Science, University of Waterloo, August 1986.

[13] I. J. Davis, "A Locally Correctable AVL Tree," to appear: *17th Annual International Symposium on Fault-Tolerant Computing*, July 1987.

[14] W. K. Fuchs, "A Specification-Based Approach to Concurrent Structure Verification in Multiprocessor Systems," *IEEE International Conference on Computer Design*, pp. 375-378, October 1986.

[15] D. J. Taylor, D. E. Morgan, and J. P. Black, "Redundancy in Data Structures: Some Theoretical Results," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 6, pp. 595-602, November 1980.

[16] R. Bayer and E. McCreight, "Organization and Maintenance of Large Ordered Indexes," *Acta Informatica*, vol. 1, no. 3, pp. 173-189, 1972.

[17] P. Elias, "Error-Free Coding," *IRE Transactions on Information Theory*, vol. IT-4, pp. 29-37, 1954.

[18] L. Koved and G. Waldbaum, "Improving Availability of Software Subsystems through On-Line Error Detection," *IBM Systems Journal*, vol. 25, no. 1, pp. 105-115, 1986.

[19] G. Waldbaum, "Audit programs - A Proposal for Improving System Availability," Research Report RC-2811, IBM Thomas J. Watson Research Center, February 1970.

87N29850

MEASUREMENT-BASED RELIABILITY/PERFORMABILITY MODELS

BY

MEI-CHEN HSUEH

B.S., Providence College of Arts and Science, 1972
M.S., Portland State University, 1981

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1987

Urbana, Illinois

PRECEDING PAGE BLANK NOT FILMED

MEASUREMENT-BASED RELIABILITY/PERFORMABILITY MODELS

Mei-Chen Hsueh, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1987
Ravi Iyer, Advisor

This thesis describes measurement-based models based on real error-data collected on a multi-processor system. Models development from the raw error-data to the estimation of cumulative reward is described.

A workload/reliability model is developed based on low-level error and resource usage data collected on an IBM 3081 system during its normal operation in order to evaluate the resource-usage/error/recovery process in a large mainframe system. Thus, both normal and erroneous behavior of the system are modeled. The results provide an understanding of the different types of errors and recovery processes. The measured data show that the holding times in key operational and error states are not simple exponentials and that a semi-Markov process is necessary to model the system behavior. A sensitivity analysis is performed to investigate the significance of using a semi-Markov process, as opposed to a Markov process, to model the measured system.

A software reliability model is also developed based on low-level error data from the MVS operating system running on an IBM 3081 machine to describe the software error and recovery process. The semi-Markov model developed provides a quantification of system error characteristics and the interaction between different types of errors. As an example, we provide a detailed model and analysis of multiple errors, which constitute approximately 17% of all software errors and result in considerable recovery overhead.

In addition, a measurement-based performability model based on real error-data collected is proposed. A reward function, based on the service rate and the error rate in each state, is defined in order to estimate the performability of the system and, to depict the cost of different error types and recovery procedures.

ACKNOWLEDGEMENT

I am sincerely grateful to my thesis advisor, Ravi Iyer, for his guidance and encouragement. His assistance was of tremendous value. I would also like to give thanks to Professor Kishor Trivedi of Duke University for making available facilities at Duke and for valuable suggestions which provided additional breadth to this work.

Being a member of the Computer Systems Group has provided me with a continuous source of encouragement, suggestions, support and friendship. I would especially like to thank Pat Duba, Mark Sloan and Rene Llames for their valuable suggestions in the drafting of this thesis and Joe Rahmeh for his technical assistance.

Finally, I would like to thank my mother for the extensive care she provided my family, thereby making it possible for me to do my graduate work.

TABLE OF CONTENTS

1 INTRODUCTION 1

1.1. Thesis Objectives 1

1.2. Related Research 2

1.3. Thesis Overview 4

2 RESOUCE-USAGE/ERROR/RECOVERY MODELING 6

2.1. Workload Modeling 6

2.1.1. Resource Usage Characterization 6

2.1.2. Workload Clustering 7

2.1.3. Resource Usage Model 10

2.2. Error Modeling 13

2.2.1. Error Clustering 14

2.3. Recovery Modeling 15

2.4. Resource-Usage/Error/Recovery Model 17

2.4.1. Waiting and Holding Time Distributions 19

2.4.2. Recovery Distributions 23

2.5. Summary 25

3 MODEL ANALYSIS 26

3.1. Model Parameters 26

3.2. Model Behavior 32

3.3. Effect of Workload 35

3.4. Model Validation 37

3.5. Markov Versus Semi-Markov 39

3.5.1. Unconditional transition probability (γ) 39

3.5.2. First Passage Time (Θ) 42

3.5.3. Summary 44

4 SOFTWARE RELIABILITY MODEL 45

4.1. Introduction 45

4.1.1. Related Research 46

4.2. Error Characterization 47

4.2.1. Multiple Errors 48

4.2.2. Recovery Modeling 49

4.3. Software Reliability Model 54

4.3.1. Overall Error/Recovery Model 54

4.3.2. Waiting Time Distributions 56

4.3.3. Recovery Time Distribution 58

4.3.4. Summary 60

4.4. Model Analysis 60

4.4.1. General Characteristics 60

4.4.2. Model Probabilities 62

4.4.3. Characteristics of A Multiple Error 63

4.5. Conclusion 65

5 PERFORMABILITY MODEL 66

5.1. Reward Function 66

5.2. Semi-Markov to Markov Conversion 69

5.3. Performability Analysis 70

6 SUMMARY AND CONCLUSIONS 78

6.1. Summary of Results 78

6.2. Suggestions for Future Research 81

REFERENCES 82

APPENDIX A 84

APPENDIX B 86

APPENDIX C 95

VITA 100

LIST OF TABLES

Table 2.1. Characteristics of workload clusters 9

Table 2.2. Frequency of errors 15

Table 2.3. Percentage distribution of recovery procedures 16

Table 2.4 Mean waiting time (in seconds) of states 21

Table 3.1. Mean time between errors 33

Table 3.2. Mean recurrence time 34

Table 3.3. Summary of model characteristics 35

Table 3.4. Holding time and transition probabilities to error states 36

Table 3.5. Comparison of occupancy probabilities for different states 38

Table 3.6. Comparison of transition probabilities, γ_{ij} 41

Table 3.7. Ratio of γ_{ij} (Markov/semi-Markov) 42

Table 3.8. Ratio of $\overline{\Theta^2}$ (Markov/semi-Markov) 44

Table 4.1. Frequency of software errors 48

Table 4.2. Percentages of recovery attempts for a software error 54

Table 4.3. Mean waiting time (in seconds) of states 56

Table 4.4. Mean time between errors 61

Table 4.5. Mean recurrence time of recovery 61

Table 4.6. Characteristics of software error/recovery model 62

Table 4.7. Characteristics of a multiple error 64

Table 5.1. Reward rates, r_i, for error states 67

LIST OF FIGURES

Figure 2.1. State-transition diagrams of CPU bound load 11

Figure 2.2 State-transition diagrams of I/O bound load 12

Figure 2.3. Flow chart of recovery processes 16

Figure 2.4. State-transition diagram of resource-usage/error/recovery model 17

Figure 2.5. State-transition diagram for multiple errors (MULT) 19

Figure 2.6. Waiting and holding time densities 22

Figure 2.7. Error duration densities 24

Figure 4.1. State-transition diagram for a multiple error 50

Figure 4.2. Reduced state-transition diagram of multiple errors 51

Figure 4.3. Flow of recovery 53

Figure 4.4. Software error/recovery model 55

Figure 4.5. Time to error density 57

Figure 4.6. Recovery time (error duration) densities 59

Figure 5.1. The conversion of non-exponential to a set of exponentials 71

Figure 5.2. The Markov conversion of State W_8 72

Figure 5.3. The expected reward rate, $E[X(t)]$ 75

Figure 5.4. The time-averaged accumulated reward, $E[Y(t)]/t$ 76

Figure 5.5. Distribution of accumulated reward until system failure 77

CHAPTER 1

INTRODUCTION

1.1. Thesis Objectives

The development of realistic models to describe the error behavior of computer systems is a difficult problem. Although many researchers have addressed the modeling issue and have significantly advanced the state of the art, there is little or no validation of these models with field data. It is, therefore, extremely valuable to model the error and recovery process in a production system using real error data. Apart from providing useful information on how errors occur, this process also provides insight into the interaction between various system components. Additionally, it will be seen that it also allows explicit modeling of the relationship between resource usage and hardware and software errors, an area that has yet to be fully explored.

In this research we build a state-transition model which describes the resource-usage/error/recovery process of a computer system. This model is based on low-level error and resource usage data collected on a production system. The data were collected on an IBM 3081 system during its normal operation. Both the normal and erroneous behavior of the system are modeled. The results, therefore, provide an understanding of the different error and recovery processes and their relationship to various types of resource usage. Hardware and software reliabilities and their interaction are also modeled. Results show

that the error and recovery process on our measured system is best described by a semi-Markov process.

1.2. Related Research

The primary motivation for this research is that there has been no attempt to explicitly model the resource-usage/error/recovery process based on real data. The only research is that in [1,2], where the authors proposed the use of a double stochastic Poisson process to model a cyclic load-error relationship. The model assumes that the instantaneous error rate can be described by a cyclostationary Gaussian process (i.e., the workload has a cyclic pattern). Thus only the external behavior has been modeled. Furthermore, only a single workload variable (time spent in the kernel mode) was modeled.

Analytical models for hardware failure have been extensively investigated [3,4,5,6,7,8]. Although the time for different components to fail is usually assumed to be exponentially distributed, time-dependent failure rates and graceful degradation have been considered along with performability issues. Repairability has been modeled by Trivedi, et. al., [3,5,6,8], all of which assume constant repair times. A job/task flow based model is described in [9]. Failure occurrence is assumed to be a linear function of the service requests from a job/task flow. As shown in [10], the assumption of linearity may result in underestimating the effect of the workload, especially when the load is high.

Most software reliability models usually refer to the development, debugging and testing phases of the software as in [11,12] and [13,14]. Few of these models have been applied to the operational phase of the software. In [2] and [15], software failures in an operating environment are studied. Both studies found that at least 60% of system failures

are software related. Another study [16] shows that *undetected* software-related errors are due to either specification errors, implementation errors, or logic errors.

There is little explicit study of hardware/software reliability. The hardware/software interface is generally hard to model and experimental measurements are not easy to obtain and analyze. In [15], software failures in the operating system, which could be related to hardware problems, were analyzed and it was shown that errors in the hardware/software interface are often fatal. In [17], a methodology for joint hardware/software model construction and model processing using Stochastic Petri Nets is described.

With the exception of the software reliability growth models, which have been validated with real data, there are few, if any, models of software reliability in an operational environment. Exceptions include the hardware and software model discussed in [18] and a measurement-based model of workload dependent failures discussed in [10]. Both, however, only describe the external behavior of the system and do not provide insight into component level behavior.

It is therefore highly instructive to construct a detailed model based on low-level error data from a production system. Toward this end we have constructed a joint resource-usage/error/recovery model using error and resource usage data collected from an IBM system. The model provides detailed information on system behavior under normal and error conditions. Hardware and software failures of different severity are modeled. Multiple errors and the effect of on-line recovery routines are also considered.

1.3. Thesis Overview

A methodology for model construction based on real error data and resource usage information is described in Chapter 2. The model construction includes the resource usage (workload) characterization, error and recovery characterization, and modeling the overall system. For the workload characterization, we use a statistical clustering method to characterize the collected resource usages of the measured system from an n-tuple variable of infinite points into a few number of sets. Thus, a state-transition model of resource usages of the system is constructed based on these sets.

Different types of component errors and recovery procedures are also described in detail and classified in Chapter 2. A two-level error data reduction scheme is employed to identify individual error incidents and ensure that the analysis is not biased by error records relating to the same problem. The interaction of hardware and software errors is modeled in this chapter. The three models describing resource usage, error and recovery are then combined to form an overall model. The conditional transition probabilities as well as the sojourn times of states are estimated from real data. Results show that the resource-usage/error/recovery process is a semi-Markov process.

In Chapter 3 we perform four different kinds of model analyses to show the characteristics of the measured system. First, we use the model built in Chapter 2 to evaluate key characteristics of the system, such as the state occupancy probability and the unconditional transition probability from one specified state to another. These measures provide us with a very fair estimation of the model behavior. Second, we estimate the error probability due to the workload from the model. The analysis shows that the error probabilities appear to be not only a function of the resource usage, but are also related to the length of the sojourn time in a resource usage state. Third, the model validation is performed by

comparing the results predicted from the model with the values estimated from the actual observations. Finally, we perform an analysis to investigate the significance of using a semi-Markov process, as opposed to a Markov process, to model the measured system.

In Chapter 4 a measurement-based software reliability model is built. In addition to describing the software error and recovery process in the measured system this model also provides a quantification of software system error characteristics and the interaction between different types of software errors.

A performability model based on real data is proposed in Chapter 5. A reward function, based on the service rate and the error rate in each state, is defined in order to estimate the performability of the measured system and to depict the cost of different error types and recovery procedures. The conversion of a semi-Markov model to its Markov version is also demonstrated in this chapter. This conversion gives us the ability to use an existing system performance estimator to estimate the performability of the measured system.

In Chapter 6 we provide a summary of this research and highlight some important conclusions drawn from this work.

CHAPTER 2

RESOUCE-USAGE/ERROR/RECOVERY MODELING

2.1. Workload Modeling

In this section we build a state-transition model to describe the variation in system activity. It will later be shown that this approach allows an error to be considered as a transition from normal activity. System activity is characterized by a number of resource usage parameters. A statistical clustering technique is employed to reduce the potential many to many transitions of the workload vector to a small number of states representative of those found in the data. The data for our studies came from an IBM 3081 system running the MVS operating system. The system consists of dual processors with two time-multiplexed channel sets. Together these two sets allow a maximum of 24 subchannels to be simultaneously active in each I/O cycle.

2.1.1. Resource Usage Characterization

The workload data was collected using the IBM MVS/370 system Resource Management Facility (RMF) [19]. RMF is a flexible tool for measuring the performance of an IBM system. It measures data in two ways: by exact count and by sampling. The exact count method checks the appropriate system indicators at the beginning and the end of an interval and calculates the difference. The sampling method checks the appropriate system indicators at each cycle within an interval (e.g.. an interval may be one hour and a cycle may be

500 milliseconds). At the end of the interval the mass of data collected at each cycle is reduced to either minimum, maximum, and average values or to a percentage value. The results presented here are based on three months of sampled RMF data, with a cycle time of 500 milliseconds and an interval of one hour.

Four different resource usage measures were selected to represent the workload of three basic components of the computer hardware system.

CPU - fraction of the measured interval for which the CPU is executing instructions

CHB - fraction of the measured interval for which the channel was busy and the CPU was in the wait state (this parameter is usually used to measure the degree of contention in our system)

SIO - number of successful Start I/O and Resume I/O instructions issued to the channel

DASD - number of requests serviced on the direct access storage devices

Although several other measures were available, we decided to use only the measures listed above so as to keep the model trackable. The methodology presented here is easily extended to incorporate other measures.

2.1.2. Workload Clustering

At any interval of time the measured workload is represented by a point in 4-dimensional space, (CPU, CHB, SIO, DASD). Cluster analysis is used to divide the workload into similar classes according to a pre-defined criterion.[1] This allows us to concisely describe the dynamics of system behavior and extract a structure that already exists in the

[1]Potentially, we can have an uncountably large number of points in the workload space. Intuitively, only a countable number of combinations of four measures do in fact occur. Further, it is seen that they usually occur in clusters.

workload data.[2] Each cluster (defined by its centroid) is then used to depict a system state and a state-transition diagram (consisting of inter-cluster transition probabilities and cluster sojourn times) is developed.

A k-means clustering algorithm [21,22] was used for cluster analysis. Briefly, the algorithm partitions an N-dimensional population into k sets on the basis of a sample. It starts with k groups each of which consists of a single random point. Each new point is added to the group with the closest centroid. After a point is added to a group, the mean of that group is adjusted in order to take the new point into account. This process is repeated until the changes in the cluster means become negligibly small. Thus at each stage the k-means are, in fact, the means of the groups they represent. Therefore, k non-empty clusters, $C_1, C_2, \ldots, C_k$, are sought such that the sum of the squares of the Euclidean distances of the cluster members from their centroids is minimized, i.e.,

$$\sum_{j=1}^{k} \sum_{i} ||x_i - \bar{x}_j||^2 \rightarrow \text{minimum}$$

where $x_i \in C_i$ and $\bar{x}_j$ is the centroid of cluster C_j.

Two types of workload clusters were formed. In the first case CPU and CHB were selected to be the workload variables. This combination was found to best describe the CPU-bound load (nearly 60% of the observations have a CPU usage greater than 0.72). In the second case the clusters were formed considering SIO and DASD as workload variables. This combination was found to best describe the I/O workload. Table 2.1 shows the results for these two cases.

An examination of Table 2.1 also shows the dynamics of the measured system behavior. We see in Table 2.1(a) that about 36% of the time the CPU is highly loaded

[2] Similar clustering techniques are also used for workload characterization in [20].

Table 2.1. Characteristics of workload clusters

(a) CPU workload

Cluster id	% of obs	Mean of CPU	Mean of CHB	Std dev of CPU	Std dev of CHB
W_1	7.44	0.0981	0.1072	0.0462	0.0436
W_2	0.50	0.1126	0.5525	0.0433	0.0669
W_3	2.73	0.1547	0.2801	0.0647	0.0755
W_4	12.41	0.3105	0.1637	0.0550	0.0459
W_5	0.74	0.3639	0.3819	0.0365	0.1923
W_6	17.12	0.5416	0.1287	0.0560	0.0511
W_7	22.58	0.7207	0.0848	0.0576	0.0301
W_8	36.48	0.9612	0.0168	0.0362	0.0143
R^2 of CPU =	0.9724				
R^2 of CHB =	0.8095				
overall R^2 =	0.9604				

(b) I/O workload

Cluster id	% of obs	Mean of SIO	Mean of DASD	Std dev of SIO	Std dev of DASD
U_1	8.89	16.80	0.95	6.80	1.30
U_2	36.05	41.59	2.99	7.51	1.92
U_3	1.48	44.37	20.62	8.55	4.18
U_4	1.73	60.07	38.84	6.77	8.42
U_5	42.72	67.34	5.19	7.92	3.72
U_6	0.49	87.30	31.19	3.87	9.84
U_7	7.9	96.20	6.02	8.73	3.34
U_8	0.74	141.10	10.10	10.28	8.50
R^2 of SIO =	0.8861				
R^2 of DASD =	0.7176				
overall R^2 =	0.8751				

(0.96) and almost 76% of the time the CPU load is above 0.5. Since the measured system is a two-processor machine, we may say that 76% of the time at least one of the processors is busy. Note that, with increasing CPU usage, CHB (CPU wait and channel busy) decreases. This indicates that resource contention is not a problem in our measured system. In Table 2.1(b) (the I/O load), both clusters U_2 and U_3 have a very close channel start I/O rate (SIO) but the disk service rate (DASD) of U_3 is as much as 10 times that of U_2. This indicates that some I/O requests result in a burst of data while the others only in a few words. A burst transfer however occurred only 4% of the time ($U_3 + U_4 + U_6$). This result may be due to the fact that our measurements were made during work hours, but I/O-bound jobs are normally executed during off-work hours.

2.1.3. Resource Usage Model

State-transition diagrams of these two different types of workload clusters are shown in Figure 2.1 and Figure 2.2. The transition probabilities from state i to state j, $p_{i,j}$, are estimated from the measured data using:

$$p_{i,j} = \frac{\text{observed number of transitions from state } i \text{ to state } j}{\text{observed number of transitions from state } i} . \qquad (2.1.1)$$

These two figure provide us with not only the details of workload dynamics but also the interactions among clusters. Figure 2.1 shows that once the CPU load reached 0.5 (W_6), the transition of the greatest probability was to its next higher load (W_7) and the transition to its next lower load ($W_{4,5}$) occurred with the second greatest probability. This can be seen in states W_6, W_7, and W_8. However, when the CPU load is low (i.e., less than 0.5), the change to a higher load is much faster. For example, with 0.333 probability the CPU load changed from W_1 to $W_{4,5}$ and 0.424 probability from $W_{4,5}$ to W_7.

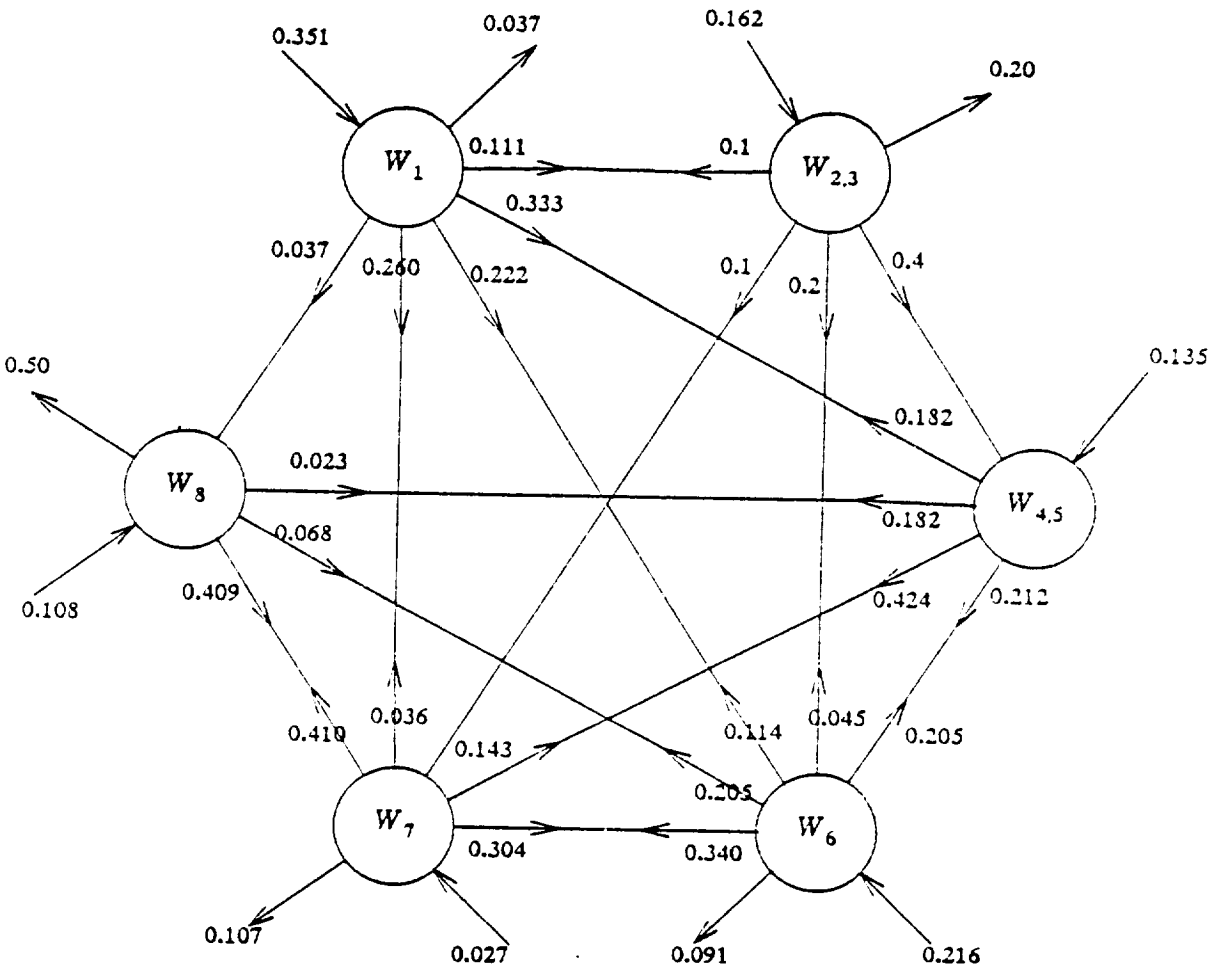


Figure 2.1. State-transition diagrams of CPU bound load

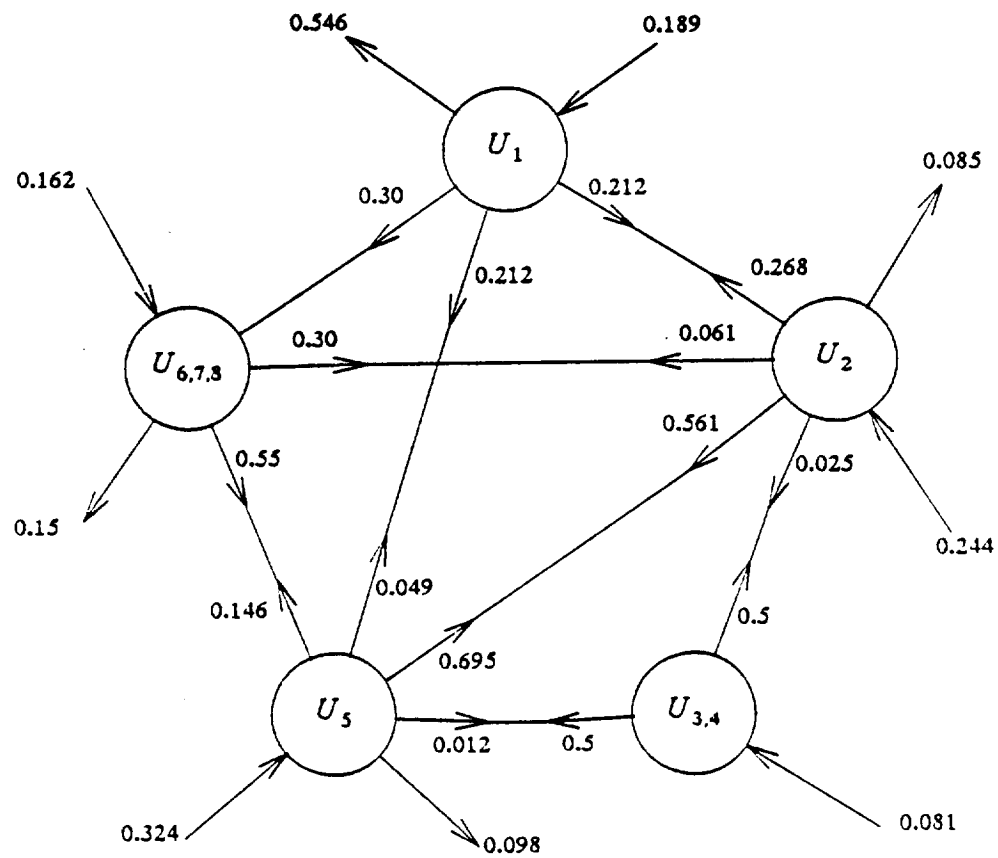


Figure 2.2 State-transition diagrams of I/O bound load

Figure 2.2 shows the transitions among various I/O loads. This figure confirms our previous observation that most often the I/O workload fluctuates back and forth between two moderate levels. U_2 and U_5 (0.69 and 0.56) and that there are occasional requests for burst I/O (0.025 from U_2 and 0.012 from U_5 to $U_{3,4}$).

2.2. Error Modeling

In this section the collection and characterization of errors is discussed. A state-transition diagram to describe different error states is developed. The measured system incorporates built-in error detection facilities, and many components also provide for recovery through retry or redundancy. The error and recovery information is logged into a permanent data set called LOGREC [23]. For each error, whether recoverable or not, the operating system creates a time-stamped record describing the error and providing relevant information on the state of the machine. In each record there are a number of bits describing the type of error, its severity, and the result of hardware and software attempts to recover from the problem. From this data six different types of errors were collected :

(1) CPU-related errors	- those that affect the normal operation of the CPU; the errors may originate in the CPU itself, in the main memory, or in a channel.
(2) Temporary channel errors	-those that are recovered by channel retry and do not result in the termination of the channel control program.
(3) Temporary (soft) disk errors	- those I/O errors that are recovered by correcting the data or by retrying the hardware instruction.
(4) Temporary (hard) disk errors	- those I/O errors that are recovered by software instruction retry or by a functional recovery routine(s).
(5) Permanent disk errors	- those I/O errors that are not correctable and can not be recovered by retrying the operation, and
(6) Software errors	- software incidents that are due to invalid supervisor calls, program checks and other software exception conditions.

2.2.1. Error Clustering

Due to the manner in which errors are detected and reported in a computer system, it is possible that a single fault may manifest itself as more than one error, depending on the activity at the time of the error. The different manifestations may not all be identical [24]. The system recovery usually treats these errors as isolated incidents. In order to address this problem and to ensure that the analysis is not biased by error records relating to the same problem, two levels of data reduction were performed.

First, a coalescing algorithm described in [10] was used to analyze the data and merge observations which occur in rapid succession and relate to the same problem. Next, a technique described in [24] to automatically group records most likely to have a common cause, was used (See Appendix A for the details).[3] By using these two methods, we classified errors into five different classes. These classes are called error events since they may contain more than one error and are defined as follows.

CPU : that caused errors to be logged as CPU-related errors

CHAN : that caused errors to be logged as channel errors

SWE : that caused errors to be logged as software errors

DASD : that caused errors to be logged as direct access storage device errors

MULT : that caused errors affecting more than one type of component

Table 2.2 lists the frequencies of different types of errors. In this table we found that about 80% of errors are disk and software errors. We also note that about 17% of the errors are classified as multiple errors (MULT). A MULT error is mostly due to a single cause but the fault has non-identical manifestations provoked by different types of system activity.

[3]Although this second reduction is not essential to this work, it allows us to notice several multiple errors which otherwise would not have been noticed.

Since the manifestations are non-identical, recovery may be complex and hence imposes considerable overhead on the system. It should be noted that such an error event (17% of our data) has not been modeled before.

2.3. Recovery Modeling

When an error is detected in the measured system, an appropriate recovery routine is invoked depending on the severity of the error. The recovery procedures were divided into four categories in increasing order of recovery cost. The recovery cost was measured in terms of the system overhead required to handle an error. The lowest level (hardware recovery), involves the use of an error correction code (ECC) or hardware instruction retry and has minimal overhead. If hardware recovery is not possible (or unsuccessful), the next level, i.e., software controlled recovery, is invoked. This could be simple, e.g., terminating the current program or task in control, or complex, e.g., invoking a specially designed recovery routine(s) to handle the problem. The third level of recovery (ALT) involves transferring the tasks to a functioning processor(s) when one of the processors experiences

Table 2.2. Frequency of errors

Type of error	Frequency	Percent
CPU	2	0.04
CHAN	119	2.23
MULT	924	17.33
SWE	1923	36.07
DASD	2364	44.34
total	5332	100.00

an un-recoverable error. If no on-line recovery is possible, the system is brought down for off-line repair. Figure 2.3 shows a flow chart of the recovery process. Table 2.3 lists the distribution of recovery levels. From Table 2.3 we note that about 73% of errors were

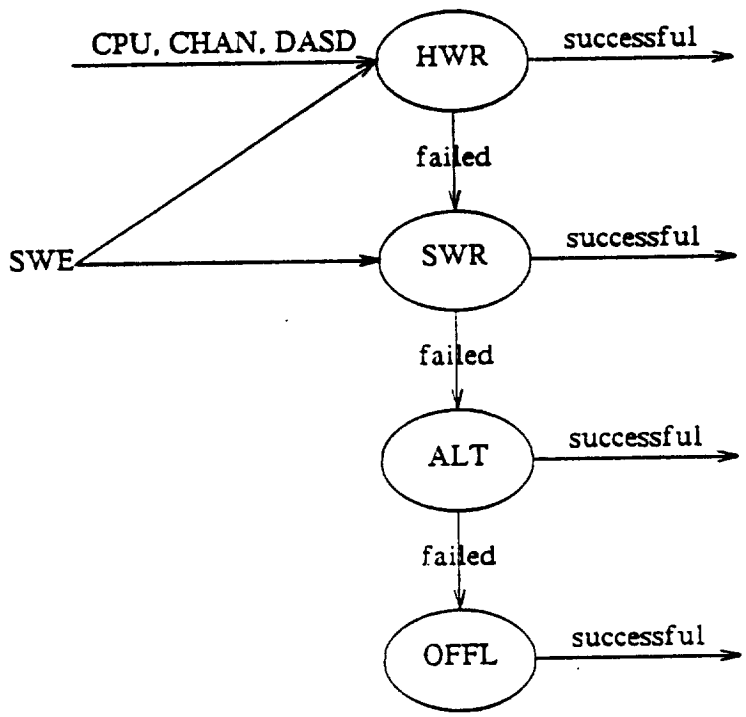


Figure 2.3. Flow chart of recovery processes

Table 2.3. Percentage distribution of recovery procedures

Recovery Procedure	Percent
HWR	73.35
SWR	26.56
ALT	0.02
OFFL	0.07

successfully handled through hardware recovery and most of the others were recovered from by use of the software recovery procedure.

2.4. Resource-Usage/Error/Recovery Model

In this section we combine the separate workload, error and recovery models, developed so far, into a single model shown in Figure 2.4. A null state W_0 is added to

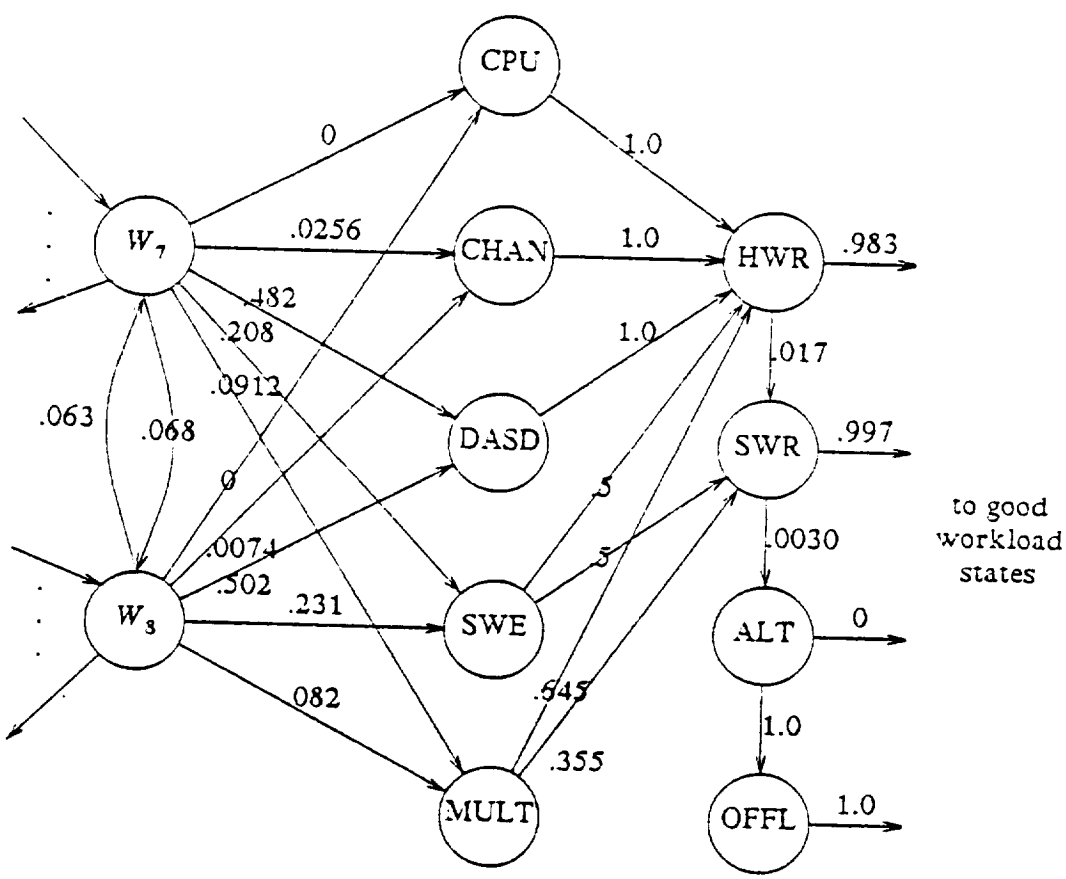


Figure 2.4. State-transition diagram of resource-usage/error/recovery model

represent the state during the non-measured period although it is not shown in Figure 2.4. The transition probabilities among states are estimated from the measured data using Equation 2.1.1. Notice that, unlike other models this describes both the normal and erroneous behavior of the system. The model has three different classes of states: normal operation states (S_N), error states (S_E), and recovery states (S_R). Note that the normal state has two different types of transitions: the first, to other normal states and the second, to error states.

Under normal conditions, the system makes transitions from one workload state to another. The occurrence of an error results in a transition to one of the error states. The system then goes into one or more recovery modes after which, with a high probability, it returns to one of the "good" workload states. The state-transition diagram of Figure 2.4 shows that nearly 98.3% of the hardware recovery requests and 99.7% of the software recovery requests are successful. Thus the error detection, fault isolation and on-line recovery mechanism allow the measured system to handle an error efficiently and effectively. In only less than 1% of the cases is the system not able to recover.

Figure 2.5 shows the state-transition diagram of a MULT error (a MULT event), i.e., given that a multiple error has occurred. The model shows that disk and software errors are strongly correlated in multiple errors. From the diagram, it is seen that in about 65% of the cases a multiple error starts as a software error (SWE) and in 32% of the cases it starts as a disk error (DASD). Given that a disk error has occurred there is nearly a 30% chance that a software error will follow. It is also interesting to note that there is a 64% chance that one software error will be followed by another different software error.

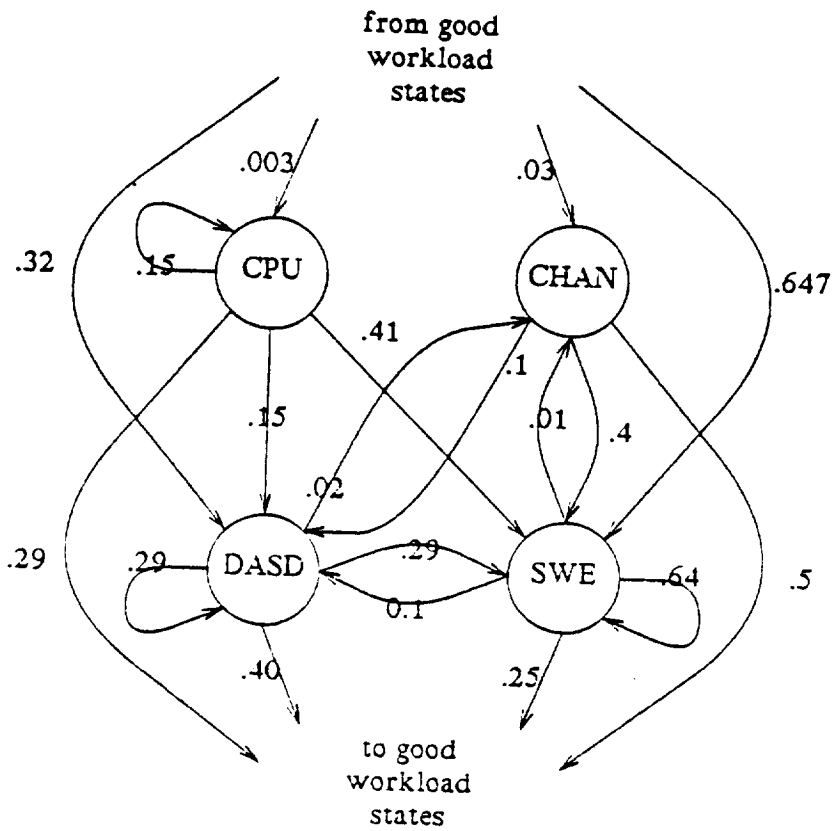


Figure 2.5. State-transition diagram for multiple errors (MULT)

2.4.1. Waiting and Holding Time Distributions

We used the state-transition diagram to show the relationship among the workload, error, and recovery processes in the measured system. We also showed the interactions among the errors. In this subsection we will present the characteristics of the measured system in terms of the state waiting and holding times.

The waiting time for state i is the time that the process spends in state i before making a transition. The holding time for a transition from state i to state j is the time that the process spends in state i before making a transition to state j [25]. Table 2.4 shows the mean waiting times of both the workload and error states. It is well-known that the mean and standard deviation of an exponential distribution are the same. Thus an examination of the mean and standard deviation of the waiting times in Table 2.4 appears to indicate that not all waiting times are simple exponentials. This is particularly pronounced in Table 2.4(c) which refers to the error states.

Figure 2.6 shows the densities of waiting and holding times for one of the CPU load states, W_8 (see Appendix B for all states). Figure 2.6(a) shows the waiting time for W_8, and Figure 2.6(b) and 2.6(c) represent the holding times from state W_8 to DASD and SWE error states. These densities are fitted to phase-type exponential density functions [26].

$$f(t) = \sum_{i=1}^{n} a_i g_i(t) ,$$

where $a_i \geqslant 0$, $\sum_{i=1}^{n} a_i = 1$, and n is the number of phases. The $g_i(t)$ function can be a simple exponential, a multi-stage hyperexponential, or a multi-state hypoexponential density function. The definitions of these three types of exponential functions are listed below.

(1) Exponential: $g(t) = \lambda e^{-\lambda t}$.

(2) Hyperexponential: $g(t) = \sum_{i=1}^{r} \alpha_i \lambda_i e^{-\lambda_i t}$, where $\lambda_i > 0$, $\alpha_i \geqslant 0$, and $\sum_{i=1}^{r} \alpha_i = 1$.

(3) Hypoexponential: $g(t) = \sum_{i=1}^{r} a_i \lambda_i e^{-\lambda_i t}$, where $\lambda_i > 0$, $\lambda_i \neq \lambda_j$ if $i \neq j$, and

$$a_i = \prod_{\substack{i=1 \\ i \neq j}}^{r} \frac{\lambda_j}{\lambda_j - \lambda_i}.$$

Table 2.4 Mean waiting time (in seconds) of states

(a). CPU bound workload states

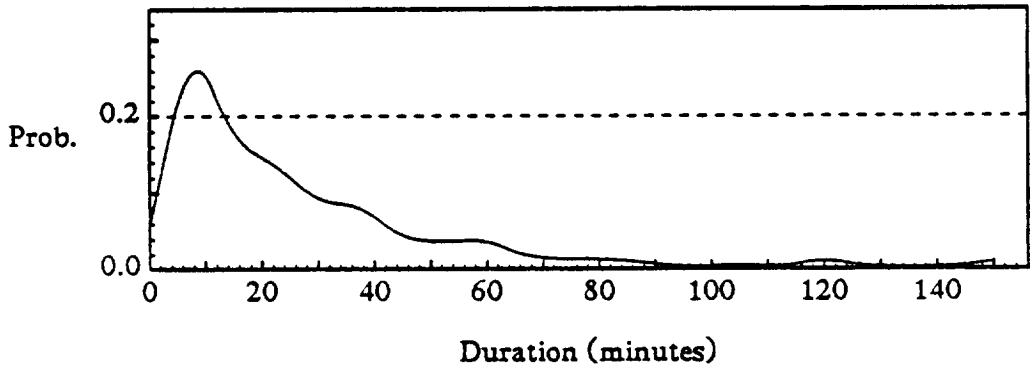
State	# of obs	Mean waiting time	Standard deviation	Std error of mean
W_1	53	1263.71	1384.20	190.13
W_2	2	289.65	1.19	0.84
W_3	20	698.79	913.30	204.22
W_4	130	1203.05	1130.28	99.13
W_5	11	613.74	421.73	127.16
W_6	147	1380.86	1588.76	131.04
W_7	268	1071.31	1004.46	61.36
W_8	266	1612.72	2576.35	157.97

(b). I/O workload states

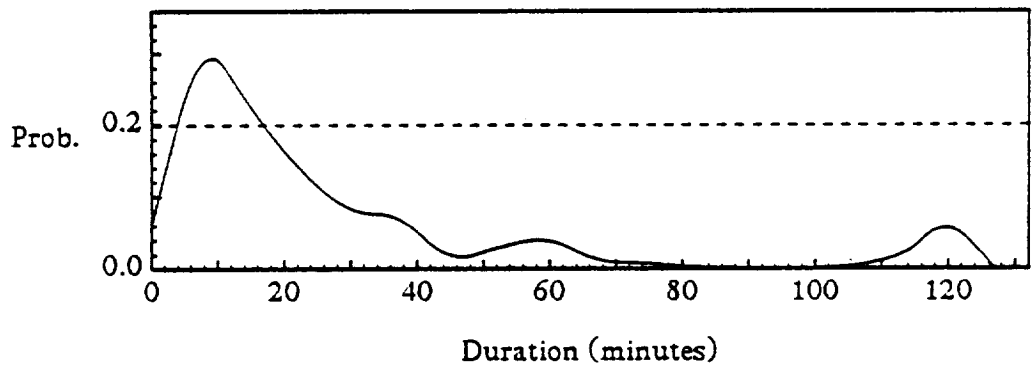
State	# of obs	Mean waiting time	Standard deviation	Std error of mean
U_1	45	1221.75	1475.70	219.98
U_2	316	1453.19	1530.75	86.11
U_3	12	1437.15	1452.86	419.40
U_4	18	1137.41	616.67	145.39
U_5	420	1243.63	1550.49	75.66
U_6	4	1696.85	1540.19	770.10
U_7	86	937.45	1127.57	121.59
U_8	9	387.74	176.96	58.99

(c). Error states

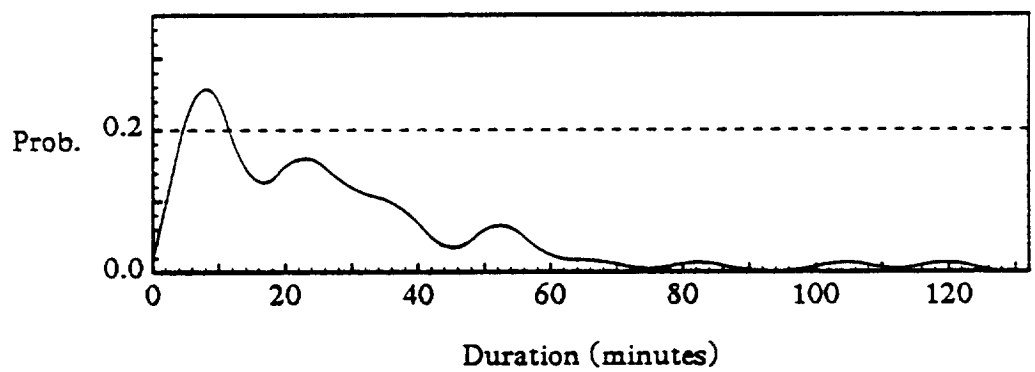
State	# of obs	Mean waiting time	Standard deviation	Std error of mean
CHAN	13	5.08	18.31	5.08
SWE	201	41.35	103.35	7.29
DASD	401	120.86	223.89	11.18
MULT	77	293.28	262.84	29.95



(a) Waiting time density of state W_8 (CPU = 0.96)



(b) Holding time density from state W_8 to state DASD



(c) Holding time density from state W_8 to state SWE

Figure 2.6. Waiting and holding time densities

Thus the graphs in figure 2.5 were fitted to the following functions (tested by using the Kolmogorov-Smirnov test [26] at the 0.01 significance level).

(1). waiting time : $f(t) = 0.000146e^{-0.002t} + 0.000939e^{-0.00103t} + 0.000033e^{-0.0002102t}$

(2). to a DASD error : $f(t) = 0.00094e^{-0.004t} + 0.0008355(e^{-0.000937t} - e^{-0.006595t})$

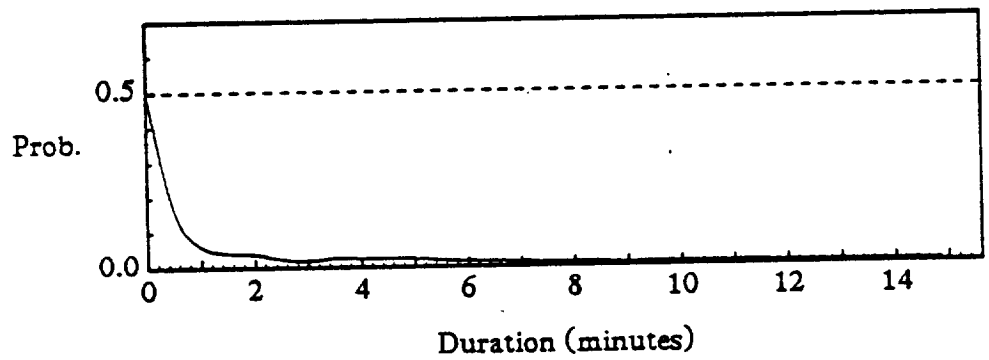
(3). to a SWE error : $f(t) = 0.00085e^{-0.005t} + 0.000701(e^{-0.000716t} - e^{-0.004688t})$

2.4.2. Recovery Distributions

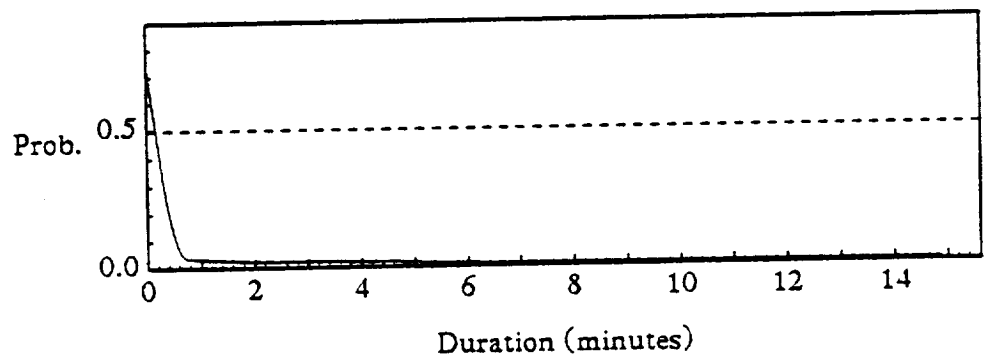
In our data, the selection of the destinations from any state of S_R was found to be independent of the holding time distribution. Further, for our system the time taken for each type of recovery can reasonably be considered constant. The overall recovery time, i.e., the duration of an error event (or the holding time in an error state), however was not constant since an error event may involve more than one recovery attempt. This time is computed as the time difference between the first detected error and the last detected error caused by the same event. The duration of an error event can be used to measure the effectiveness of recovery from this event and also the severity of error. Figure 2.7 shows examples of error duration densities for three different types of errors. Again, the following phase-type exponential densities were fitted to the graphs shown in Figure 2.7 (tested at the 0.01 significance level).

(1) DASD : $f(t) = 0.0375e^{-0.15t} + 0.007e^{-0.1t} + 0.008635e^{-0.014562t}$
$+ 0.0001861e^{-0.0021377t}$

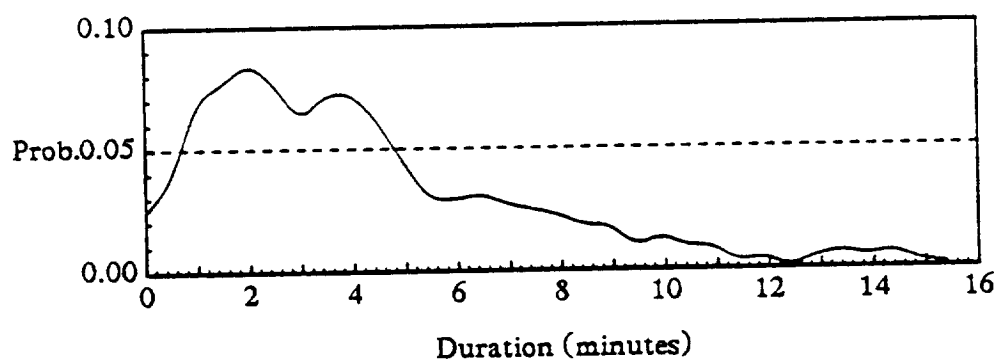
(2) SWE : $f(t) = 0.041181e^{-0.044518t} + 0.0002704e^{-0.0036075t}$



(a) DASD error duration density



(b) SWE error duration density



(c) MULT error duration density

Figure 2.7. Error duration densities

(3) MULT: $f(t) = 0.004371(e^{-0.003817t} - e^{-0.0301092t})$

2.5. Summary

In summary, we have developed a state-transition model which describes the normal and error behavior of the system. Some key characteristics of the model are:

(1) workload dynamics are explicitly described,

(2) error/recovery is explicitly described,

(3) waiting times in some workload and in most error states can not be modeled as simple exponentials, and

(4) the holding times from a given workload state to various error states are dependent on the destinations.

Thus, the resource-usage/error/recovery process is modeled as a complex irreducible semi-Markov process with the state OFFL as recurrent, making the overall model ergodic. Furthermore, the process is not an independent semi-Markov process since the waiting and holding time distributions are distinct for some states.

CHAPTER 3

MODEL ANALYSIS

Now that we have an overall model, we show the usage of this model to predict key system characteristics. The mean time between different types of errors is evaluated along with model characteristics such as the occupancy probabilities of key error and workload states. Since the normal state transitions are also available, we can explicitly examine those states which are crucial from a error viewpoint. In order to evaluate the model behavior, the model parameters, however, have to be defined and then the derivations of the measures can be carried out. Thus in the next section we provide the definitions of the model parameters and the derivations of some important measures.

3.1. Model Parameters

From Chapter 2 we know that the measured system is best modeled as a semi-Markov process. Assume that M is an n-state semi-Markov model and given a stochastic transition probability matrix P = $\left[p_{ij}\right]$, $p_{ij} \geqslant 0$, $i=1,2,\ldots,n$, $j=1,2,\ldots,n$, $\sum_{j=1}^{n} p_{ij}=1$, and a holding time density function matrix H (t) = $\left[h_{ij}(t)\right]$, $t \in (0,\infty)$, the mean holding time of the process staying in state i before making transition to state j, $\bar{\tau}_{ij}$, is

$$\bar{\tau}_{ij} = \int_{t=0}^{\infty} t\, h_{ij}(t)dt \ . \tag{3.1.1}$$

We mentioned that in Section 2.4.1 the waiting time for state i is the time that the process spends in state i before making a transition. Thus, a waiting time is merely a holding time that is unconditional on the destination state. Hence the mean waiting time $\bar{\tau}_i$ is related to the mean holding time $\bar{\tau}_{ij}$ by

$$\bar{\tau}_i = \sum_{j=1}^{n} p_{ij}\bar{\tau}_{ij} \ . \tag{3.1.2}$$

Suppose that a process has been operating unobserved a long time and given that the process is now making a transition, the probability that the transition is to state j, π_j, must satisfy n simultaneous equations

$$\pi_j = \sum_{i=1}^{n} \pi_i p_{ij} \ . \tag{3.1.3}$$

We note that these n equations are linearly dependent. This linear dependency can be easily shown by summing these n equations, which results in $1 = 1$. Therefore no unique solution for π_j can be obtained from just by solving the equations (3.1.3). Since we know that the probabilities that the transition to all states have to sum to one, i.e.,

$$\sum_{i=1}^{n} \pi_i = 1 \ . \tag{3.1.4}$$

Then we can use Equation 3.1.3 in conjunction with Equation 3.1.4 to provide an unique solution for the steady state transition probability. After we substitute Equation 3.1.4 into the left hand side of Equation 3.1.3, we have

$$1 = \pi_j p_{jj} + \sum_{\substack{i=1 \\ i \neq j}}^{n} \pi_i (1 + p_{ij}). \tag{3.1.5}$$

The unique solution for π_i can be obtained by solving n linear equations of (3.1.5). The matrix form of the solution is

$$\pi = \mathrm{O} \left[\mathrm{U} - \mathrm{I} + \mathrm{P}\right]^{-1}. \tag{3.1.6}$$

where π, O, U and I are:

(1) $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$,

(2) O is an unit row vector, i.e., all elements are one.

(3) U is an unit matrix, and

(4) I is an identity matrix.

After deriving the steady state probability (also called limiting state probability), the probability of a state being occupied by the process and the probability of the process entering a specified state can be obtained accordingly.

The steady state occupancy probability of state j, denoted as Φ_j, is the probability that the process occupies state j when the system reaches a stable stage, and is evaluated as [25]:

$$\Phi_j = \Phi_{ij} = \frac{\pi_j \bar{\tau}_j}{\bar{\tau}}. \tag{3.1.7}$$

where $\bar{\tau} = \sum_{i=1}^{n} \pi_i \bar{\tau}_i$.

We are sometimes interested not only in the probability that the process will occupy a state at some time in the future but also in the probability that the process will enter a

state at some particular future time. Thus, the probability that the process is just entering state j at some time instant after the system is in the steady state, e_j, is just the state occupancy probability Φ_j divided by its mean waiting time $\bar{\tau}_j$ [25]:

$$e_j = e_{i,j} = \frac{\Phi_j}{\bar{\tau}_j} . \tag{3.1.8}$$

After substituting the result of Equation 3.1.7 into Equation 3.1.8 we have

$$e_j = \frac{\pi_j}{\bar{\tau}} . \tag{3.1.9}$$

In a semi-Markov process — as in every life — an important question is "How long does it take to get from here to there?". Assume that the time it takes to reach state j for the first time if the system is in state i at time zero is $\Theta_{i,j}$, then $f_{i,j}(t)$, the probability that $\Theta_{i,j} = t$, is defined as [25]:

$$\begin{aligned} f_{i,j}(t) &= \text{Pro}\,(\Theta_{i,j} = t) \\ &= \sum_{\substack{r=1 \\ r \neq j}}^{n} p_{i,r} \int_0^t h_{i,r}(\sigma) f_{r,j}(t-\sigma) d\sigma + p_{i,j} h_{i,j}(t) . \end{aligned} \tag{3.1.10}$$

The process can make transitions to other states before it first reaches state j at time t, or it may stay in state i and then make a direct transition to state j at time t. The first term of the right hand side of Equation 3.1.10 computes the probability of being in state i for any $\sigma \epsilon [0, t)$ and the probability of the process being in another state r at the beginning time of $t-\sigma$ after the process is out of state i. The second term computes the probability if the process makes a transition directly from state i to state j at time t. Therefore, the time to move from state i to state j can be estimated as the mean first passage time for a process from state i to j, and it is evaluated as

$$\overline{\Theta}_{ij} = \int_0^\infty t f_{ij}(t)dt \ . \tag{3.1.11}$$

Since the $f_{ij}(t)$ is a recursive function which is shown in Equation 3.1.10, the computation time increases exponentially as t increases. However, in statistical the mean of a random variable can be estimated as the first moment of its moment generating function, e.g., the first derivative of its exponential transformation at point zero. The exponential transformation of a function $g(t)$, denoted as $g^e(s)$ is defined as

$$g^e(s) = \int_0^\infty g(t)e^{-st}\,dt \ . \tag{3.1.12}$$

So, the exponential transformation of $f_{ij}(t)$, is

$$\begin{aligned}
f^e_{ij}(s) &= \int_0^\infty f_{ij}(t)e^{-st}\,dt \\
&= \sum_{\substack{r=1 \\ r\neq j}}^{n} p_{ir} \int_0^\infty \int_0^t h_{ir}(\sigma) f_{rj}(t-\sigma)e^{-st}\,d\sigma dt + p_{ij}\int_0^\infty h_{ij}(t)e^{-st}\,dt \\
&= \sum_{r=1}^{n} p_{ir} \int_0^\infty \int_0^t h_{ir}(\sigma) f_{rj}(t-\sigma)e^{-st}\,d\sigma dt - p_{ij}\int_0^\infty \int_0^t h_{ij}(\sigma) f_{jj}(t-\sigma)e^{-st}\,d\sigma dt \\
&\quad + p_{ij}\int_0^\infty h_{ij}(t)e^{-st}\,dt \\
&= \sum_{r=1}^{n} p_{ir} h^e_{ir}(s) f^e_{rj}(s) + p_{ij} h^e_{ij}(s)[1 - f^e_{jj}(s)]
\end{aligned} \tag{3.1.13}$$

Therefore, the mean first passage time for a process from state i to j is

$$\bar{\Theta}_{ij} = \frac{d}{ds} f^e_{ij}(s)\big|_{s=0}$$

$$= \sum_{r=1}^{n} p_{ir}\left[\frac{d}{ds} h^e_{ir}(s) f^e_{rj}(s) + h^e_{ir}(s)\frac{d}{ds} f^e_{rj}(s)\right]\Big|_{s=0} + p_{ij}\frac{d}{ds} h^e_{ij}(s)\big|_{s=0}$$

$$- p_{ij}\left[\frac{d}{ds} h^e_{ij}(s) f^e_{jj}(s) + h^e_{ij}(s)\frac{d}{ds} f^e_{jj}(s)\right]\Big|_{s=0}$$

Since $\frac{d}{ds} h^e_{ij}(s)\big|_{s=0}$ is the first moment of holding time distribution, i.e., the mean holding time $\bar{\tau}_{ij}$, and

$$f^e_{ij}(s)\big|_{s=0} = \int_0^\infty f_{ij}(t)dt = 1 ,$$

the mean first passage time $\bar{\Theta}_{ij}$ can be derived as below.

$$\bar{\Theta}_{ij} = \sum_{r=1}^{n} p_{ir}(\bar{\tau}_{ir} + \bar{\Theta}_{rj}) - p_{ij}\bar{\Theta}_{jj} \tag{3.1.14}$$

By Equation 3.1.2, Equation 3.1.14 can be written as :

$$\bar{\Theta}_{ij} = \bar{\tau}_i + \sum_{r=1}^{n} p_{ir}\bar{\Theta}_{rj} - p_{ij}\bar{\Theta}_{jj} . \tag{3.1.15}$$

However, the mean recurrence time $\bar{\Theta}_{jj}$ is actually the reciprocal of the steady state entrance rate into state j, i.e.

$$\bar{\Theta}_{jj} = \frac{1}{e_j} . \tag{3.1.16}$$

Thus Equation 3.1.15 can be written as

$$\bar{\Theta}_{ij} = \left[\bar{\tau}_i - \frac{p_{ij}}{e_j}\right] + \sum_{r=1}^{n} p_{ir}\bar{\Theta}_{rj} \ . \tag{3.1.17}$$

and its matrix form is

$$\Theta = K + P\Theta \ . \tag{3.1.18}$$

where $\Theta = \left[\bar{\Theta}_{ij}\right]$ and $K = (TU) - E^{-1}$ where T, U and E are :

(1) T is a diagonal matrix in which $T_{i,i} = \bar{\tau}_i$, and $T_{ij} = 0$ otherwise.

(2) U is a unit matrix, i.e., all elements are one, and

(3) E is a diagonal matrix in which $E_{i,i} = e_i$, and $E_{ij} = 0$ if $i \neq j$.

Therefore, the mean first passage time matrix Θ can be derived as:

$$\Theta = \frac{K}{I - P} \tag{3.1.19}$$

3.2. Model Behavior

In this section we use the measures that were defined previously to predict the key system characteristics for given stochastic transition probability matrix and holding time density function matrix which were estimated from the collected data. By solving the semi-Markov model we discover that the system makes a transition every 9 minutes and 8 seconds, on average. In comparing this with the mean time between error (MTBE) listed in Table 3.1, it is clear that most often the transitions are from one workload state to another. Also note that the model indicates an MTBE of 4152 hours for CPU errors. This number is estimated by solving the model equations although there were no observations in the measured period. (In examining the error data over a one year period we found two CPU

Table 3.1. Mean time between errors

Type of error	Frequency count	%	Mean time between errors (hour)
CPU	0	0	4152
CHAN	13	1.88	26.88
SWE	201	29.05	1.75
DASD	401	57.95	0.87
MULT	77	11.12	4.62

errors.) The table also shows that a disk error occurs (as indicated in the model) almost every 52 minutes while a software error is detected every 1 hour and 45 minutes. Most of the disk errors (95%) are recovered through hardware recovery (i.e., hardware instruction retry or ECC correction), thus resulting in negligible overhead. This shows that on-line recovery is highly effective and provides a system with the ability to tolerate a fault and recover almost instantaneously. Thus, a highly reliable system is achieved.

Table 3.2 lists the mean recurrence time for recovery routines. It shows that the on-line hardware recovery routine is invoked once every 0.62 hours, while a software recovery occurs every 2.57 hours. As mentioned earlier, hardware recovery involves hardware instruction retry or ECC correction. The maximum number of retries is predetermined. In the measured system each CPU has a 26-nanosecond machine cycle time and the disk seek time is about 25 milliseconds. We estimate a worst case hardware recovery cost of 0.5 seconds, i.e., incorporating twenty I/O retries: ten through the original I/O path and another ten through an alternative I/O path if the alternative is available. This, of course, overestimates the cost of hardware retry used for the CPU errors. However, the impact is very

Table 3.2. Mean recurrence time

Type of recovery	Mean recurrence time (hour)
Hardware	0.62
Software	2.57
Alternative	-
Off-line	651.37

insignificant. This can be seen by comparing the estimated time for each hardware recovery with the recovery overhead. The comparison shows that the cost of hardware recovery is worth only 0.02% of total computation time. The mean recurrence time of the alternative recovery routine, is not estimated due to lack of data, i.e., this event seldom occurred.

The model characteristics are summarized in Table 3.3. A dashed line in this table indicates a negligible value (less than 0.00001 probability). Table 3.3(a) shows the normal system behavior. Given that a transition has occurred the system CPU load is most likely to reach to W_7 or W_8, i.e. 0.72 or above. This is also reflected in the entry and occupancy probabilities (e and Φ). From the occupancy probabilities we see that almost 34% of the time the CPU load is as high as 0.96 (W_8); 39% of the time the CPU is moderately loaded (W_6 + W_7).

Table 3.3(b) shows the erroneous system behavior. The table indicates that about 30% of the transitions are to an error state (obtained by summing all the π's for all the error states). The DASD errors have the highest transition and entry probabilities. Since a transition occurs every 9 minutes, we estimate that an error is detected, on the average, every 30 minutes. Of course, over 98% of these errors caused negligible overhead.

Table 3.3. Summary of model characteristics

(a). CPU bound workload states

Measure	Workload state								
	W_0	W_1	W_2	W_3	W_4	W_5	W_6	W_7	W_8
Φ	0	0.0625	0.0008	0.0136	0.1258	0.0054	0.1639	0.2255	0.3398
π	0.0257	0.0264	0.0014	0.0104	0.0559	0.0047	0.0635	0.1125	0.1127
e	0.00005	0.00005	-	0.00002	0.0001	0.00001	0.00012	0.00021	0.00021
$\overline{\Theta}$	5.78	5.62	102.56	14.32	2.65	31.38	2.33	1.32	1.32

(b). Error and recovery states

Measure	Error state					Recovery state			
	CPU	CHAN	SWE	DASD	MULT	HWR	SWR	ALT	OFFL
Φ	0	0.00005	0.0066	0.0383	0.0179	0.00022	0.00011	-	-
π	0.00004	0.0055	0.0850	0.1692	0.0322	0.2379	0.0572	0.00004	0.00023
e	-	0.00001	0.00016	0.00032	0.00006	0.00045	0.00011	-	-
$\overline{\Theta}$	4152	26.88	1.75	0.37	4.62	0.62	2.57	4089.5	651.57

An interesting characteristic of the multiple error events is also seen in Table 3.3(b). Although, the transition probability (π) of a MULT error is lower than that for SWE (0.0322 vs. 0.0850), its occupancy probability (Φ) is higher (0.0179 vs. 0.0066). This is due to the fact that a MULT error has a longer sojourn time as compared to SWE error events (293 seconds vs. 41 seconds from Table 2.4).

3.3. Effect of Workload

In this section we compute the steady state probability of being in a specified workload state and making a transition to a specified error state. Table 3.4 shows the

probabilities of a error occurring at various load levels. In this table, "time" refers to the mean holding time in the specified workload state (e.g., CPU = 0.96) before the process making a transition to the selected state (e.g., CHAN). An important relation between error probability and holding time in a workload state is seen in this table. The error probabilities appear to be not only a function of resource usage [10], but also related to the length of the holding time in a resource usage state. For example, in Table 3.4(a) the probability of a

Table 3.4. Holding time and transition probabilities to error states

(a). CPU workload

CPU	Error state								Total
	CHAN		SWE		DASD		MULT		
Load	Time	Prob	Time	Prob	Time	Prob	Time	Prob	Prob
0.96	668.18	0.0011	1609.71	0.0786	1218.62	0.1296	1641.20	0.0285	0.2377
0.72	596.28	0.0032	1118.12	0.0492	971.62	0.0990	757.09	0.0146	0.1661
0.54	1304.96	0.0010	1507.92	0.0471	1070.10	0.0489	722.26	0.0052	0.1027

Time - in seconds.

(b). I/O workload

DASD	Error state								Total
	CHAN		SWE		DASD		MULT		
Load	Time	Prob	Time	Prob	Time	Prob	Time	Prob	Prob
96.20	0	0	256.23	0.0022	434.54	0.0162	578.97	0.0046	0.023
67.34	898.35	0.0049	1243.35	0.0987	1170.84	0.1840	928.12	0.0262	0.3138
41.59	4522.67	0.0214	1516.92	0.0875	1148.18	0.1198	1286.67	0.0234	0.2521

Time - in seconds.

channel error is almost the same for two different CPU loads, 0.96 and 0.54. The mean holding time before a channel error occurs at the lower load is larger than that for the higher load, i.e. 1304.96 seconds versus 668.18 seconds. When the holding times are similar, however, (or increasing with increased usage), the error probabilities do increase with increasing resource usage. A similar phenomenon also exists for the I/O workload (see Table 3.4(b)). Thus, not only does a higher workload result in a higher error probability (for similar holding times), but the error probability also increases with increased holding time in a particular state. In other words, the error probability appears to be a function of the absolute amount of resource consumed in a given state, be it through increased workload and/or increased holding times. An explanation for this apparent "wear out" phenomenon is not clear (since a large majority of our errors are transient), but it certainly calls into further question the validity of the frequently used constant error probability assumption often made in reliability modeling.

3.4. Model Validation

In Chapter 2 we had shown that the resource-usage/error/recovery process of the measured system is best modeled as a semi-Markov process. This is due to the fact that the waiting and holding time distributions of some states are not exponentials. In order to validate this semi-Markov assumption we will compare the occupancy probabilities of states predicted from the model with the values estimated from the collected data.

From Equation 3.1.7 we know that the state occupancy probability of the model, Φ_j, is defined as :

$$\Phi_j = \frac{\pi_j \bar{\tau}_j}{\sum_{i=1}^{n} \pi_i \bar{\tau}_i} .$$

However the actual occupancy probability, denoted as $\hat{\Phi}_i$, can be estimated from the collected data by using the following equation.

$$\hat{\Phi}_i = \frac{\text{total time that system observed to be in state } i}{\text{length of the observation period}} . \tag{3.4.1}$$

Table 3.5 lists the comparison of these two measures, Φ and $\hat{\Phi}$, for the normal states with significant occupancy probability (greater than 0.1 probability) and for one key error state (DASD). From Table 3.5 we see that the predicted probabilities closely match those estimated from the collected data (with the maximum of 0.025 tolerance, i.e., $\frac{\epsilon}{\hat{\Phi}}$). This

Table 3.5. Comparison of occupancy probabilities for different states

State	W_4	W_6	W_7	W_8	DASD
Φ	0.1258	0.1639	0.2255	0.3398	0.0383
$\hat{\Phi}$	0.1259	0.1634	0.2311	0.3452	0.0386
ϵ	0.0001	0.0005	0.0056	0.0054	0.0003
$\frac{\epsilon}{\hat{\Phi}}$	0.0008	0.0031	0.0242	0.0156	0.0078

Φ : predicted occupancy probability
$\hat{\Phi}$: actual occupancy probability
ϵ : the absolute error, $|\Phi - \hat{\Phi}|$

indicates that the semi-Markov model is a good model for the resource-usage/error/recovery process of the measured system.

3.5. Markov Versus Semi-Markov

In this section we investigate the significance of using a semi-Markov model to describe the overall resource-usage/error/recovery process. It has been argued that since errors only occur infrequently (i.e., λ is small), a Markov model may well approximate the real behavior. Thus, although the collected data shows that the semi-Markov process is a better model for the resource-usage/error/recovery process, i.e., more closely approximates the data from the measured system, it is reasonable to ask what deviations may occur if a Markov process is assumed. In order to answer this question we use a Markov model to describe the resource-usage/error/recovery process of the measured system and compare the results with those obtained through the more realistic semi-Markov model.

Two measures, the unconditional transition probability to the next state (γ) and the first passage time (Θ), are used as the basis for comparison.

3.5.1. Unconditional transition probability (γ)

Given a stochastic transition probability matrix $\left[p_{ij}\right]$ and a holding time density function matrix $\left[h_{ij}(t)\right]$, the unconditional transition probability from state i to state j, denoted as γ_{ij}, in the semi-Markov process is given by [25]:

$$\gamma_{ij} = \frac{\pi_i p_{ij} \bar{\tau}_{ij}}{\bar{\tau}}. \tag{3.5.1}$$

where $\bar{\tau}_{ij}$ is the mean holding time before a transition occurs from state i to state j and $\bar{\tau}$ is the mean holding time of the process. Because the selection of the next state in the Markov process is not dependent on the holding time in the current state, i.e. $\bar{\tau}_{ij} = \bar{\tau}_{ik}$ for every j and k, so from Equation 3.1.2 we can have

$$\bar{\tau}_i = \sum_{j=1}^{n} p_{ij}\bar{\tau}_{ij} = \bar{\tau}_{ij} \sum_{j=1}^{n} p_{ij} = \bar{\tau}_{ij} \ . \tag{3.5.2}$$

Substitute this into Equation 3.5.1 we have

$$\gamma_{ij} = \frac{\pi_i p_{ij} \bar{\tau}_i}{\bar{\tau}}. \tag{3.5.3}$$

Further, from Equation 3.1.7 we have

$$\gamma_{ij} = \Phi_i p_{ij}. \tag{3.5.4}$$

Table 3.6 compares the unconditional transition probability for semi-Markov and Markov models. We see from Table 3.6(a) that when the CPU load is as high as 0.96, the transition probabilities to the software and multiple errors are close for both models. This is also true for channel error when the CPU load is 0.54. This is because for some destinations j the holding time to the next state is the same as the waiting time of the current state, i.e., $\bar{\tau}_i = \bar{\tau}_{ij}$. For the majority of the cases, however, the Markov and semi-Markov models are not in agreement. Table 3.7 shows the ratios of the unconditional transition probability γ_{ij} estimated from both models, Markov versus semi-Markov. If the ratio is less then 1 then the Markov process underestimates the transition probability, otherwise, it overestimates. From this table we see that the Markov assumption sometimes overestimates and sometimes underestimates the transition probability. In particular it overesti-

Table 3.6. Comparison of transition probabilities, γ_{ij}
(Markov versus Semi-Markov)

(a). From CPU workload to error states

Load	Model	Error			
		CHAN	SWE	DASD	MULT
0.96	semi-Markov	0.0011	0.0786	0.1296	0.0285
	Markov	0.0025	0.0783	0.1705	0.0278
0.72	semi-Markov	0.0032	0.0492	0.0990	0.0146
	Markov	0.0058	0.0469	0.1086	0.0206
0.54	semi-Markov	0.0010	0.0471	0.0489	0.0052
	Markov	0.0011	0.0429	0.0627	0.0099

(b). From I/O workload to erro states

Load	Model	Error			
		CHAN	SWE	DASD	MULT
96.20	semi-Markov	0	0.0022	0.0162	0.0046
	Markov	0	0.0082	0.0350	0.0075
67.34	semi-Markov	0.1840	0.0987	0.0263	0.0049
	Markov	0.0068	0.0987	0.1955	0.0352
41.59	semi-Markov	0.0214	0.1198	0.0875	0.0234
	Markov	0.0069	0.0839	0.1516	0.0264

Table 3.7. Ratio of γ_{ij} (Markov/semi-Markov)

Workload		Error			
Resource	Load	CHAN	SWE	DASD	MULT
CPU	0.96	2.273	0.996	1.316	0.976
	0.72	1.818	0.953	1.097	1.411
	0.54	1.100	0.911	1.282	1.904
DASD	96.20	-	3.727	2.161	1.631
	67.34	0.037	1.0	7.446	7.184
	41.59	0.322	0.700	1.733	1.128

mates the transition probabilities to most error states, regardless of the state. Overestimation will lead to an unduly conservative reliability estimate and underestimation to an overly optimistic estimate. Thus both are undesirable.

3.5.2. First Passage Time (Θ)

We now examine the difference between the first passage times under the Markov and the semi Markov assumptions. The first passage time distribution can be used to estimate the MTBE and its variance.

The mean first passage time from state i to state j, $\overline{\Theta}_{ij}$ in a semi-Markov process is given in Section 3.1 as :

$$\overline{\Theta}_{ij} = \begin{cases} \dfrac{1}{e_j} & \text{if} \quad i = j \\ \overline{\tau}_i + \sum\limits_{\substack{r=1 \\ r \neq j}}^{n} p_{ir} \overline{\Theta}_{rj} & \text{otherwise} \end{cases} \tag{3.5.5}$$

From this equation, we notice that the mean first passage time depends on only the mean holding time and the conditional transition probability of the current state. Clearly, if the first moment of the first passage time to the error state (e.g., the MTBE) is the only main concern, the Markov process should be able to provide adequate information. If the distribution (or the higher moments) of the first passage time is of interest, the Markov model may be inadequate, particularly if the variance of the first passage time is large. This can be seen clearly from the following equation [25].

$$\overline{\Theta^2}_{ij} = \begin{cases} \dfrac{1}{\pi_j}\left(\sum_{i=1}^{n} \pi_i \overline{\tau^2}_i + \sum_{\substack{r=1 \\ r \neq j}}^{n} \sum_{i=1}^{n} 2\pi_i p_{ir} \overline{\tau}_{ir} \overline{\Theta}_{rj}\right) & \text{if} \quad i = j \\ \overline{\tau^2}_i + \sum_{\substack{r=1 \\ r \neq j}}^{n} p_{ir}\left(2\overline{\tau}_{ir}\overline{\Theta}_{rj} + \overline{\Theta^2}_{rj}\right) & \text{otherwise} \end{cases} \tag{3.5.6}$$

This equation indicates that the second moment of the first passage time is a function of the second moment of the state waiting time, as well as the mean holding time to the next state. Since the mean ($\overline{X}$) and the standard deviation (σ) of an exponential distribution are the same, and the second moment of an exponential distribution is only a function of its mean, i.e. $E[X^2] = 2*E[X]^2$. Thus, a Markov assumption may under- or over-estimate the second moment, $E[X^2]$, if $\sigma \neq E[X]$.

Table 3.8 shows the ratio of $\overline{\Theta^2}_{ij}$ (Markov/semi-Markov) for transitions from a few selected workload states to the error states. From Table 3.8, we see that the Markov assumption frequently underestimates the second moment of the first passage time (to the error state). The underestimation can be as much as 30%. However, it overestimates the variation of first passage time among different resource usage states, although this is not

Table 3.8. Ratio of $\overline{\Theta^2}$ (Markov/semi-Markov)

Workload		Error			
Resource	Load	CHAN	SWE	DASD	MULT
CPU	0.96	0.989	0.876	0.694	0.939
	0.72	0.996	0.998	0.909	0.972
	0.54	0.992	0.937	0.830	0.953
DASD	96.20	1.006	1.010	0.901	0.991
	67.34	1.002	0.963	0.888	0.972
	41.59	1.001	0.948	0.871	0.963

shown in the table.

3.5.3. Summary

In summary, our measurements show that using a Markov model frequently overestimates the unconditional transition probabilities and underestimates the variance of the first passage times to the error states. The overestimation, of course, will lead to an unduly conservative reliability prediction. It can be argued that such gross overestimation (as seen in some cases here) is undesirable and may not be cost beneficial. The underestimation is no doubt a serious problem which may lead to unduly optimistic reliability prediction.

CHAPTER 4

SOFTWARE RELIABILITY MODEL

4.1. Introduction

The problem of modeling software reliability during the development, debugging and validation phases of the software cycle is a well researched area. However, there are few studies which model software error and recovery processes in a fully operational production environment. The difficulties are partly due to the fact that, unlike computer hardware, which is reasonably modularized, each software system can have its own peculiar characteristics. At this stage, it is extremely valuable to develop a comprehensive model quantifying the software error and recovery processes in a production system using real data. In addition to providing useful information on how and when errors occur in the real world, this process provides the quantification of the interaction among different types of software errors; an important result for developing analytical models.

In this chapter, a state-transition model to describe the software error and recovery processes in a complex operating system is described. Measurements were made on an MVS (Multiple Virtual Storage) system running on an IBM 3081 mainframe. Time-stamped low level error and recovery data from MVS, collected during the normal operation of the system, formed the basis for developing the model. The semi-Markov model developed from the real data provides a quantification of the system error characteristics and also gives an

insight into the interaction between the various software error and recovery processes occurring during normal system operation.

4.1.1. Related Research

Most software reliability models usually refer to the development, debugging and testing phases of the software [11, 12, 13, 14]. Few of these models have been applied to the operational phase of the software. In [2] and [15], software failures in an operating environment are studied. Both studies found that at least 60% of system failures were software related. There has been little explicit study of hardware/software reliability. In [15], software failures related to hardware problems in the operating system are analyzed and it is shown that errors in the hardware/software interface are often fatal. In [27], a resource-usage/reliability model was developed from real data and it was seen that about 36% of detected errors (not necessarily system failures) were related to software problems.

With the exception of software reliability growth models, which have been validated with real data, there are few, if any, models of software reliability in an operational environment. Exceptions are the hardware and software model discussed in [18] and a measurement-based model of workload dependent failures discussed in [10]. However, these only describe the external behavior of the system and do not provide insight into component-level behavior.

It is therefore highly instructive to develop a detailed model based on low-level error data from a production system. In the following sections, we construct a error/recovery model for the MVS operating system. Software problems of differing severity are modeled. Multiple errors are also considered and and the effect of on-line recovery routines is taken into account.

4.2. Error Characterization

In this section the collection and characterization of the software error and error recovery data are discussed. A state-transition diagram is developed to describe the different error and recovery states. This allows us to determine the serverity of errors and effectiveness of recovery.

Error data based on different causes were collected. Information on software errors is automatically logged by an operating system module. Details of the logging mechanism are described in [23]. In order to ensure that the analysis is not biased by error records relating to the same problem, two levels of data reduction which were described in Chapter 3 were performed. As a result, the software errors were classified into eight classes. These eight classes are called error events, since they may contain more than one error, and are defined as follows.

(1) Control (CTRL)	- incidents indicating the invalid use of control statements and invalid supervisor calls
(2) Deadlocks (DLCK)	- incidents indicating system or operator detected endless loop, endless wait state or violation of system or user-defined time limits
(3) I/O and Data Management (I/O)	- incidents indicating problems occurred during I/O management or during the creation and processing of data sets
(4) Storage Management (SM)	- incidents indicating errors in the storage allocation/de-allocation process or in virtual memory mapping
(5) Storage Exceptions (SE)	- incidents indicating addressing of nonexistent or inaccessible memory locations
(6) Programming Exceptions (PE)	- incidents indicating program errors other than storage exceptions

(7) Others (OTHR) - incidents indicating that problems occurred which do not fit the above categories

(8) Multiple Errors (MULT) - incidents indicating more than one type of error listed above

Table 4.1 lists the frequencies of different types of software error events defined above. The table shows that more than one half (52.5%) of software errors were I/O and data management errors and another 11.4% of the errors were storage management errors. A significant percentage (17.4%) of errors were classified as multiple errors and are specifically modeled in the following sub-section.

4.2.1. Multiple Errors

A multiple error most often is due to a single fault that has non-identical manifestations provoked by different types of system activity. Since the manifestations are not

Table 4.1. Frequency of software errors

Type of Errors	Frequency	Percent
Control	213	7.72
Deadlock	23	0.84
I/O & Data Management	1448	52.50
Program Exception	65	2.43
Storage Exception	149	5.40
Storage Management	313	11.35
Others	66	2.32
Multiple Error	481	17.44
Total	2758	100.00

identical, recovery may be complex. Figure 4.1 shows the state-transition diagram of a multiple error developed from the data. The transition probability from state i to state j, p_{ij}, is estimated from the measured data using:

$$p_{ij} = \frac{\text{observed number of transitions from state } i \text{ to state } j}{\text{observed number of transitions from state } i}.$$

This figure not only illustrates the possible interactions among different software errors but also provides detailed information on the occurrence of transitions. For example, if a program exception error (PE) occurs, there is about a 63% chance that a storage exception (SE) on error will follow. Further, there is more than a 50% chance that one storage error will be followed by another error of the same type (52% for storage management and also for storage exception). If we only focus on those transitions with significant probabilities (i.e., higher than 0.1), the number of states in Figure 4.1 can be reduced to five. The state-transition diagram for these active states is illustrated in Figure 4.2. Notice that a cyclic path is formed by the I/O and data management (I/O) along with the two different types of exception states (program exception and storage exception).

4.2.2. Recovery Modeling

Recovery in MVS is designed as a means by which the system can prevent a total loss. Whenever a program is abnormally interrupted due to the detection of an error, the Supervisor gets control. If the problem is such that further processing could degrade the system or destroy data, the Supervisor gives control to the Recovery Termination Manager (RTM). If a recovery routine is available for the problem program, RTM gives control to this routine before deciding to terminate the program.

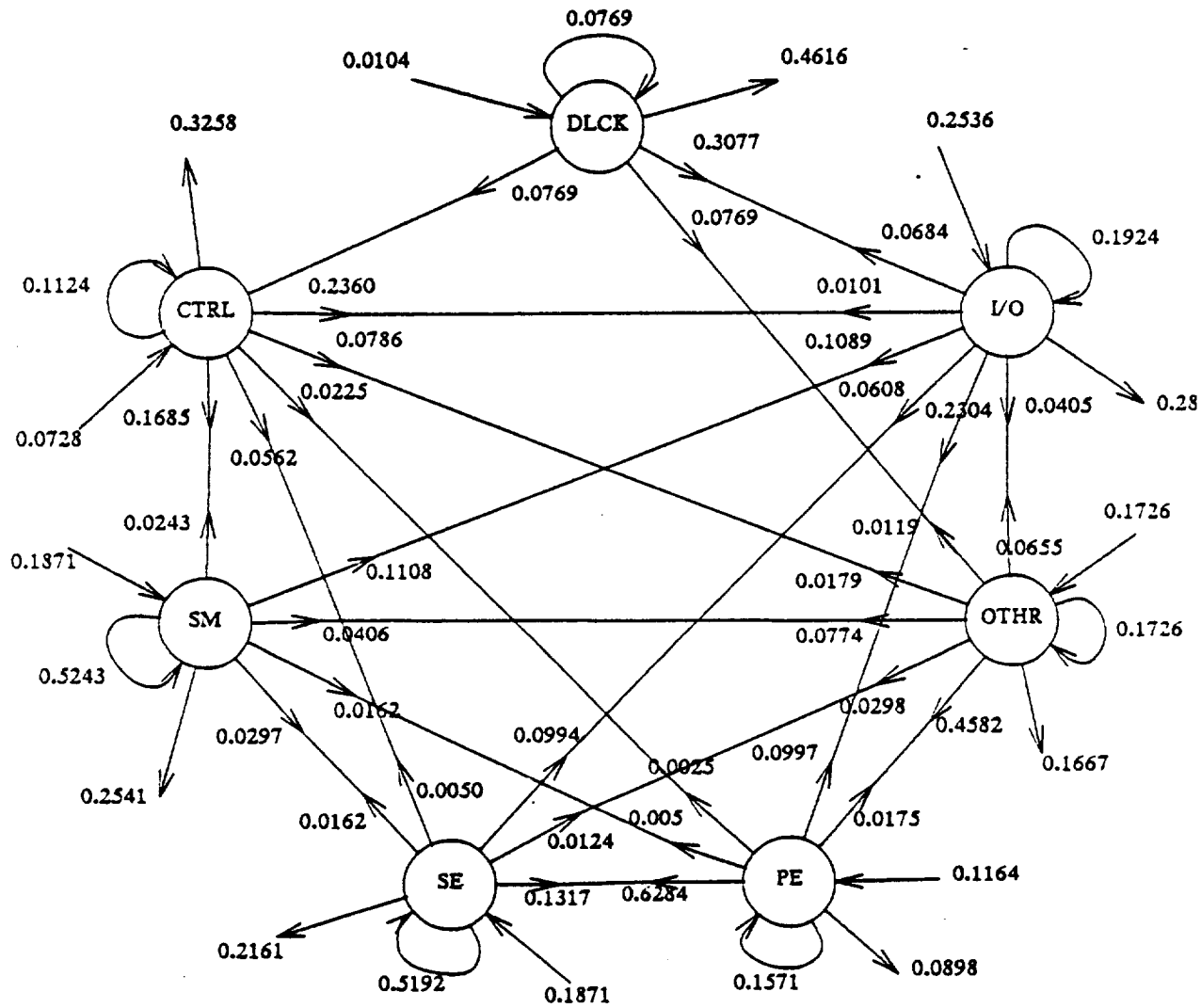


Figure 4.1. State-transition diagram for a multiple error

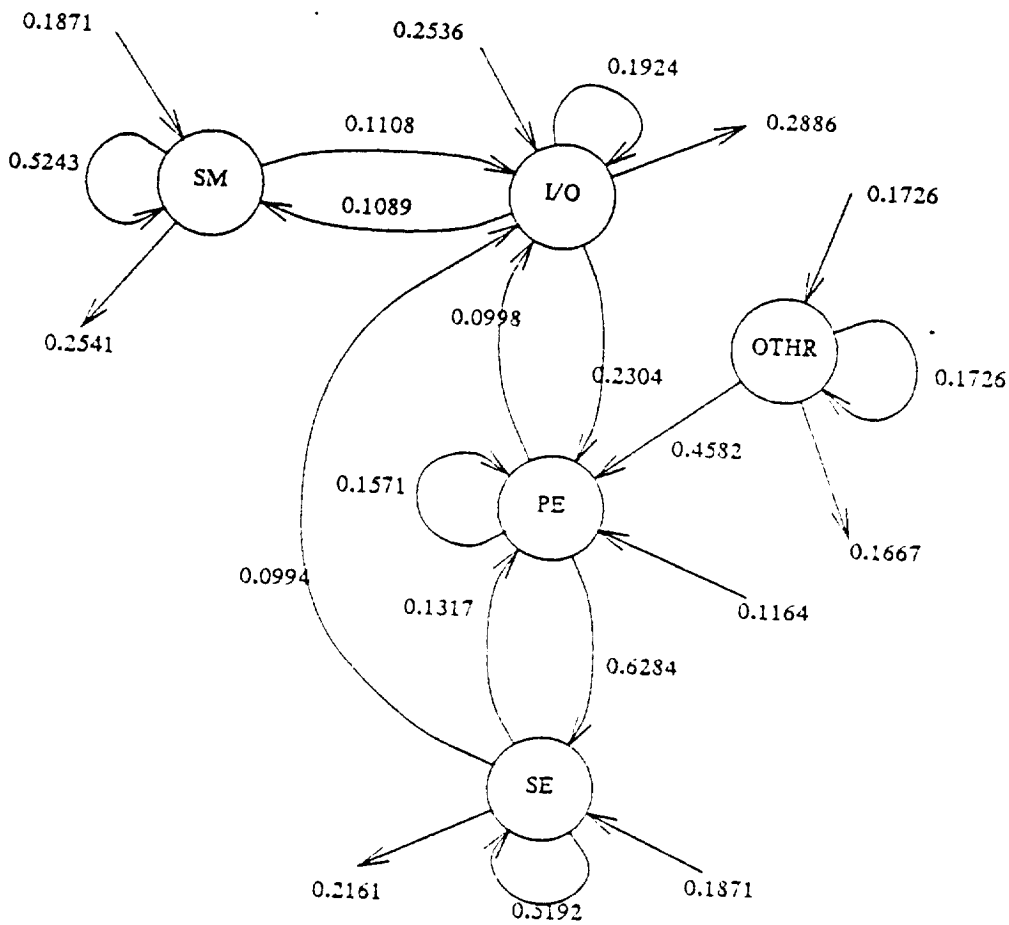


Figure 4.2. Reduced state-transition diagram of multiple errors

The purpose of a recovery routine is to free the resources kept by the failing program (if any), to locate the error, and to request either a continuation of the termination process or a retry. Recovery routines are generally provided to cover critical MVS functions. It is

however, the responsibility of the installation (or of the user) to write a recovery routine for other programs.

More than one recovery routine can be specified for the same program: if the latest recovery routine asks for a termination of the program, the RTM can give control to another recovery (if provided). This process is called "percolation." The percolation process ends if either a routine issues a valid retry request or no more routines are available. In the latter case, the program and its related subtasks are terminated. If a valid retry is requested, a retry routine restores a valid status using the information supplied by the recovery routine(s) and gives control to the program. In order for a retry to be valid, the system should verify that there is no risk of error-recurrence and that the retry address is properly specified. An error may have four possible effects.

1) Retry	- The system successfully recovered and returned control to the problem program.
2) Task Termination	- The program and its related subtasks are terminated, but the system is not affected.
3) Job Termination	- The job in control at the time of the error is aborted.
4) System Damage	- The job or task in control at the time of the error was critical for system continuation. Thus, job/task termination resulted in system failure.

Figure 4.3 illustrates the steps in the recovery process. It is clear that recovery can be as simple as a retry or more complex, requiring several percolations before a retry. The problem can also be such that no retry or percolation is possible. Table 4.2 shows the percentage for these different types of situations. For example, for storage management errors, approximately 8% of the cases resulted in a direct retry, 84% involved some percolation and over

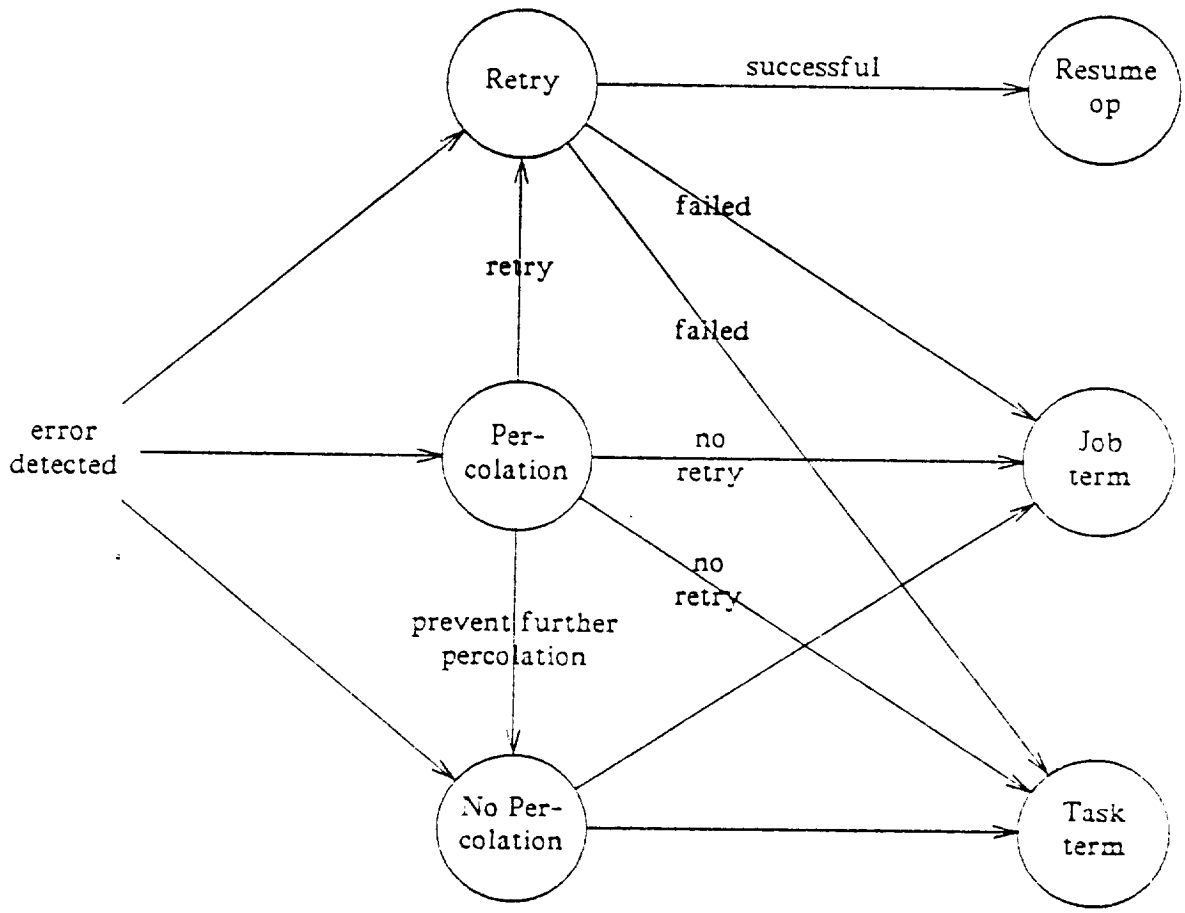


Figure 4.3. Flow of recovery

Table 4.2. Percentages of recovery attempts for a software error

Type of error	Retry (%)	Percolation (%)	No-Percolation (%)
Control	78.38	21.62	0.0
Deadlock	2.78	97.22	0.0
I/O & Data Management	93.49	6.51	0.0
Program Exception	20.09	79.91	0.0
Storage Exception	28.09	71.91	0.0
Storage Management	7.77	83.73	8.50
Others	14.89	85.11	0.0

8% could not be percolated any further (i.e. jobs/task termination). The table shows that only in a small percentage of the cases was the problem un-recoverable (no-percolation).

4.3. Software Reliability Model

4.3.1. Overall Error/Recovery Model

In this section we combine the separate error and recovery models to construct a single overall model shown in Figure 4.4. Note that a state, Normal, represents the normal system operation. The results of the recovery process are classified into three different states (resume op, task term and job term) to reflect the severity of errors. The model thus provides a complete overview of software error and recovery from an error occurrence to its recovery.

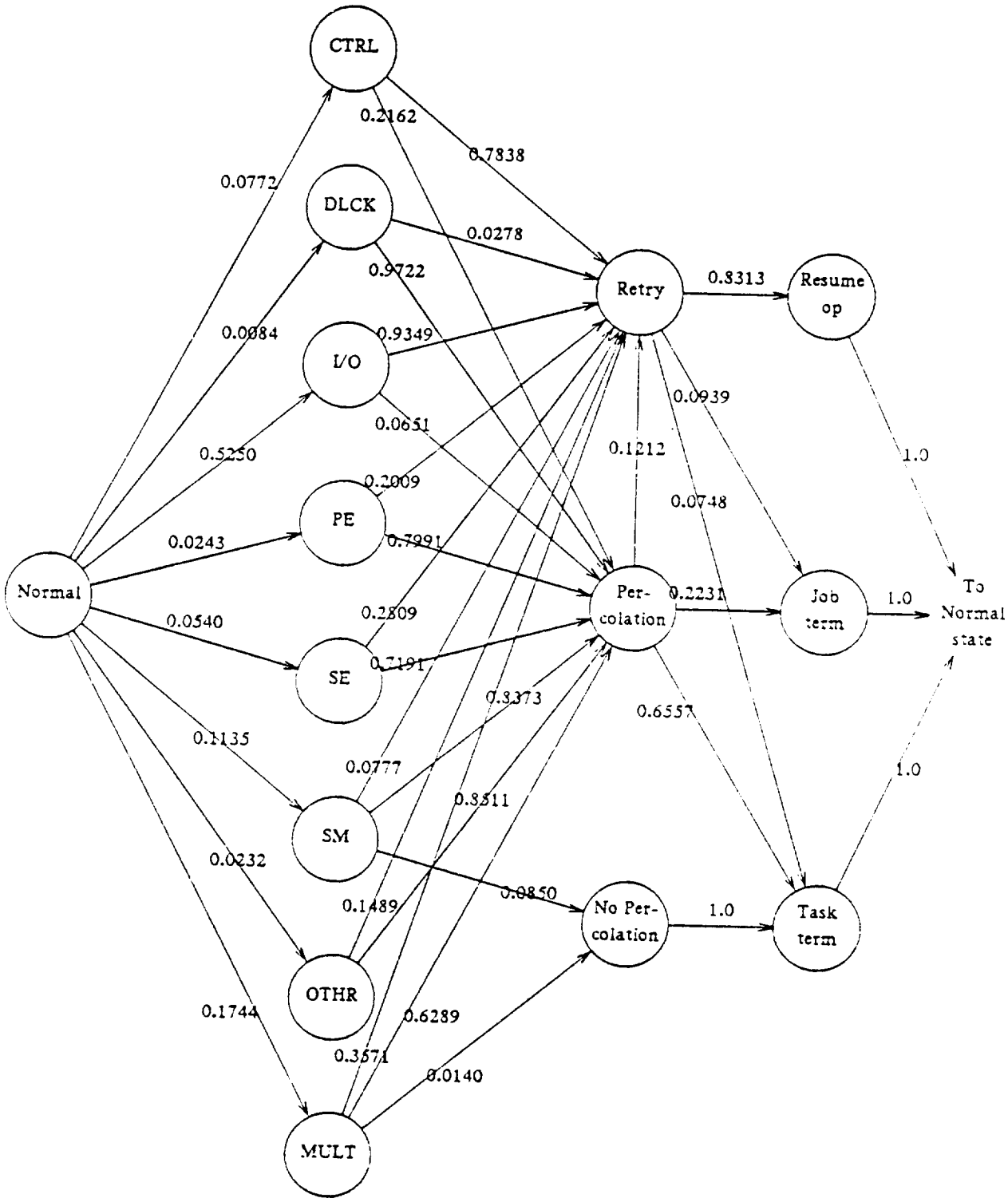


Figure 4.4. Software error/recovery model

4.3.2. Waiting Time Distributions

Table 4.3 shows the characteristics of both normal and error states in terms of their waiting times. Note that the duration of a single error is generally in the range of 20-40 seconds on the average, except for deadlock and "others". The table also shows that the errors not classified are relatively insignificant since their duration is less than 2 seconds. Program exceptions take twice as long as control errors (42 seconds versus 21 seconds). This is possibly due to the extensive software involvement in recovering from program exceptions. Figure 4.5 shows the density of waiting time in the normal operation state, i.e., the density of the time to error. This density could not be fitted to a simple exponential, and because of the shape of this density we found that it was fitted to a multi-stage gamma

Table 4.3. Mean waiting time (in seconds) of states

State	# of obs	Mean waiting time	Standard deviation	Std Error of mean
Normal	2757	10461.33	32735.04	623.44
Control	213	21.92	84.21	5.77
Deadlock	23	4.72	22.61	4.72
I/O & Data Management	1448	25.05	77.62	2.04
Program Exception	65	42.23	92.98	11.53
Storage Exception	149	36.82	79.59	6.52
Storage Management	313	33.40	95.01	5.37
Others	66	1.86	12.98	1.60
Multiple Error	481	175.59	252.79	11.53

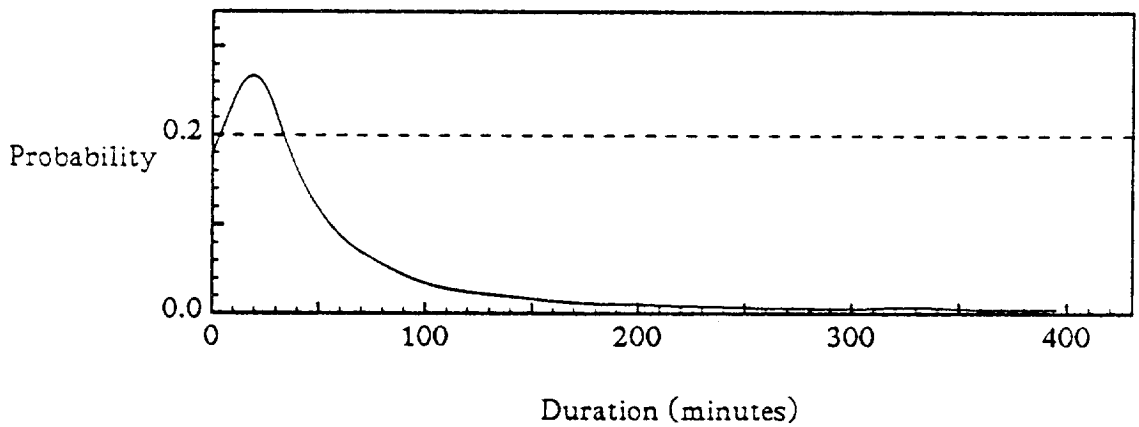


Figure 4.5. Time to error density

density function better than to a phase-type exponentials at the same acceptable level. The multi-stage gamma density function $f(t)$ is defined as

$$f(t) = \sum_{i=1}^{n} a_i g(t; \alpha_i, s_i), \tag{4.3.1}$$

where $a_i \geqslant 0$, $\sum_{i=1}^{n} a_i = 1$, and n is the number of stages. The $g(t; \alpha, s)$ is a gamma density function (with s the distance shifting from the origin),

$$g(t; \alpha, s) = \begin{cases} 0 & t < s \\ \dfrac{1}{\Gamma(\alpha)}(t-s)^{\alpha-1}e^{-(t-s)} & t \geqslant s \end{cases}, \tag{4.3.2}$$

where $\Gamma(\alpha)$ is a gamma function. Hence, the density in Figure 5 so obtained has five stages, given by

$$f(x) = 0.748\ g(x;\ 2.1,\ -1) + 0.055\ g(x;\ 0.5,\ 0) + 0.069\ g(x;\ 3.5,\ 3)$$
$$+\ 0.030\ g(x;\ 5.0,\ 8) + 0.098\ g(x;\ 5.0,\ 17),$$

tested using the Kolmogorov-Smirnov test [26] at the 0.01 significance level.

4.3.3. Recovery Time Distribution

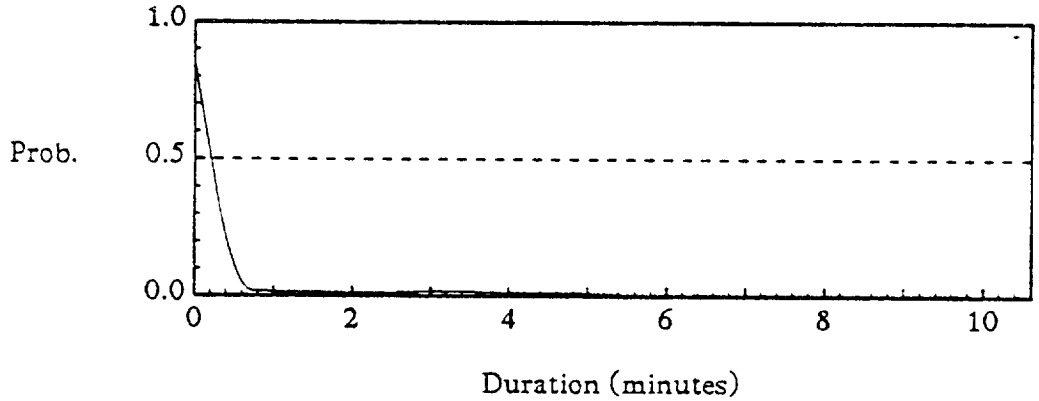
For the purposes of evaluating the time for recovery, we assumed that each recovery mode takes a constant amount of time. The overall recovery time, i.e., the duration of an error event (or the waiting time in an error state), however was not constant, since an error event can involve more than one recovery attempt or may require more than one recovery routine. The recovery time was then computed as the time difference between the first and the last detected error caused by the same event. The duration of an error event was used to measure the effectiveness of recovery from this event and also the severity of the error.

Figure 4.6 shows the recovery time densities for three different types of errors: I/O and data management, storage management, and multiple errors. Note that none of these densities could be fitted by simple exponentials at an acceptable level of significance. Thus, they were fitted to phase-type exponential density functions [26].

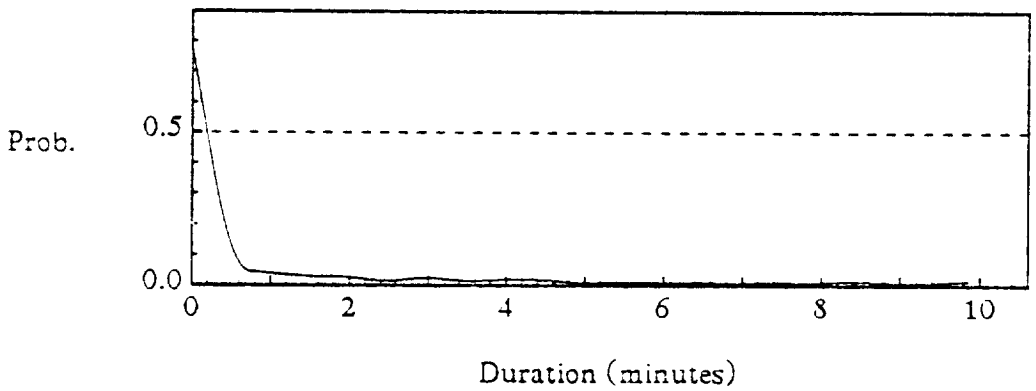
$$f(t) = \sum_{i=1}^{n} a_i g_i(t),$$

where $a_i \geqslant 0$, $\sum_{i=1}^{n} a_i = 1$ and n is the number of phases. The g_i function can be a simple exponential, a multi-stage hyperexponential, or a muti-stage hypoexponential density function. These exponentials are defined in Section 2.4. The densities in Figure 4.6 were fitted to the following functions (tested using the Kolmogorov-Smirnov test at the 0.01 significance level):

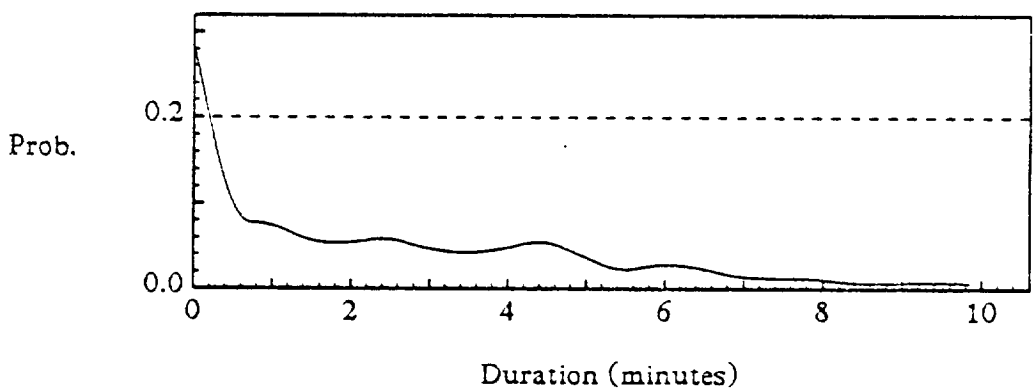
C1-C6



(a). Density for I/O and Data Management state



(b). Density for Storage Management State



(c). Density for multiple error state

Figure 4.6. Recovery time (error duration) densities

(1) I/O: $f(t) = 0.07825e^{-0.08594t} + 0.000354e^{-0.003958t}$,

(2) SM: $f(t) = 0.1642e^{-0.5t} + 0.030424e^{-0.053294t} + 0.0006634e^{-0.006586t}$, and

(3) MULT: $f(t) = 0.078e^{-0.4t} + 0.002426e^{-0.005866t} + 0.002163e^{-0.005525t}$.

4.3.4. Summary

In summary, the model developed explicitly quantifies the error and recovery process in the measured software. We note both the time to error and the recovery time distributions in several key states cannot be modeled as simple exponentials. Hence the overall process is modeled as a semi-Markov process. Further, the semi-Markov process is irreducible with the resume operation (resume op) state, the job termination (job term) state, and the task termination (task term) state being recurrent.

In the next sub-section we analyze the overall model to determine key software error characteristics. The mean time between different types of errors is evaluated along with model characteristics such as the occupancy probability of key error states.

4.4. Model Analysis

4.4.1. General Characteristics

By solving the semi-Markov model, we discover that the measured software system made a transition, on the average, every 43 minutes and 22 seconds. Table 4.4 lists the mean time between different software errors (i.e., mean time between errors) and Table 4.5 shows the mean recurrence time for recovery processes. By examining the mean recurrence time for I/O and MULT errors from Table 4.4 and comparing them with the mean waiting times in Table 4.3, we find that although the I/O errors occur about 3 times as often as the

Table 4.4. Mean time between errors

Error state	Frequency count	Percent (%)	MTBE (hour)
CTRL	213	7.72	37.83
DLCK	23	0.84	351.58
I/O	1448	52.50	5.56
PE	65	2.43	120.15
SE	149	5.40	54.08
SM	313	11.35	25.73
OTHR	66	2.32	125.84
MULT	481	17.44	16.75

Table 4.5. Mean recurrence time of recovery

Type of recovery	MTBR (hour)
Retry	4.25
Percolation	8.55
No-Percolation	241.43

multiple errors, the system spends nearly 6 times longer in recovering from a multiple error (25 seconds for I/O errors versus 175.6 seconds for multiple errors). This is because recovery from a multiple error involves several different types of recovery attempts. In addition, 63% of the multiple errors invoke percolation compared with the fact that 94% of the I/O errors recovered through retry (see Figure 4.4).

4.4.2. Model Probabilities

Given the irreducible semi-Markov model of Figure 4.4, the following steady state probabilities were evaluated. The derivations of these measures are given in Section 3.1.

(1) transition probability (π_j) - given that the process is now making a transition, the probability that the transition is to state j

(2) occupancy probability (Φ_j) - at any instant time the probability that the process occupies state j

(3) entry probability (e_j) - at any instant given that the process is entering a state, the probability that the process enters state j

(4) mean recurrence time ($\overline{\Theta}_j$) - mean recurrence time of state j

The model characteristics are summarized in Table 4.6. A dashed line in this table

Table 4.6. Characteristics of software error/recovery model

(a)

Measure	Normal state	Error state							
		CTRL	DLCK	I/O	PE	SE	SM	OTHR	MULT
π	0.2474	0.0191	0.0020	0.1299	0.0060	0.0134	0.0281	0.0057	0.0431
Φ	0.9950	0.00016	-	0.00125	0.000098	0.000189	0.00036	-	0.002913

(b)

Measure	Recovery state			Result		
	Retry	Percolation	No-Percolation	Resume op	Task term	Job term
π	0.1704	0.0845	0.0030	0.1414	0.0712	0.0348
$\overline{\Theta}^*$	4.25	8.55	241.43	5.11	10.16	20.74

* - in hour

indicates a negligible value (less than 0.00001 probability). From the occupancy probability, Φ, of the normal state in Table 4.6(a), we see that in about 99.5% of time the software system is operating normally, i.e., only 0.5% of time the system detects software errors. This indicates that the reliability of the measured software system can be as high as 0.995. In Section 2.2 we know that about 35% of observed errors were software errors. Thus the effect on the overall system reliability due to the software errors is very significant.

The table also shows that, of all possible transitions made, 24.73% are to an error state (obtained by summing all the π's for all the error states) and another 25.79% are to a recovery state. Since it was seen earlier that a transition occurs every 43 minutes, we estimate that a software error is detected, on the average, every 3 hours. From Table 4.6(b), we notice that although an error is detected almost every 3 hours, a successful recovery (i.e., results in resume operation), only occurs once every five hours, i.e., nearly 43% of the errors result in task/job termination.

Multiple-error events formed a significant category on their own. Since this type of event involves several errors and result in considerable overhead, it is analyzed separately in the next section.

4.4.3. Characteristics of A Multiple Error

In Section 4.2 we pointed out that about 17% of software errors were multiple errors. We also noticed that the multiple errors mostly consist of I/O, storage, or program errors. A strong connection between program and storage exception was seen in the occurrence of a multiple error. Table 4.7 lists the characteristics for a multiple error and was obtained by solving the semi-Markov model described in Figure 4.1 with a zero holding time in the normal state (i.e., given a multiple error occurs). From Table 4.7 we see (from π,

Table 4.7. Characteristics of a multiple error

Measure	Normal state	Error state						
		CTRL	DLCK	I/O	PE	SE	SM	OTHR
π	0.1767	0.0327	0.0048	0.1451	0.1473	0.2957	0.1360	0.0617
Φ	0	0.0648	0.0130	0.3004	0.0837	0.2202	0.2717	0.0462
e	0.00568	0.00105	0.00015	0.00466	0.00473	0.00950	0.00437	0.00198
$\overline{\Theta}^*$	0.0489	0.2647	1.8126	0.0596	0.0587	0.0292	0.0636	0.1401

* - in hour

transition probability) that nearly 30% of the transitions are made to the storage exception state when the process enters a multiple error mode. Once in a multiple error mode, a storage exception error occurs every 1 minute and 45 seconds ($\overline{\Theta}$ = 0.0292 hours in Table 4.7), while the average duration of multiple errors is about 2 minutes and 56 seconds ($\overline{\Theta}$ = 0.0489 hours, the recurrence time of the normal state). Note that the average duration of a multiple error predicted here from the model is very close to the mean duration of a multiple error, 175.5 seconds obtained from real data, listed in Table 4.3. This provides a strong evidence that the semi-Markov process is a good model for our measured system due to its fairly accurate prediction. As soon as an entry into a multiple error is made, consecutive errors are detected almost every 31 seconds (by taking the reciprocal of the sum of all entry probabilities e in Table 4.7). This indicates that about 5 to 6 errors will be detected on average, once a multiple error occurs.

There are several interesting characteristics of multiple errors which can be derived from the model of Figure 4.1. For example, if we want to know the probability of a storage exception error given an I/O error, we can evaluate it by the multi-step transition

probability to the SE state from the I/O state. This turns out to be very small, only 0.0076. However, we find that the probability of an I/O occurring given a SE occurs at any time instant, is as high as 0.668. This is partly due to the fact that for a semi-Markov process the unconditional transition probability at any time instant, γ_{ij}, is not only a function of conditional transition probability p_{ij} but also a function of mean holding time. This can be seen in Equation 2.5.1.

4.5. Conclusion

In this study, we have developed a semi-Markov model to describe the error and recovery processes in the MVS system. The model is based on real error data collected during normal system operation. The semi-Markov model developed provides a quantification of system error characteristics and the interaction between different types of errors. As an example, we provide a detailed model and analysis of multiple errors, which constitute approximately 17% of all software errors and result in considerable overhead. It is suggested that other systems be similarly analyzed and modeled so that a wide range of realistic models of software reliability in an operating environment are available.

CHAPTER 5

PERFORMABILITY MODEL

A workload/reliability model is built based on real data. Given a stochastic transition probability matrix and a holding time density matrix, the system behavior such as the unconditional transition probability and state occupancy probability in the steady state can be estimated. However, the performability of the measured system is not yet addressed. Thus in this chapter we use the resource-usage/error/recovery model to estimate the performability of the system. Reward functions are used to depict the performance degradation due to errors and also due to different types of recovery procedures. Toward this end, we define a reward rate for each state of the resource-usage/error/recovery model.

5.1. Reward Function

First, we propose the reward rate r_i (per unit time) for each state i in our model as follow:

$$r_i = \begin{cases} \dfrac{s_i}{s_i + e_i} & \text{if } i \in S_N \cup S_E \\ 0 & \text{if } i \in S_R \,. \end{cases} \tag{5.1.1}$$

where, the s_i and e_i are the service rate and the error rate in state i, respectively. Thus one

unit of reward is given for each unit of time when the process stays in the normal states S_N. The penalty paid depends on the number of errors generated by an error event. With an increasing number of errors the penalty per unit time increases, and accordingly, the reward rate decreases. Zero reward is assigned to recovery states. This is due to the fact that during the recovery process the system does not contribute any useful work toward the system performance besides recovering from an error. Based on this proposal, reward rates for the error states are estimated and shown in Table 5.1. We know that from Table 3.3(b) the transition probability to the DASD error is about as much as twice to the SWE error and Table 5.1 shows that the reward gained from the DASD state is also as much as twice from the SWE state. Thus we expect that the impact due to the DASD error on the performability is much higher than that due to the SWE error. In order to understand the effectiveness of various errors, first we show some important performability measures that can be derived from the model.

Since the system can be in any state at any instant, so the reward rate of the system at time t, $X(t)$, is the reward rate of the state where the system is currently occupied. It is a random variable and denoted as

Table 5.1. Reward rates, r_i, for error states

State	DASD	SWE	CHAN	MULT
r_i	0.5708	0.2736	0.9946	0.2777

$$X(t) = \left\{ r_i \mid \text{process is in state } i \text{ at time } t \right\}. \tag{5.1.2}$$

Therefore the expected reward rate at time t, $E[X(t)]$, can be evaluated as

$$E[X(t)] = \sum_i p_i(t) r_i \ , \tag{5.1.3}$$

where $p_i(t)$ is the probability of the process being in state i at time t. The cumulative reward by time t, $Y(t)$, can be derived from

$$Y(t) = \int_0^t X(\sigma) d\sigma \ . \tag{5.1.4}$$

Therefore, the expected cumulative reward at time t, $E[Y(t)]$, is given by [28]:

$$E[Y(t)] = E\left[\int_0^t X(\sigma) d\sigma\right] = \sum_i r_i \int_0^t p_i(\sigma) d\sigma \ . \tag{5.1.5}$$

In order to solve for $p_i(t)$ and hence other measures, we convert the semi-Markov process into a Markov chain using the method of stages [26,29]. The conversion of the semi-Markov model to the Markov model for the measured system is described in section 5.2. Thus the state probability vector $P(t) = (....p_i(t),....)$ can be computed by solving the set of differential equations of the form:

$$\frac{d}{dt} P(t) = P(t) Q$$

where Q is transition rate matrix of the Markov chain [25].

5.2. Semi-Markov to Markov Conversion

As we know, the sojourn time distribution of states is the only difference between semi-Markov and Markov models. For a Markov model the sojourn time distributions of states must be exponentials, however, it can be any distribution for a semi-Markov model. Thus, to convert a semi-Markov to a Markov process, one must change the non-exponential distributions to exponentials. In this section, we show how to convert a state with non-exponential distribution to a number of states in which each state is exponential.

In Section 2.4 we fitted the state holding times of our resource-usage/error/recovery process to the phase-type exponentials. The phase-type exponential function $f(t)$ can be expressed as

$$f(t) = \sum_{i=1}^{n} a_i g_i(t) ,$$

where $a_i > 0$, $\sum_{i=1}^{n} a_i = 1$, and n is the number of phases. For each phase i, the $g_i(t)$ can be a simple exponential, a multi-stage hyperexponential, or a multi-stage hypoexponential. The definitions of these three types of functions are listed below.

Exponential : EXP (λ)

$$\mathrm{EXP}(\lambda) = \lambda e^{-\lambda t}$$

Hyperexponential : Hyper ($\lambda_1, \lambda_2, \ldots, \lambda_r$)

$$\mathrm{Hyper}(\lambda_1, \lambda_2, \ldots, \lambda_r) = \sum_{i=1}^{r} a_i \mathrm{EXP}(\lambda_i) = \sum_{i=1}^{r} a_i \lambda_i e^{-\lambda_i t} ,$$

where $a_i > 0$, $a_i \geqslant 0$, and $\sum_{i=1}^{r} a_i = 1$.

Hypoexponential : Hypo $(\lambda_1, \lambda_2, \ldots, \lambda_r)$

$$\text{Hypo}(\lambda_1, \lambda_2, \ldots, \lambda_r) = \sum_{i=1}^{r} a_i \text{EXP}(\lambda_i) = \sum_{i=1}^{r} a_i \lambda_i e^{-\lambda_i t},$$

where $\lambda_i > 0$, $\lambda_i \neq \lambda_j$ if $i \neq j$, and $a_i = \prod_{\substack{i=1 \\ i \neq j}}^{r} \frac{\lambda_j}{\lambda_j - \lambda_i}$.

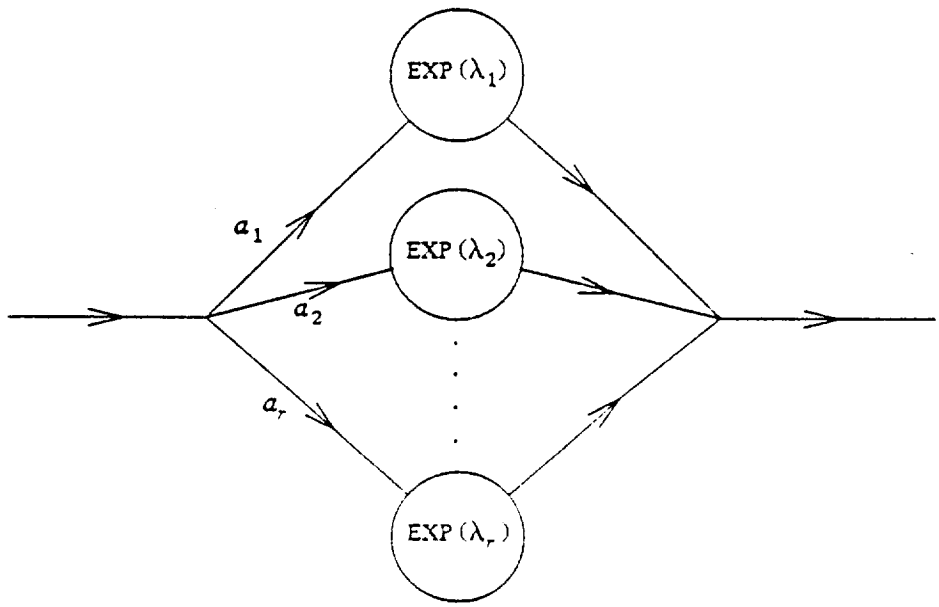
By using the method of stages [26], a hyperexponential distribution can be modeled as a set of parallel exponential stages and a hypoexponential distribution as a set of series exponential stages. Figure 5.1(a) and 5.1(b) show the conversions of these two types. In Figure 5.1(a) we note that each a_i of hyperexponential function is converted to the probability to the associated state having density EXP(λ_i), however this is not the case for the hypoexponential. From Figure 5.1(b), we know that the Markov version of the hypoexponential is just a series connection of states in which each state has an simple exponential density function and the probability from one state to another is one. As an example we know in section 2.3 that the holding time density from state W_8 to error state DASD is fitted by

$$f(t) = 0.235 \text{ EXP}(0.004) + 0.765 \text{ Hypo}(0.00093, 0.006595),$$

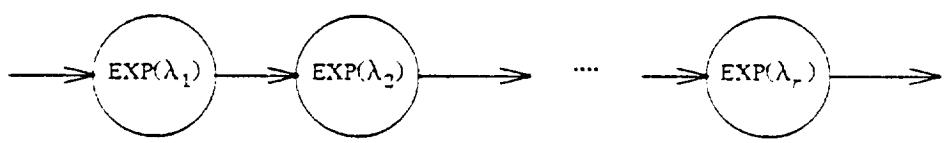
which is a combination of hyper and hypo exponentials. The Markov conversion of the state with Equation 5.2.1 holding time density is shown in Figure 5.2. Note that the state W_8 in Figure 5.2(a) is modeled as a three state Markov process. This is shown in the dotted area of Figure 5.2(b).

5.3. Performability Analysis

After converting a semi-Markov process to a Markov process, analysis can be carried out on the resulting Markov reward model of the measured system using SHARPE (the

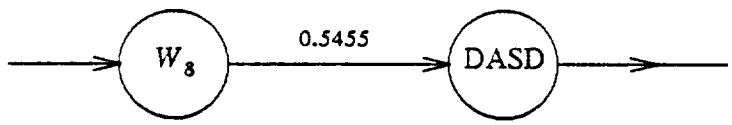


(a). Hyperexponential distribution as a set of parallel exponential stages

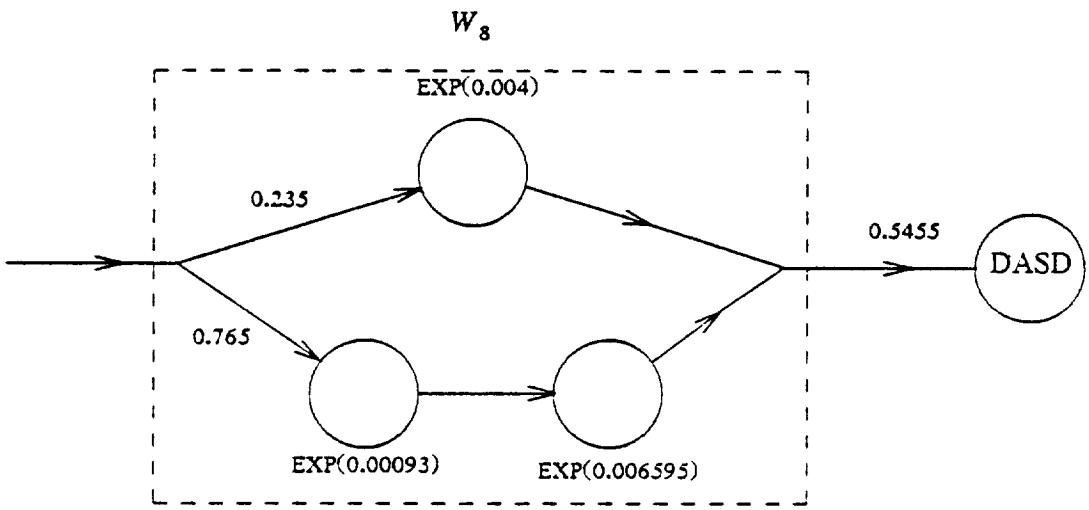


(b). Hypoexponential distribution as a set of series exponential stages

Figure 5.1. The conversion of non-exponential to a set of exponentials



(a). Semi-Markov



(b). Markov

Figure 5.2. The Markov conversion of State W_8

Symbolic Hierarchical Automated Reliability and Performance Evaluator) [29]. SHARPE is a modeling tool developed at Duke University. It provides several model types ranging from reliability block diagrams to complex semi Markov models, and allows the user to construct and analyze performance, reliability and availability models. However, this tool can only be used to analyze a model with size less then 200 states, thus we assume our

resource-usage/error/recovery model is a independent semi-Markov process. The conversion of the waiting times of states are shown in Appendix C.

In order to study the impact of different types of errors, the irreducible semi-Markov process is converted to one with absorbing states in the following manner:

a) with OFFL as the absorbing state (OFFL),

b) with MULT and OFFL as the absorbing states (MULT),

c) with SWE, MULT and OFFL as the absorbing states (SWE),

d) with DASD, MULT and OFFL as the absorbing states (DASD), and

e) with DASD, SWE, MULT and OFFL as the absorbing states (ALL).

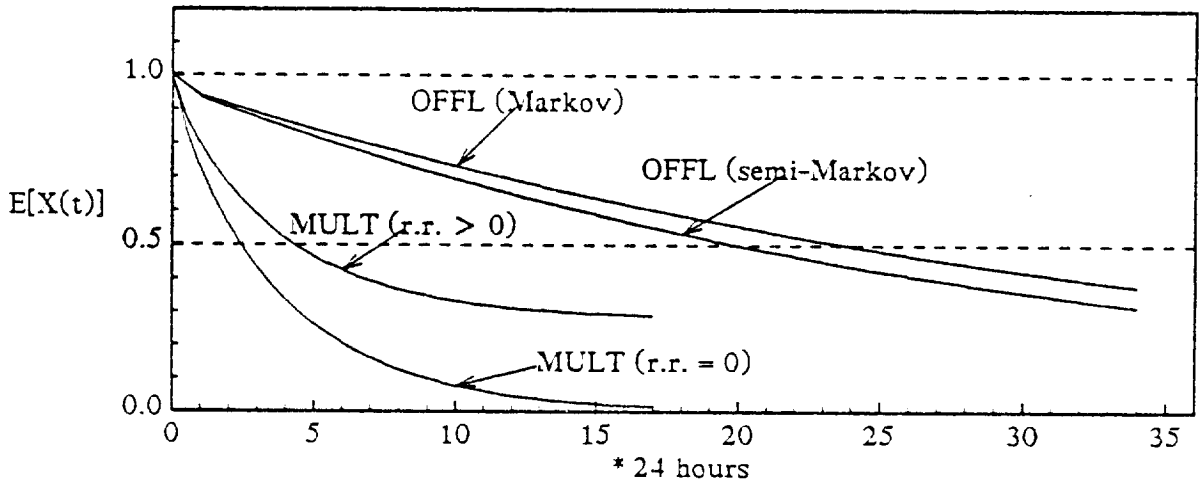
In case (a) we assess system performability in which all but off-line failures are not recovered from. This actually provides us with the result of the system reliability. In case (b) we discontinue recovering from multiple errors. Here, we expect to measure the impact on the reward to a multiple error. Since multiple errors happen much more frequently than OFFL and the sojourn time is much longer comparing with others, we expect to measure the impact of SWE and DASD on the reward to a MULT error. Thus, in case (c) we not only stop recovering from multiple and off-line failures but we also stop the recovery from software errors. In case (d) we recover from SWE errors but stop recovery from DASD errors. Finally, in case (e) we do not recover from any errors besides CHAN.

We compare these scenarios first using the expected instantaneous reward rate $E[X(t)]$ which is defined by Equation 5.1.3, then using the time-averaged expected accumulated reward $\frac{E[Y(t)]}{t}$. In all but case (a) and (e) we consider two variations: when a state such as DASD (MULT or SWE) is made absorbing, we can either let the reward rate in such a state be non-zero or we can set its reward rate to zero. The impact of the non-zero assign-

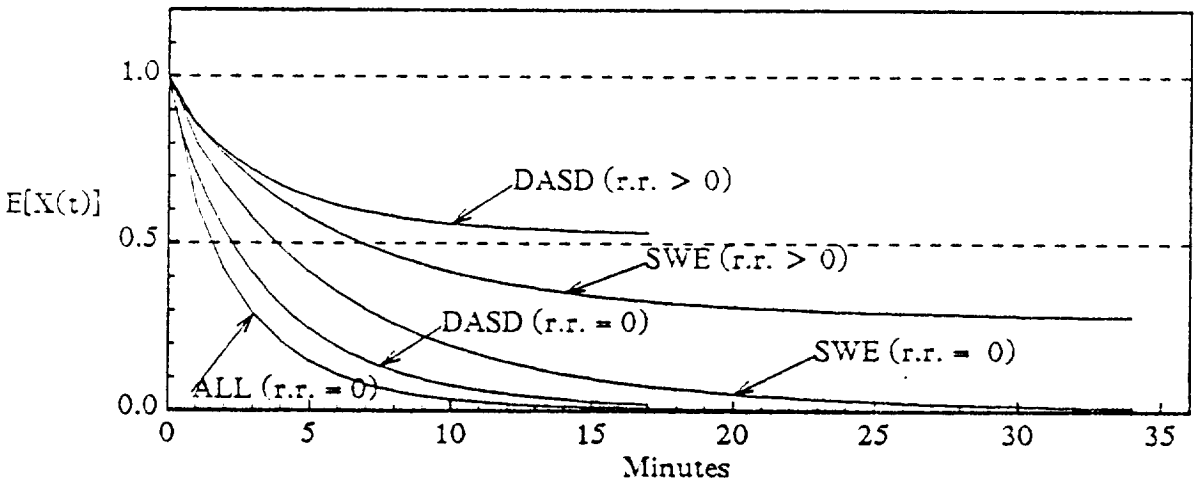
ment is that upon reaching the absorbing state, the system continues to operate in a degraded mode. In the latter case, i.e., zero reward assignment, we conservatively assume that the system stops functioning when it reaches the absorbing state(s).

In Figure 5.3(a), we plot $E[X(t)]$ for cases (a) and (b). In the case (b) we use two different assumptions for the reward rate for the MULT state, $r_{MULT}=0.27777$ and $r_{MULT}=0$. We also plot $E[X(t)]$ for case (a) with the assumption that all states have exponentially distributed holding times. We note that such a Markovian assumption leads to an overestimation of the system's capability to perform useful work, and the degree of overestimation increases as the system operating time increases. We also note that not recovering from multiple errors considerably degrades the system's performability. Moreover, changing from non-zero to zero reward rate further reduces the system's effectiveness drastically.

In Figure 5.3(b), we plot the $\frac{E[X(t)]}{t}$ for cases c, d and e. In each case, except case e, we also have two versions with reward rates for absorbing states being non-zero and zero, respectively. Note that not recovering from SWE errors degrades system effectiveness considerably compared with the effect of not recovering from DASD errors, provided we assume that absorbing states continues to provide service in a degraded mode. On the other hand, if we assume that absorbing states are system failure states, i.e., zero reward rates for absorbing states, then not recovering from DASD failures is more severe than not recovering from SWE failures. This behavior is explained by the fact that the reward rate in the DASD state is about twice that in the SWE state (0.5708 versus 0.2736 in Table 5.1). Figures 5.4(a) and 5.4(b) are the counterparts of Figures 5.3(a) and 5.3(b) where the measure plotted is $\frac{E[Y(t)]}{t}$ rather than $E[X(t)]$. The trends are similar.



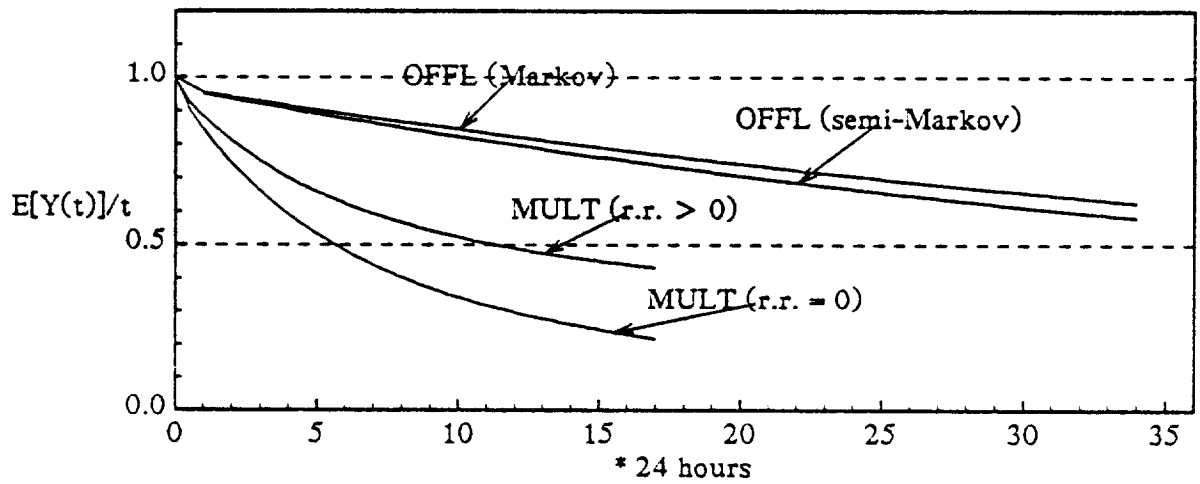
(a)



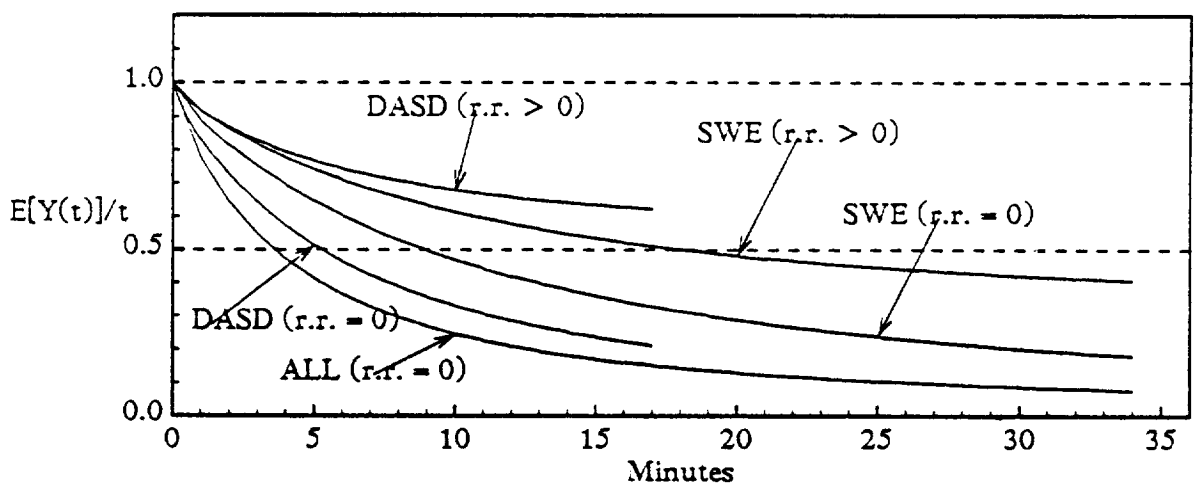
(b)

(r.r. : reward rate)

Figure 5.3. The expected reward rate, E[X(t)]



(a)



(b)

(r.r. : reward rate)

Figure 5.4. The time-averaged accumulated reward, $E[Y(t)]/t$

Finally, in Figure 5.5, we show the distribution function of $Y(\infty)$, the accumulated reward until system failure, for two cases: Markov versus semi-Markov. Both assume that the OFFL state is the only absorbing state. Once again we note that the Markovian assumption implies an overestimation of the system's performability.

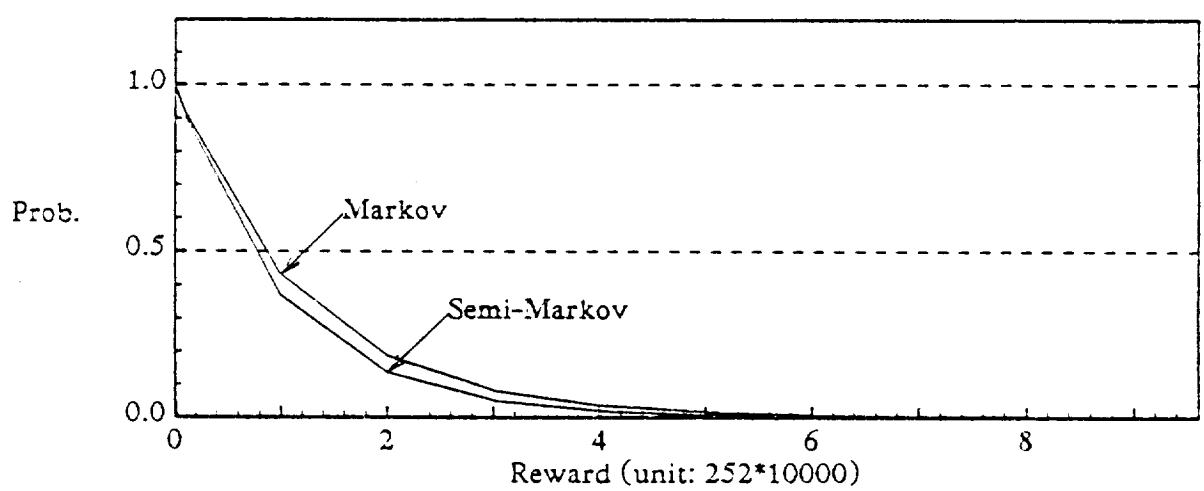


Figure 5.5. Distribution of accumulated reward until system failure

CHAPTER 6

SUMMARY AND CONCLUSIONS

6.1. Summary of Results

This thesis has developed a methodology to construct a resource-usage/reliability/performability model for a complex system based on real data. The model obtained is capable of reflecting both the normal and error behavior of the system. Both hardware and software reliability and their interactions are modeled. The effect of recovery through the built-in recovery mechanisms is also considered. By modeling the recovery process we are able to evaluate the severity of errors in general and the cost of specific error type in particular. Low-level error and resource-usage data to develop the model was collected on an IBM 3081 machine running the MVS operating system. The results of this research suggest that other production systems should be similarly analyzed so that a body of realistic data on computer error (including failure) and recovery models is available.

Chapter 2 described the development of the model, using the low level data on resource usage and errors. A statistical clustering method (k-means clustering) was employed to characterize the resource usages into a few workload clusters. A two-level error data reduction (error coalescing and grouping) scheme was used to identify individual error incidents. Results showed that about 17% of errors are multiple errors (believed to be multiple manifestations of the same problem). The state-transition diagram for a multiple

error was obtained to study the interaction between system components (hardware and software). For example, it was seen that software and disk errors were strongly correlated.

From the measurement data it was seen that the holding times in key operational and error states were not simple exponentials. A semi-Markov process was used to model the system behavior. This (semi-Markov) assumption was also validated by comparing the state occupancy probabilities predicted by the model with the actual state occupancy probabilities estimated from observed data. The results show that the proposed model provided a fairly accurate prediction of the real behavior.

The analysis of model behavior was performed in Chapter 3. The analysis showed that on-line recovery is highly effective and provides the system with the ability to tolerate many faults and recover almost instantaneously. An analysis to extract the effect of the workload on the error probability showed that not only does a higher workload result in a higher error probability (for similar holding time), but the error probability also increases with increased holding time in a particular workload state. In other words, the error probability appears to be a function of the absolute amount of resource consumed, be it through increased workload and/or increased holding times. An explanation for this "wear out" phenomenon is not clear since a large majority of the collected errors are transient, but it certainly calls into question the validity of the frequently used constant error probability assumption used in reliability modeling.

The significance of the use of a semi-Markov model, as opposed to the simple Markov model, to describe the overall resource-usage/error/recovery process was also investigated. The results showed that a simple Markov model frequently overestimates the unconditional transition probabilities and underestimates the variance of the first passage times to the error states. The overestimation can lead to an unduly conservative reliability predic-

tion and the underestimation may lead to unduly optimistic reliability prediction. Both over- and under- estimations are not desirable.

In Chapter 4, the software error data was used to build a software reliability model to describe the error and recovery processes in the MVS operating system. The semi-Markov model developed provided a quantification of the operating system error characteristics and also the interaction between different types of OS errors. We estimate that in only 0.5% of the cases the measured software system is unable to recover. A detailed model and analysis of multiple software errors, (which constitute approximately 17% of all software errors) was provided, showing how a single software problem can have multiple manifestations. To investigate the validity of this model, the duration of a multiple error predicted from the model was compared with the value estimated from the observed data. The agreement between two results was found to be within 1%.

A measurement-based performability model was discussed in Chapter 5. A reward function, based on the service rate and the error rate in each state, was proposed. In order to investigate the impact due to different errors, the expected reward rate, as well as the cumulative reward, at time t were estimated. The results show that the software error (SWE) degrades the system performance more severely than the disk error (DASD) although the error probability of DASD errors is about twice as much as that of SWE errors (0.169 versus 0.085). This may be due to the cost for DASD errors, which is less than that for SWE errors, i.e., the reward rate in DASD state is higher than that in SWE state. If, however, both error types result in system failure then, as expected, the DASD error degrades the system performance more severely than the SWE error.

The system performability under a Markov assumption is also estimated and compared with that estimated from the more realistic semi-Markov model. It was found that

the Markov assumption overestimates the system performability and that the degree of overestimation increased with increased system operation time. Once again, this indicates that the traditional Markov process is not good enough to model a computer system and to provide accurate predictions.

6.2. Suggestions for Future Research

The results of this study suggest that other systems be similarly studied so that a wide body of realistic results on computer system hardware and software performability are available. This is useful both, from the point of view of validating existing analytical models and from the point of view of generating realistic models of system behavior.

A possible extension is the area of adaptive model construction. The workload and error clustering methods employed here have potential for use in an adaptive algorithm which is capable of real-time model construction. The use of such models for adaptive tuning for optimum performability under various conditions needs to be investigated. To be successful such a system would require learning capabilities so as to use valid past information together with some knowledge of the environment for both reconfiguration under failure and for system tuning.

In this thesis we have used past data on errors and workload for model construction. It would be interesting to investigate the possibility of doing the same on the basis of data generated from error/failure injection on a prototype or into a simulation model of a system. Such a procedure has the potential of providing realistic feedback to system designers early in the development stage. A comparison of the results from such a model with those obtained through analytical models would be instructive as well.

REFERENCES

[1] Castillo, X. and Siewiorek, D.P., "A Performance-Reliability Model for Computing Systems," *IEEE*, 1980.

[2] Castillo, X. and Siewiorek, D.P., "A Workload Dependent Software Reliability Prediction Model," in *Proceedings of the 12th International Symposium on Fault-Tolerant Computing*, Santa Monica, California, pp. 279-285, June 22-24, 1982.

[3] Kulkarni, V.G., Nicola, V.F., Smith, R.M., and Trivedi, K.S., "Numerical Evaluation of Performability Measures and Job Completion Time in Repairable Fault-Tolerant Systems," in *Proceedings of the 16th International Symposium on Fault-Tolerant Computing*, Vienna, Austria, pp. 252-257, July 1-4, 1986.

[4] Meyer, J.F., "Closed-Form Solutions of Performability," *IEEE Transactions on Computers*, pp. 648-657, July 1982.

[5] Geist, R.M. and Trivedi, K., "Ultrahigh Reliability Prediction for Fault-Tolerant Computer Systems," *IEEE Transactions on Computers*, pp. 1118-1127, December 1983.

[6] Trivedi, K., Dugan, J.B., Geist, R., and Smotherman, M., "Modeling Imperfect Coverage in Fault-Tolerant Systems," in *Proceedings of the 14th International Symposium on Fault-Tolerant Computing*, Kissimme, Florida, pp. 77-82, June 20-22, 1984.

[7] Ng, Y.W. and Avizienis, A.A., "A Unified Reliability Model for Fault-Tolerant Computers," *IEEE Transactions on Computers*, pp. 1002-1011, November 1980.

[8] Goyal, A. and Tantawi, A.N., "Numerical Evaluation of Guaranteed Availability," in *Proceedings of the 15th International Symposium on Fault-Tolerant Computing*, Ann Arbor, Michigan, pp. 324-329, June 19-21, 1985.

[9] Schoen, O., "On a Class of Integrated Performance/Reliability Models based on Queuing Networks," in *Proceedings of the 16th International Symposium on Fault-Tolerant Computing*, Vienna, Austria, pp. 90-95, July 1-4, 1986.

[10] Iyer, R.K., Rossetti, D.J., and Hsueh, M.C., "Measurement and Modeling of Computer Reliability as Affected by System Activity," *ACM Transactions on Computer Systems*, vol. 4, no. 3, pp. 214-237, August, 1986.

[11] Goel, A.L., "Software Reliability Models: Assumptions and Applicability," *IEEE Transactions on Software Engineering*, vol. SE-11, pp. 1411-1423, December 1985.

[12] Yamada, S. and Osaki, S., "Software Reliability Growth Modeling," *IEEE Transactions on Software Engineering*, vol. SE-11, pp. 1431-1437, December 1985.

[13] Musa, J., "The Measurement and Management of Sofware Reliability," *IEEE Proceedings*, vol. 68, pp. 1131-1143, September 1980.

[14] Littlewood, B, "Theories of Software Reliability: How good are they and how can they be improved?," *IEEE Transactions on Software Engineering*, vol. SE-6, pp. 489-500, September 1980.

[15] Iyer, R. K. and Velardi, P., "Hardware-Related Software Errors: Measurement and Analysis," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 2, pp. 223-231, February 1985.

[16] Kelly, J. and Avizienis, A.A., "A Specification-Oriented Multi-Version Software Experiment," in *Proceedings of the 13th International Symposium on Fault-Tolerant Computing*, Milano, Italy, pp. 120-126, June 28-30, 1983.

[17] Beounes, C and Laprie, J.C., "Dependability Evaluation of Complex Computer Systems: Stochastic Petri Net Modeling," in *Proceedings of the 15th International Symposium on Fault-Tolerant Computing*, Ann Arbor, Michigan, pp. 364-369, June 19-21, 1985.

[18] Castillo, X., *A Compatible Hardware/Software Reliability Prediction Model*. PhD Thesis, Carnegie-Mellon University, July, 1981.

[19] IBM Corperation, "MVS Architecture Resource Measurement Facility (RMF) Reference and User's Guide," *IBM publications*, vol. LC28-1138-1.

[20] Ferrari, D., Serazzi, G., and Zeigner, A., *Measurement and Tuning of Computer Systems*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1981.

[21] MacQueen, J., "Some Methods for Classification and Analysis of Multivariate Observations," in *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, vol. 1, Berkeley, California, pp. 281-297, June 21-July 18, 1965.

[22] Spath, H. *Cluster Analysis Algorithms*. West Sussex, England: Ellis Horwood Ltd., 1980.

[23] IBM Corperation, *Environmental Record Editing & Printing Program*. International Business Machines Corporation, 1984.

[24] Iyer, R.K., Young, L.T., and Sridhar, V., "Recognition of Error Symptoms in Large Systems," in *Proceedings of the 1986 IEEE-ACM Fall Joint Computer Conference*, Dallas, Texas, pp. 797-806, November 2-6, 1986.

[25] Howard, Ronald A., *Dynamic Probabilistic Systems*. New York: John Wiley & Sons, Inc.,, 1971.

[26] Trivedi, K.S., *Probability & Statistics with Reliability, Queuing, and ComputerScience Applications*. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1982.

[27] Hsueh, M.C., Iyer, R.K., and Trivedi, K.S., "A Measurement-Based Performability Model for a Multiprocessor System," in *Proceedings of the 2nd International Workshop on Applied Mathematics and Performance/Reliability Models of Computer/Communication Systems*, Rome, Italy, May 25-29, 1987.

[28] Smith, R.M., Trivedi, K.S., and Nicola, V.F., "The Analysis of Computer Systems Using Markov Reward Processes," 1987.

[29] Sahner, R.A. and Trivedi, K.S., "SHARPE: Symbolic Hierachical Automated Reliability and Performance Evaluator," in *User's Guide*, Durham, NC, September 1986.

APPENDIX A

(I). Error Clustering

Identical errors occurring within 5 minutes of each other were coalesced into a single event. This was done to ensure that the analysis is not biased by failure records relating to the same problem. The clustering algorithm analyzes the data and merges observations which occur in rapid succession and relate to the same problem. For each failure point, the following test was performed :

```
IF    <error type> = <type of previous error> AND
      <time away from previous error> ≤ 5 minutes
THEN
      <fold error into cluster being built>
ELSE
      <start a new cluster>
```

The result is a set of clustered errors. Associated with each cluster is information consisting of error classifications, number of points in the cluster, time of first and last errors in the cluster, and a variety of status data provided by the hardware and operating system.

(II). Error Grouping

A visual examination of the error clusters showed the existence of sets of clusters occurring within a short time interval. The close time proximity among some clusters means a substantial increase in the system error rate during that period. The high error

rate introduces the suspicion that the errors occurring during the high error rate period may be related, i.e., different errors may be due to a single cause, to multiple but related causes, or to multiple and independent causes. Therefore, the high error rate periods are formed by grouping all error clusters occurring within a small time interval of each other. This interval was chosen to be 5 minutes. The result is a set of grouped errors. The primary difference between a cluster and a group is that clusters contain only occurrences of the same error (same error type and machine state), whereas groups contain occurrence of different errors (different error type or machines state).

APPENDIX B

The Characteristics of the Resource-Usage/Error/Recovery Model

(I). Stochastic transition probability matrix.

(Due to the size of the matrix, it is broken into two parts, (a) and (b).)

	W_0	W_1	W_2	W_3	W_4	W_5	W_6	W_7	W_8
W_0	0	0.351	0.055	0.108	0.135	0	0.216	0.027	0.108
W_1	0.019	0	0	0.057	0.170	0	0.113	0.132	0.019
W_2	0	0	0	1	0	0	0	0	0
W_3	0.143	0.048	0	0	0.143	0.048	0.048	0.048	0
W_4	0.023	0.045	0	0	0	0.015	0.046	0.099	0.038
W_5	0.091	0	0	0	0	0	0.091	0	0.091
W_6	0.034	0.034	0	0.013	0.054	0	0	0.087	0.067
W_7	0.040	0.011	0	0	0.022	0	0.051	0	0.069
W_8	0.093	0	0	0	0.007	0	0.015	0.063	0
CPU	0	0	0	0	0	0	0	0	0
CHAN	0	0	0	0	0	0	0	0	0
SWE	0	0	0	0	0	0	0	0	0
DASD	0	0	0	0	0	0	0	0	0
MULT	0	0	0	0	0	0	0	0	0
HWR	0.018	0.036	0	0.014	0.013	0.013	0.144	0.314	0.329
SWR	0.006	0.046	0	0.008	0.160	0.007	0.183	0.286	0.306
ALT	0	0	0	0	0	0	0	0	0
OFFL	1	0	0	0	0	0	0	0	0

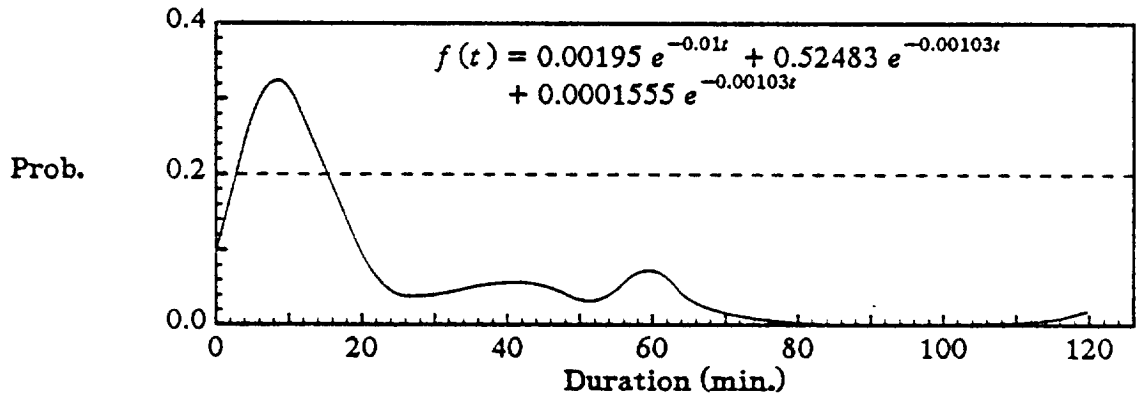
(a)

	CPU	CHAN	SWE	DASD	MULT	HWR	SWR	ALT	OFFL
W_0	0	0	0	0	0	0	0	0	0
W_1	0	0.019	0.151	0.226	0.094	0	0	0	0
W_2	0	0	0	0	0	0	0	0	0
W_3	0	0	0.143	0.381	0	0	0	0	0
W_4	0	0.015	0.227	0.386	0.106	0	0	0	0
W_5	0	0	0.091	0.545	0.091	0	0	0	0
W_6	0	0.007	0.262	0.383	0.060	0	0	0	0
W_7	0	0.026	0.208	0.482	0.091	0	0	0	0
W_8	0	0.007	0.231	0.502	0.082	0	0	0	0
CPU	0	0	0	0	0	1	0	0	0
CHAN	0	0	0	0	0	1	0	0	0
SWE	0	0	0	0	0	0.5	0.5	0	0
DASD	0	0	0	0	0	1	0	0	0
MULT	0	0	0	0	0	0.643	0.355	0	0
HWR	0	0	0	0	0	0	0.017	0	0
SWR	0	0	0	0	0	0	0	0	0.003
ALT	0	0	0	0	0	0	0	0	0
OFFL	0	0	0	0	0	0	0	0	0

(b)

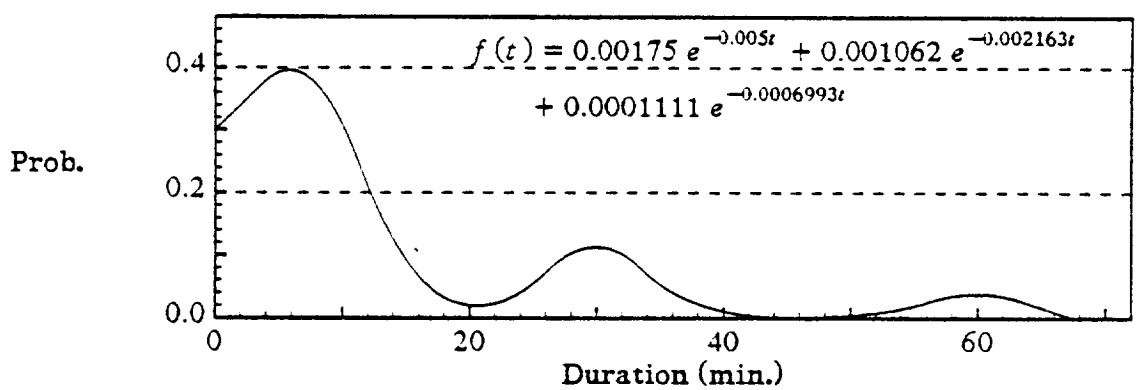
(II). Waiting and holding time densities. Constant sojourn times are assigned in W_0, W_2, CPU, CHAN and recovery states. The others are shown below.

Waiting time density of W_1



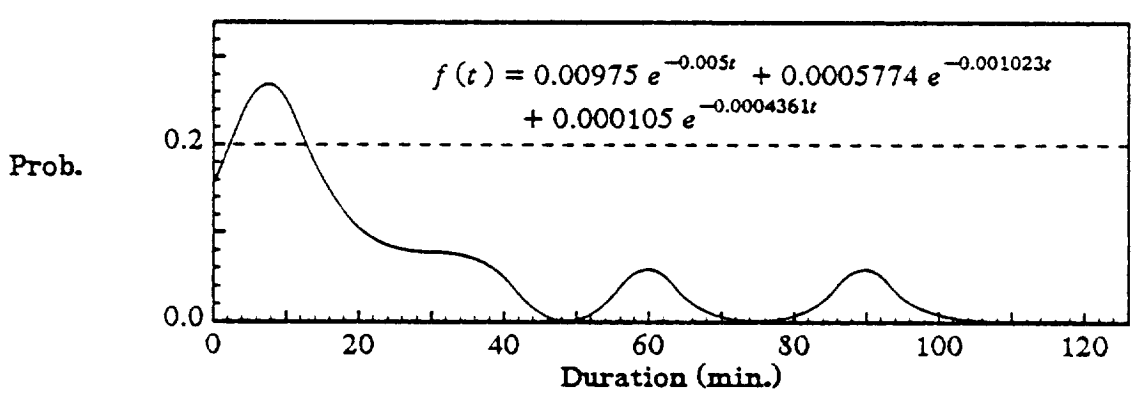
Waiting time density of W_1

Waiting time density for W_3



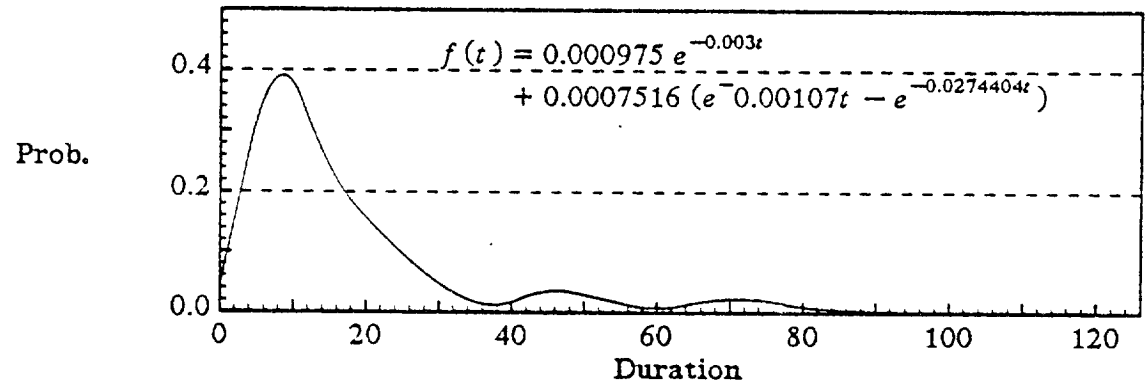
Waiting time density for W_3

Waiting/holding time densities for W_4

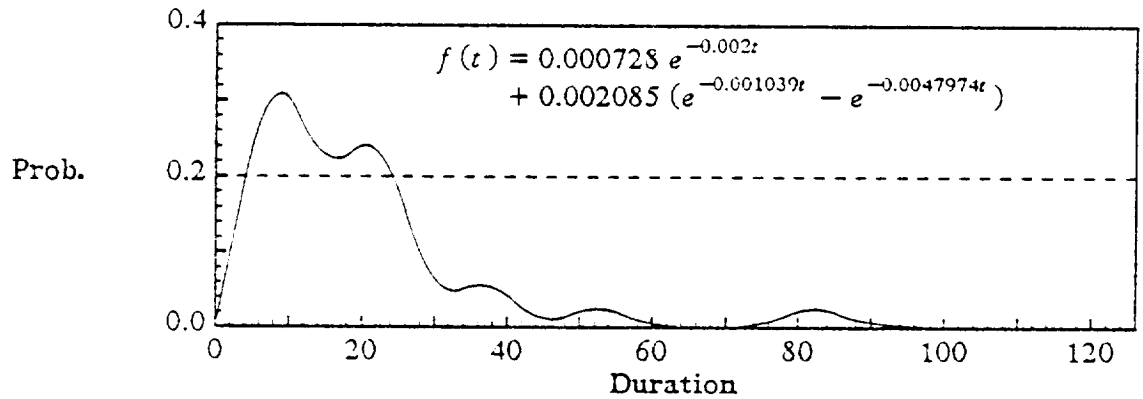


Holding time density to W_7

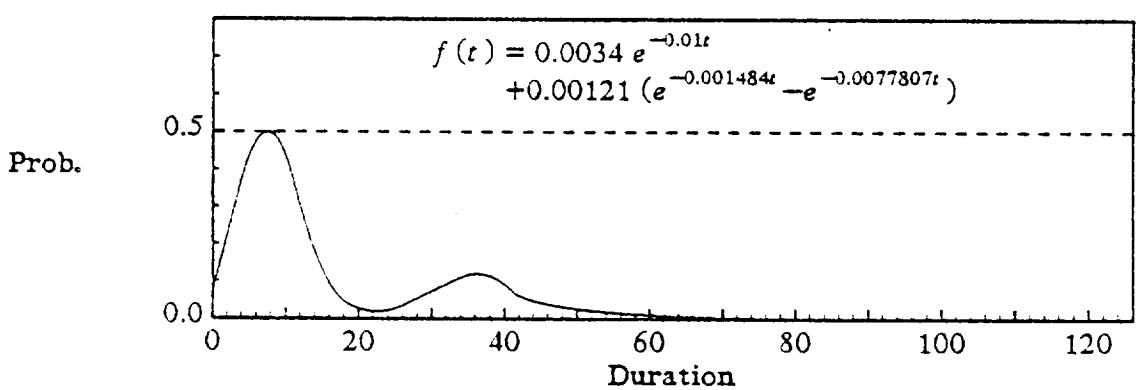
Waiting/holding time densities for W_4



Holding time density to DASD

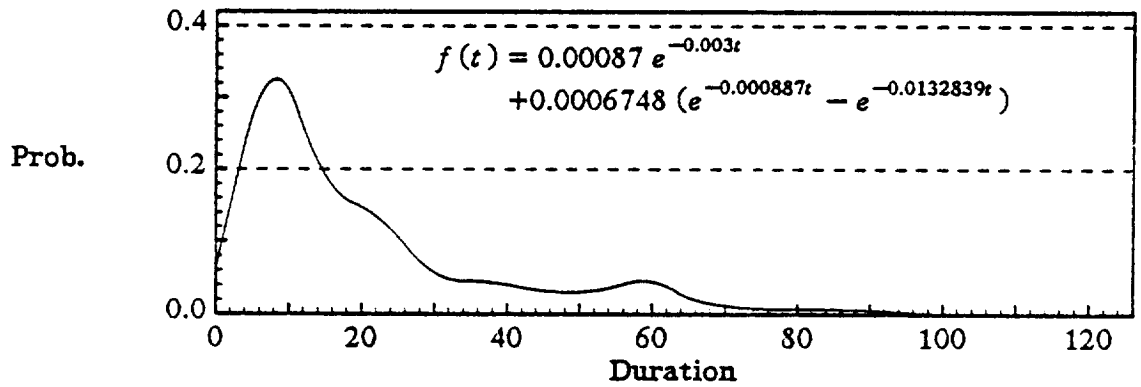


Holding time density to SWE



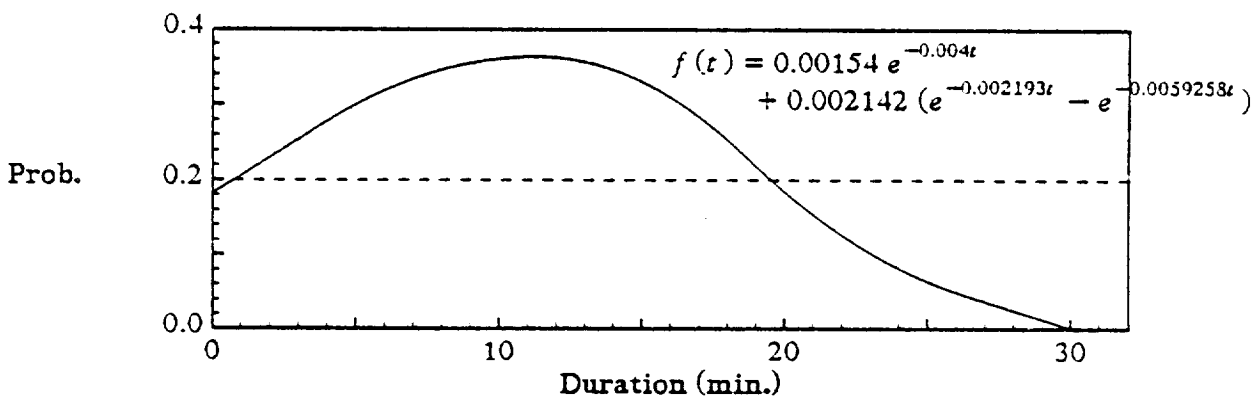
Holding time density to MULT

Waiting/holding time densities for W_4



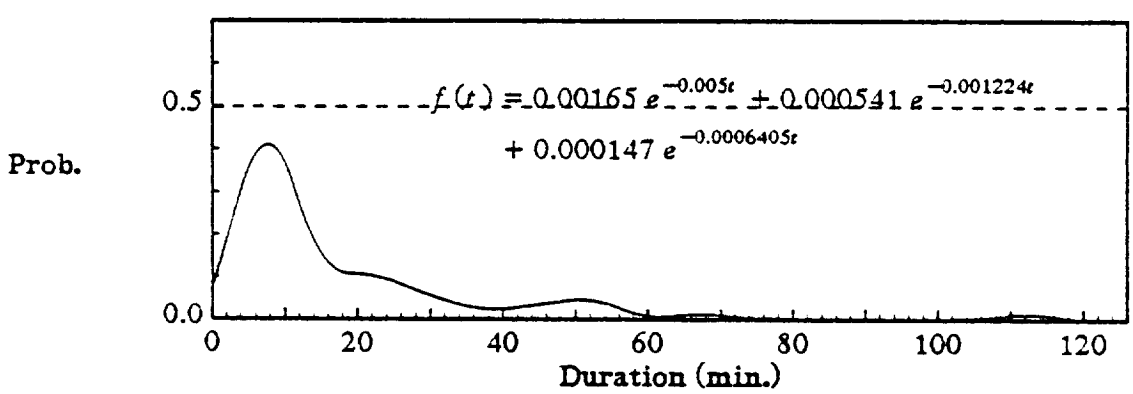
Holding time density to the others

Waiting time density of W_5



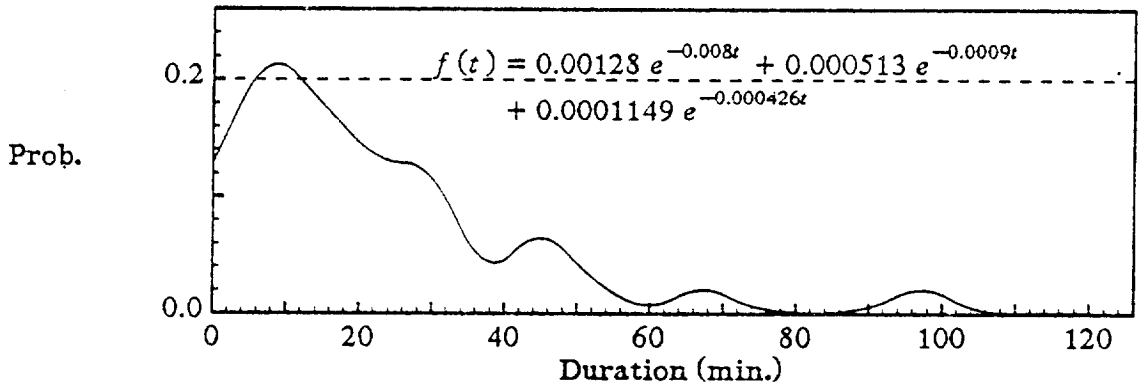
Waiting time density of W_5

Waiting/holding time densities for W_6

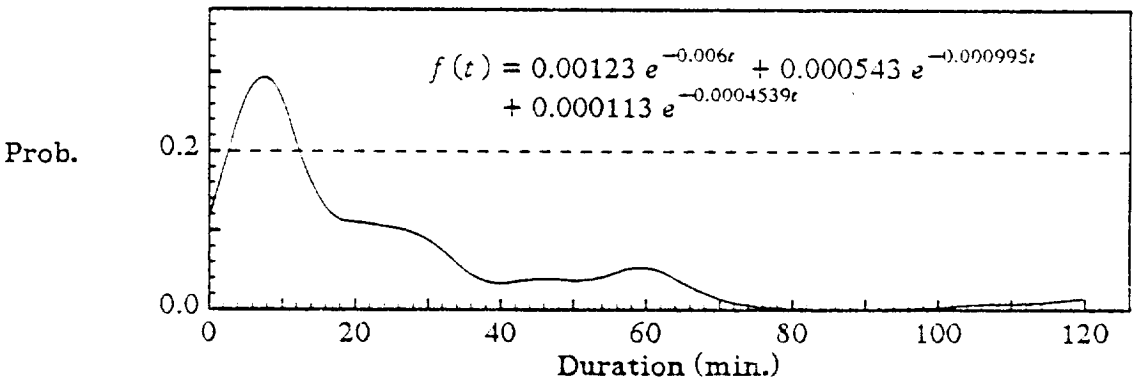


Holding time density to DASD

Waiting/holding time densities for W_6

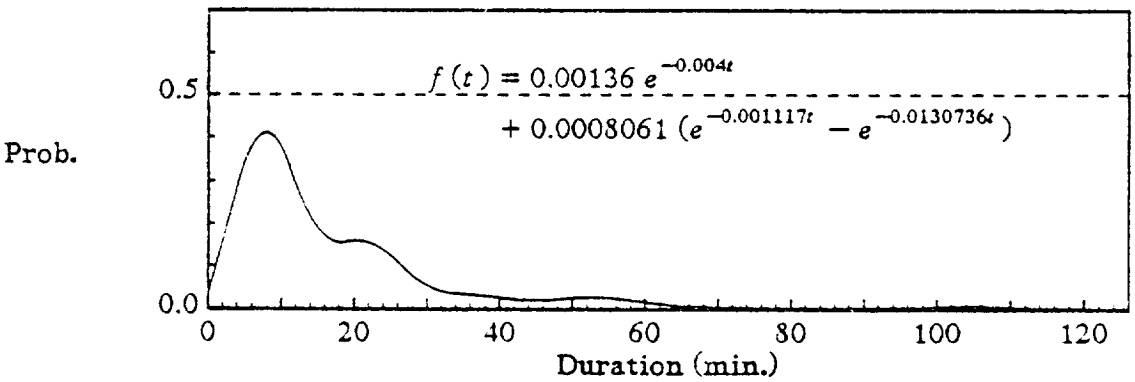


Holding time density to SWE



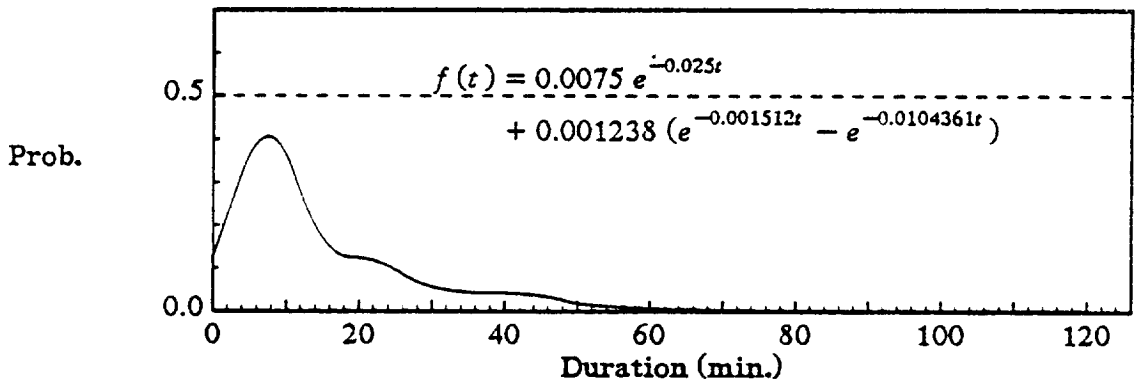
Holding time density to the others

Waiting/holding time densities for W_7

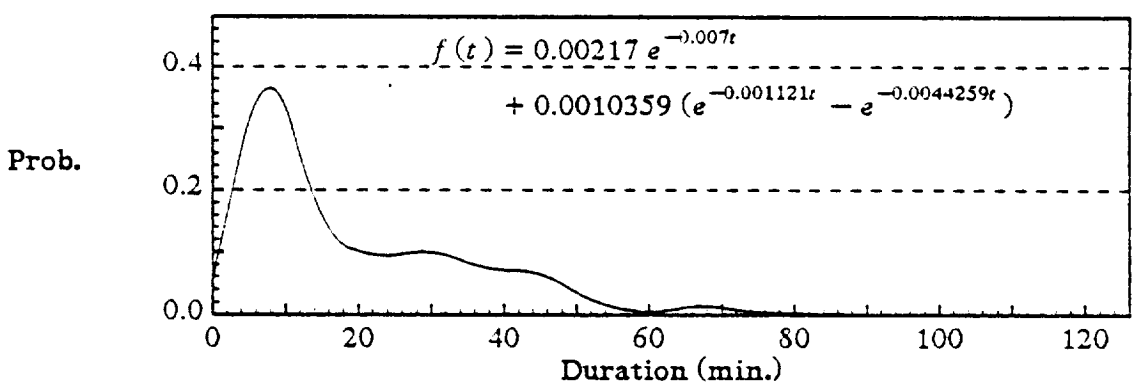


Holding time density to DASD

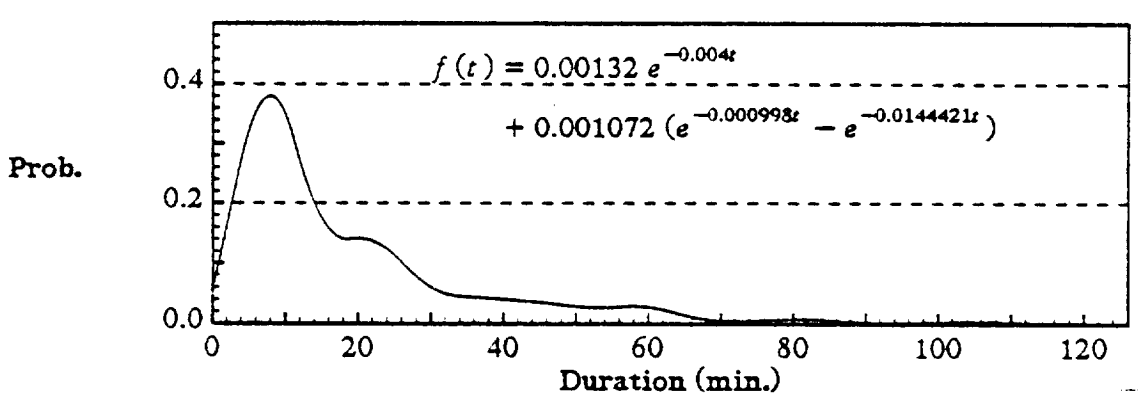
Waiting/holding time densities for W_7



Holding time density to MULT

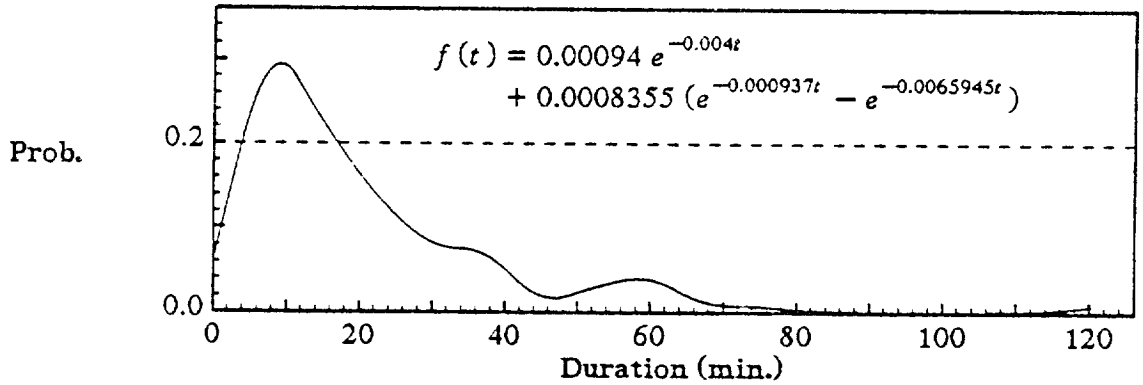


Holding time density to SWE

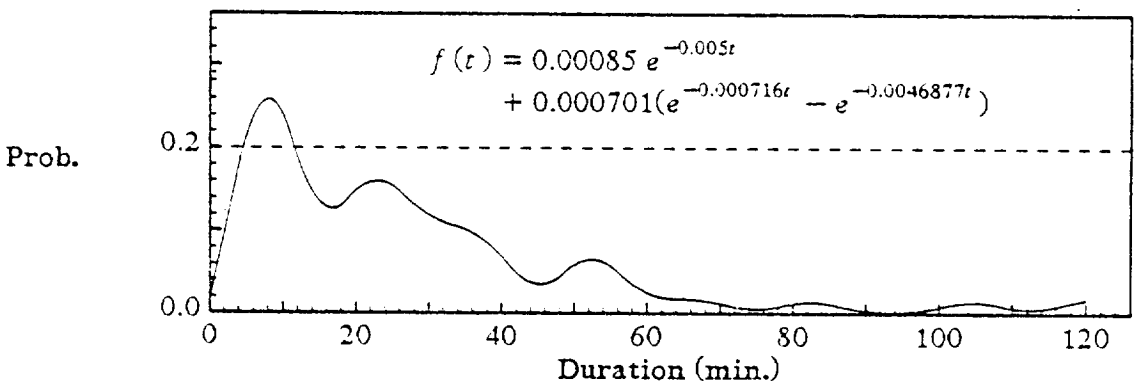


Holding time density to the others

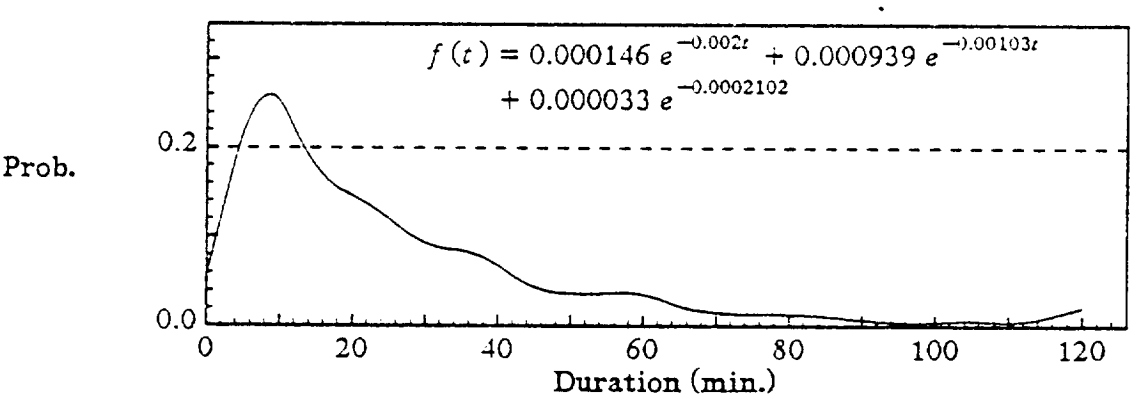
Waiting/holding time densities for W_8



Holding time density to DASD

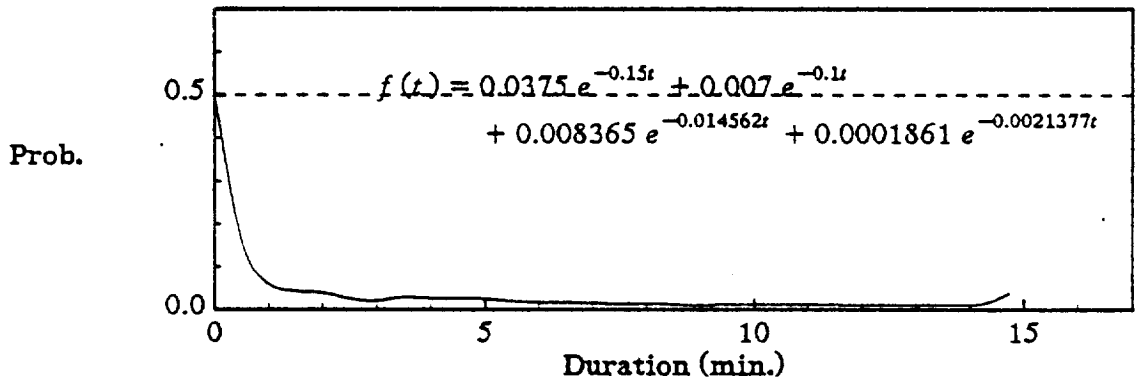


Holding time density to SWE

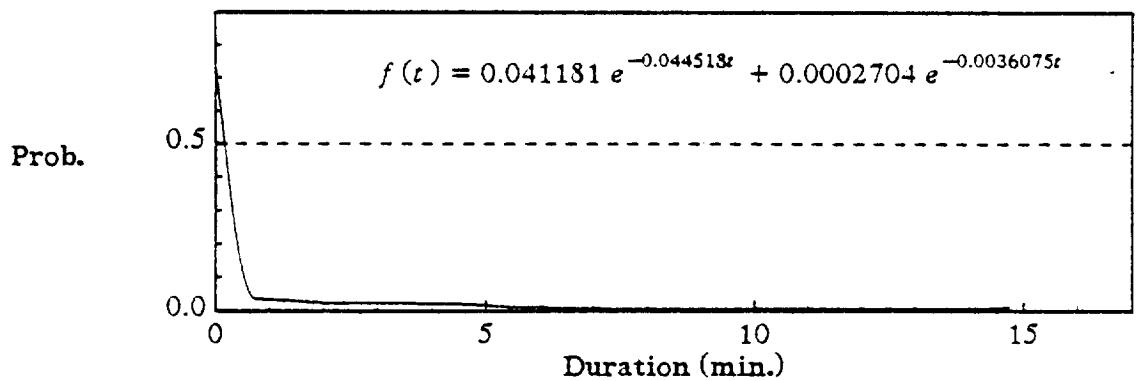


Holding time density to the others

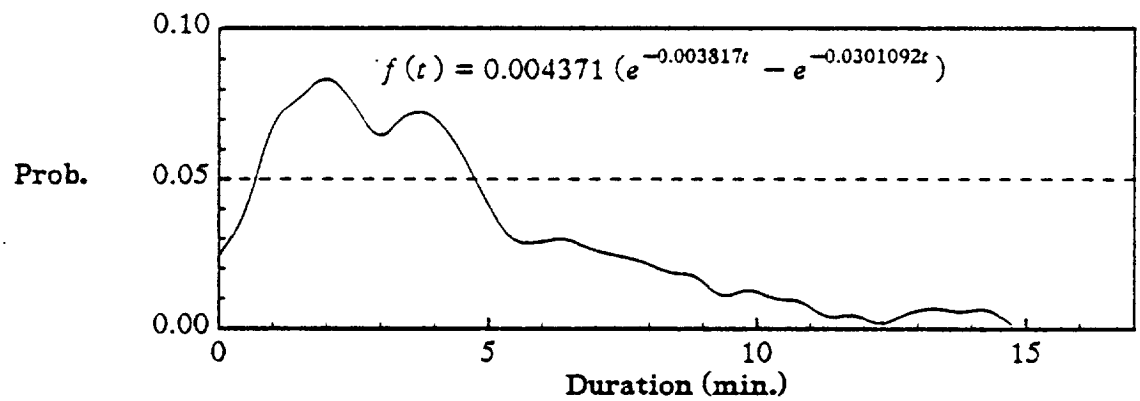
Waiting time densities for Error States



Waiting time density of DASD



Waiting time density of SWE



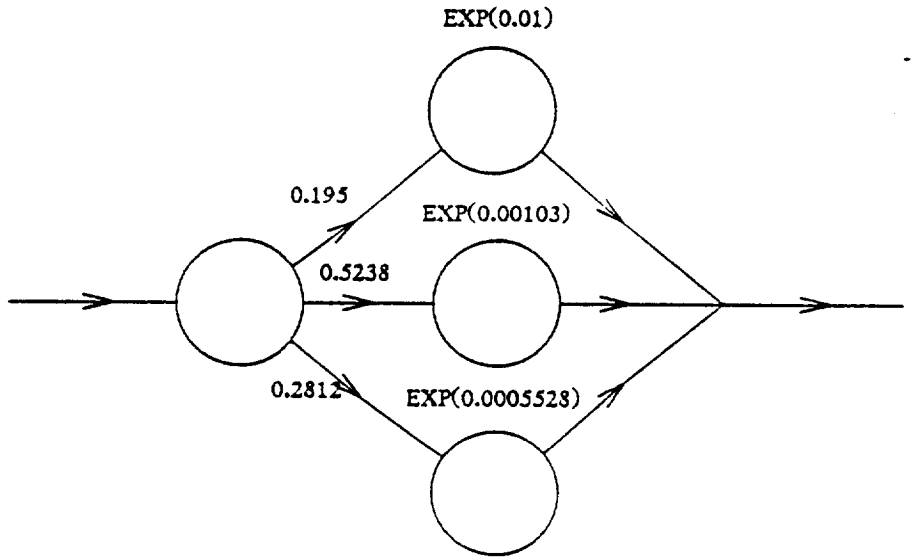
Waiting time density of MULT

APPENDIX C

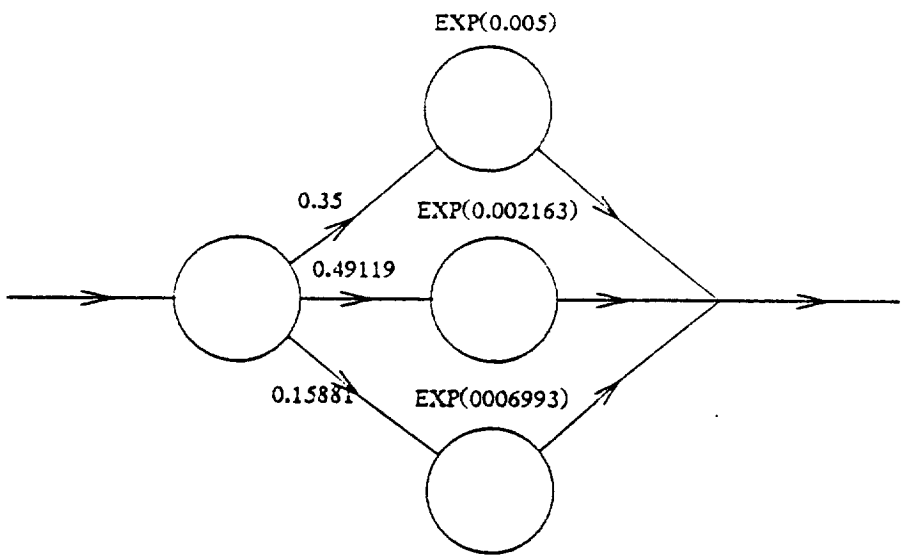
Semi-Markov to Markov Conversion

The state conversions of the resource-usage/error/recovery process from a semi-Markov model to Markov model is demonstrated in this appendix. Here, we assume that the model is an independent semi-Markov process because of the limitation of SHARPE.

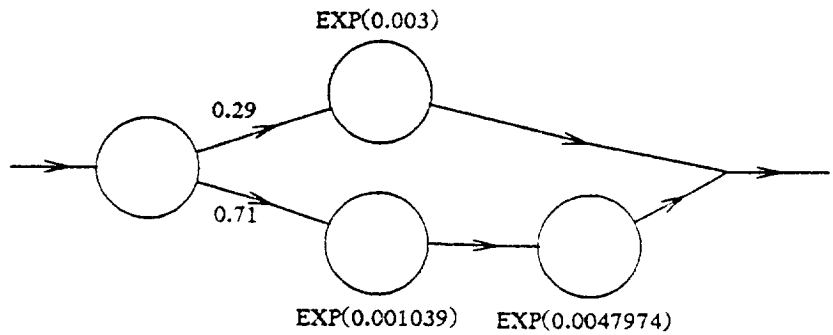
The CPU bound workload state is used to estimate the system's performability. State W_2 is combined with W_3 because W_2 has very few observations. CHAN error state is also ignored because it has very few observations. The semi-Markov to Markov conversion of the workload states are shown from Figure (a) through (g) and the conversion of three error states are shown from Figure (h) through (j). After the conversion, the overall model is expanded from 15 states to 34 states.



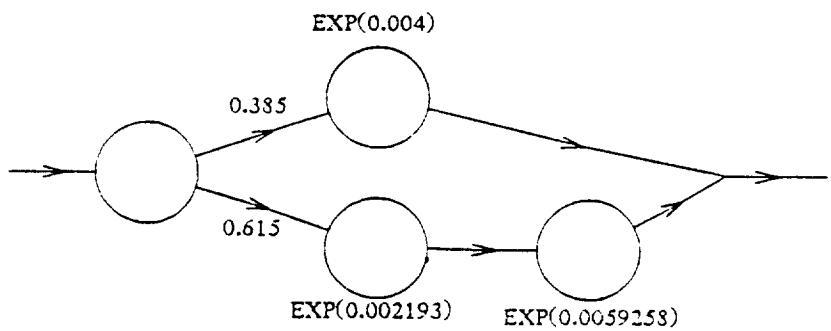
(a). W_1 state



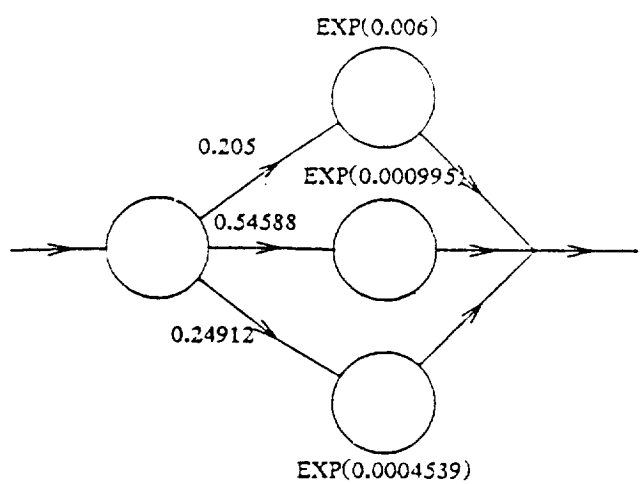
(b). W_3 state



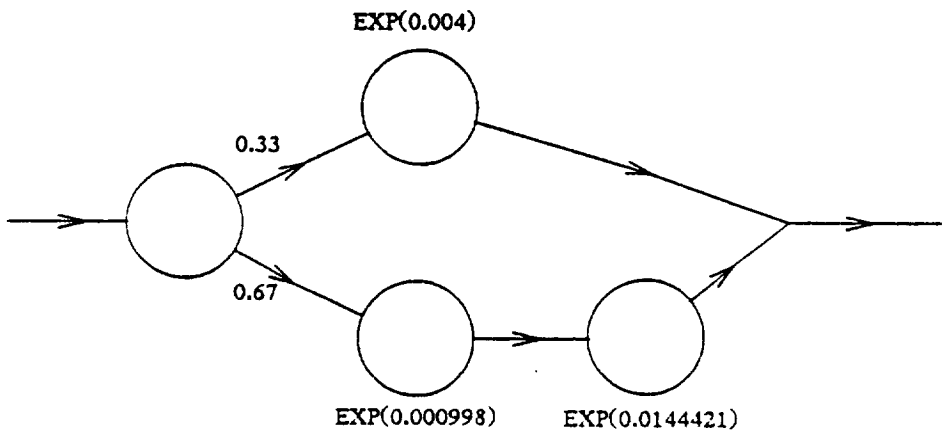
(c). W_4 state



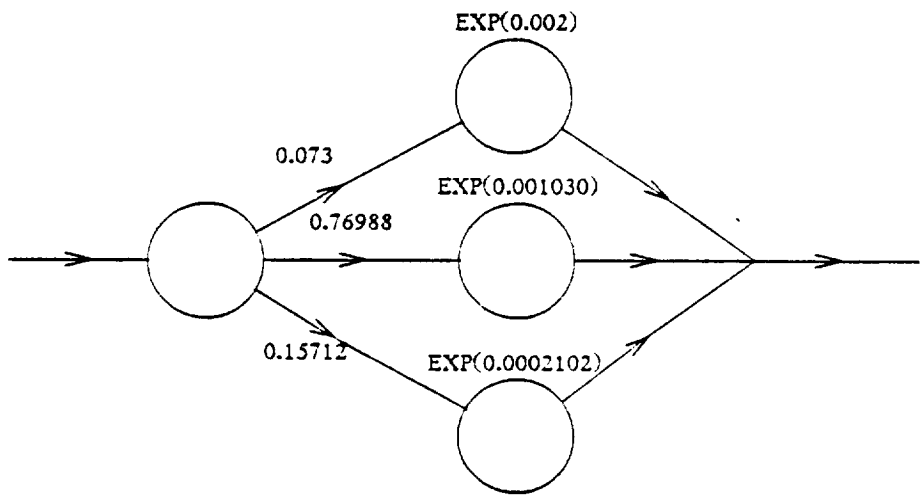
(d). W_5 state



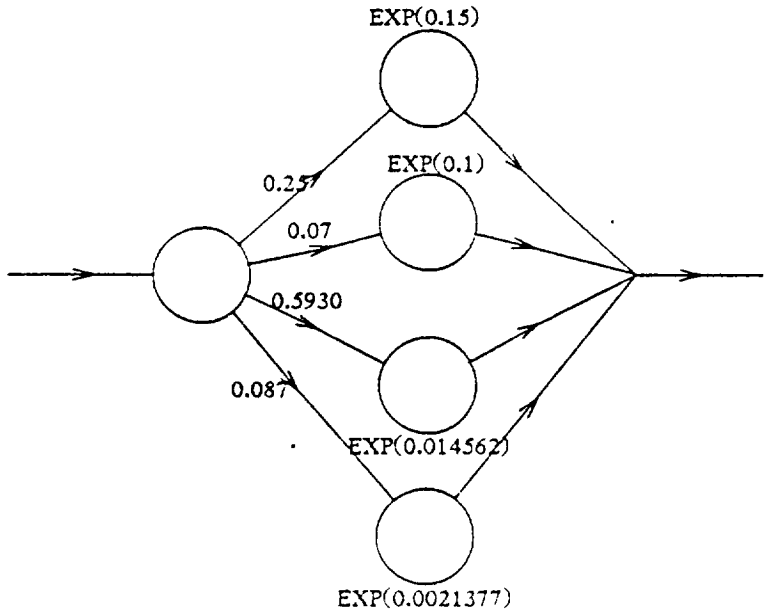
(e). W_6 state



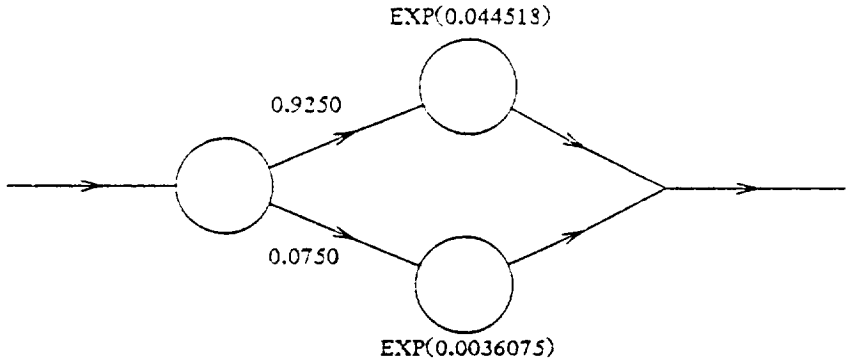
(f). W_7 state



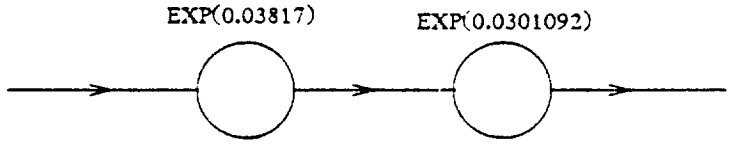
(g). W_8 state



(h). DASD state



(i). SWE state



(j). MULT state

VITA

Mei-Chen Hsueh was born on September 21, 1950 in Taiwan, Republic of China. She received her B.S. degree and M.S. degree both in Mathematics from Providence College, Taiwan, 1972 and Portland State University, Portland, Oregon, 1981, respectively. She obtained her Ph.D. degree in Computer Science from the University of Illinois at Urbana-Champaign, 1987.

She joined the Department of Industrial Engineering, Fen-Chia University as a full time teaching assistant from 1972 to 1974 and worked as a quality assurance engineer for Ampex Corporation, Taiwan, from 1975 to 1979. While attending Portland State University she worked as a teaching assistant for the Department of Mathematics and as a programming consultant for the Computer Center. After graduation in 1981, she joined Floating Point Systems, Inc. as a software engineer and left as a senior software engineer in 1983. As a graduate student in the University of Illinois, she was a research assistant in both the Department of Political Science (1983-1984) and the Computer Systems Group, Coordinated Science Laboratory (1984-1987).

Proceedings, Computer Software and Applications Conference, Tokyo, Japan, October 1987 (to appear)

N94-76427
91927
p. 7

DESIGN AND EVALUATION OF EXECUTABLE ASSERTIONS FOR CONCURRENT ERROR DETECTION

Kien A. Hua

IBM Corporation
Neighborhood Road
Kingston, NY 12401

Jacob A. Abraham

Computer Systems Group
Coordinated Science Laboratory
University of Illinois
Urbana, IL 61801

ABSTRACT

System level concurrent detection of errors due to hardware faults can be accomplished by introducing software redundancy in the form of executable assertions into a program to monitor the correct operation of the system during its execution. This paper attempts to formalize the use of executable assertions for the purpose of concurrent error detection, and discusses a transformation approach to the design of assertion statements.

The second part of the paper deals with the effectiveness of the error detection technique. A fault simulation method for the evaluation of the reliability of the assertion statements is described. This scheme was used to study the effectiveness of a nontrivial program in providing protection against faults in the underlying system. The experimental results shows that the use of executable assertions provides an effective approach to achieve concurrent error detection.

Keywords: Concurrent error detection, executable assertion, transformation, fault simulation, program design.

1. INTRODUCTION

Off-line circuit testing is one of the widely used techniques to detect physical failures and to ensure that a system is defect-free [1]. Unfortunately, shrinking device dimensions, lower energy levels for indicating logical values and higher speed operation have increased the possibility of transient errors in computing systems that are based on VLSI technology [2]. Since these errors are usually nonrecurring and not reproducible, off-line testing (useful for permanent faults) will not reliably detect transients. The current trend is to include *Concurrent Error Detection* (CED) capability in the design of digital systems in order to detect errors concurrently during normal operation.

Traditionally, systems with CED are implemented using self-checking circuits [3] and/or hardware duplication [4]. A self-checking circuit is a logic circuit designed to process information encoded in error-detecting codes. Through the use of proper design techniques, the circuit can be designed so that most faults will produce errors that are detectable by observing the circuit's encoded output. Duplication circuits can be considered as a special case of self-checking circuits that employ the duplication code. Since these techniques usually require a large degree of hardware redundancy, they are very expensive.

Recently, a new approach that utilizes a combination of hardware and software redundancy to accomplish concurrent error detection has been proposed [5, 6]. These techniques assume that malfunctions such as processor failures and bus faults will cause errors in the control flow of the program. These techniques partition the assembly language level instructions of the application program into blocks without branch instructions. At compile time, cyclic codes or signatures are generated for the instruction stream of each block. During the execution, the same cyclic encoded signatures are regenerated by a linear feedback shift register. Error detection is performed by comparing the run-time generated signatures with the ones precomputed by the compiler. A similar technique that employs only software redundancy is discussed in [7]. This technique partitions the program into loop-free intervals and a database containing the path information in each of the loop-free intervals is derived from the detailed design. Redundant code is inserted in the program so that the path actually traversed at run time can be recoreded and then checked against the information provided in the database. Any discrepancy indicates an error. These techniques are inexpensive and provide excellent control flow monitoring capability. However, they do not perform well in cases of errors due to data type faults [8].

This paper discusses another software approach to achieve system level CED. This technique inserts software redundancy in the form of executable assertions into the program to check the correct operation of the system during its execution. Since the assertions are designed to monitor the intended behavior of the program, control sequence errors or errors due to faults in the data values will change the intended semantics of the program and therefore will be detected by the assertion statements at run time.

An assertion-checking statement is a statement of the form:

if not ASSERTION *then* ERROR;

where ASSERTION is a Boolean expression that specifies conditions about the program variables that the current state of the computation has to satisfy, and ERROR is the action taken if the ASSERTION is FALSE (i.e., the conditions are not satisfied). Since the subject of this paper is error detection (rather than correction), and we simply consider ERROR as an output statement that sends an error message. In order to achieve fault tolerance, ERROR can be a call to some error recovery routine. This,

Acknowledgment: This research was supported in part by a contract from IBM Corporation and in part by the National Aeronautics and Space Administration (NASA) under contract NASA NAG 1-613.

however, is beyond the scope of this discussion; for such a discussion the reader is referred to [9] and [10].

The use of executable assertions for CED has been discussed in [11, 12, 13], and found to be very helpful. However, assertions when used carelessly may provide only a poor coverage of errors. In this paper, we attempt to formalize the use of assertions for the purpose of CED. A structured technique for the design of self-checking programs is presented in [14], in which the assertions are derived from the functional specifications of the program by performing stepwise refinement. The process of stepwise refinement, however, relies on intuition and is therefore very much ad-hoc. In Section 2, we will present a transformation approach to the design of programs with assertion statements; an example will also be given to illustrate the application of this technique. In Section 3 the technique is extended to handle procedure and function calls.

Section 4 is concerned with the effectiveness of run-time assertions in providing protection against faults in the underlying system, (i.e., against hardware and transient faults). Although executable assertions were originally proposed for the detection of errors caused by software faults, it has long been suspected that they can also detect errors caused by faults in the hardware. We conducted experiments to investigate this issue of assertion statements. In this section, we will describe the experiments and discuss the experimental results. They show that use of the executable assertions provides an effective approach to achieve CED.

2. DESIGN ISSUES

In this paper we use a program design language similar to that described in [15] to discuss the assertion design technique. The syntax description for the language is informally given in the following:

```
<statement list>::= <statement>+
<statement>::= <assignment> | <alternative> | <iterative>
               | <call>
<assignment>::= <var> ← <English statement>
<alternative>::= if  <Boolean expression>
                     then  <statement list>
                     else  <statement list> fi;
<iterative>::= while  <Boolean expression> do
                      <statement list>
               enddo;
```

Here, <call> is a procedure or function call whose syntax is the same as subroutine calls in Pascal. "(...)+" denotes "one or more instances of the enclosed." An example design written in this program design language is given in the following. The program computes the inverse of a matrix M and returns the inverted matrix in MI.

```
procedure MATRIXINV(n: integer;  (* dimension of M *)
        M: array [1..10,1..10] of real;  (* matrix to invert *)
        var MI: array [1..10,1..10] of real);  (* inverted matrix *)
var
    i: integer;  (* loop index *)
    B,X: array [1..10] of real;
begin
1   i←1;
    while i ≤ n do
2       B←i-th column of the identity matrix, I[i-th column];
3       MI[i-th column]←compute X such that MX = B;
4       i←increment i;
    enddo
end;  (* MATRIXINV *)
```

The properties of being intuitive and being rigorous often seem in conflict. A program design language that is too rigorous is usually too tedious to use, and it is error-prone. Program design languages, such as the one informally described in this paper, offer several advantages.

(1) The design is written in terms of the concepts and functions of the discipline from which the problem is taken. It is therefore very readable and easy to comprehend.
(2) The use of "pidgin" English statements allows the ideas to be expressed at whatever level of detail is appropriate to avoid ambiguity.
(3) Although English-like statements are used, formalism is imposed on the control constructs so that the correctness of the design can be formally verified using traditional verification techniques [16].
(4) The use of the structured controls makes the translation of the design into code in structured programming languages (e.g., Pascal, PL/I) a straightforward process.

In the following, we will use the term "program design" to denote the design of the program written in the program design language, and use the term "program code" to designate the final product written in the target programming language.

In the following, we assume that the design of a program has been successfully written in the program design language. We will present a constructive technique that transforms the design into a new design with embedded assertion statements. The following definitions are needed for the discussion.

Definition 1: Let SL be a statement list in a program design. *ASSERT(SL)* denotes the assertion statements (knowledge) of what must be true after the "execution" of the SL.

Definition 2: Let A_1 and A_2 be assertion statements. $A_1 \circ A_2$ denotes the new assertion statement that expresses the combined knowledge of what must be true derivable from A_1 and A_2.

Note that since the alternation of the order of the program statements may not preserve the semantics of the program, the operation $\circ$ is not commutative. For instance, suppose we have the following assertions:

$$A_1 \equiv (X=a)$$
$$A_2 \equiv (X=X_o+2)$$
$$A_3 \equiv (Y=X+2)$$

where X_o denotes the previous value of the variable X. We then have

$$A_1 \circ A_2 \circ A_3 \equiv (X=a+2 \wedge Y=a+4)$$
$$A_1 \circ A_3 \circ A_2 \equiv (X=Y=a+2)$$

An assertion statement should have the form "*if not* ASSERTION *then* ERROR." However, we will use the ASSERTION to denote the assertion statement whenever it is clear from the surrounding context.

Definition 3: Let SL be a statement list in a program design. *ASSERTED(SL)* denotes the transformed version of the SL - the new SL with associated assertions inserted, (e.g., ASSERTED(SL) = (SL ASSERT(SL))).

In order to be able to transform a program design into a design with embedded assertion statements, we need a set of rules that transforms the control constructs into versions with the associated assertions. In the following we will use the symbols p to denote a Boolean expression, k to denote an expression, SLi to denote statement list i

and α_i to denote constant i. A set of heuristic rules based on function expressions discussed in [17] is given below:

(1) Rule of Decomposition:
ASSERTED(SL1 SL2) = (ASSERTED(SL1) ASSERTED(SL2))

(2) Rule of Composition (Sequential):
ASSERTED(SL1 SL2)
= (SL1 SL2 ASSERT(SL1) ∘ ASSERT(SL2))

(3) Rule of Alternation:
ASSERTED(*if* p *then* SL1 *else* SL2 *fi*;)
= (*if* p *then* ASSERT(p) ASSERTED(SL1)
else ASSERT(¬p) ASSERTED(SL2) *fi*;)

(4) Rule of Iteration:
ASSERTED(*while* p *do* SL *enddo*;)
= (*while* p *do* ASSERT(p) ASSERTED(SL) *enddo*;
ASSERT(¬p))

Assertions, designed using these rules, check whether the run-time computation computes the intended semantics defined by the program design. We note that assertions for a complete check can be expensive for some programs, and we may have to use less rigorous assertions that check on the acceptability of the computed results rather than guarantee their complete correctness. In the following we give an example to illustrate the application of the transformation rules to design both a complete check and a more cost-effective check for a program design.

Example 1: We will transform the procedure MATRIXINV given earlier into a design with assertion statements. In this example we will refer to the statements by their statement numbers as given in the program design. Also i_0 denotes the previous value of the variable i.

ASSERTED(MATRIXINV)
= ASSERTED(1 *while* i ≤ n *do* 2 3 4 *enddo*;)

Apply the Rule of Decomposition, the program can be divided into two segments:

ASSERTED(MATRIXINV)
= (ASSERTED(1) ASSERTED(*while* i ≤ n *do* 2 3 4 *enddo*;)) (1)

The transformed version of statement 1 is

ASSERTED(1) = (1 ASSERT(1))
= (1 *if* i ≠ 1 *then* ERROR;) (2)

Applying the Rule of Iteration, we have

ASSERTED(*while* i ≤ n *do* 2 3 4 *enddo*;)
= (*while* i ≤ n *do* ASSERT(i ≤ n) ASSERTED(2 3 4) *enddo*;
ASSERT(i > n)) (3)

Applying the Rule of Composition, we have

ASSERTED(2 3 4)
= (2 3 4 ASSERT(2) ∘ ASSERT(3) ∘ ASSERT(4))
= (2 3 4 (B = I[i-th column]) ∘ (M · MI[i-th column] = B)
∘ (i = i_0+1))
= (2 3 4 (M · MI[i-th column] = I[i-th column]) ∘ (i = i_0+1))
= (2 3 4 (M · MI[i-th column] = I[i-th column]) ∧ (i = i_0+1)) (4)

Substitute (4) into (3), and we have

ASSERTED(*while* i ≤ n *do* 2 3 4 *enddo*;)
= (*while* i ≤ n *do*
if i > n *then* ERROR
i_0 ← i; (* save previous value *)
2 3 4
if (M · MI[i-th column] ≠ I[i-th column]) *then* ERROR;
if (i ≠ i_0+1) *then* ERROR;
enddo
if (i ≤ n) *then* ERROR;) (5)

Substitute (2) and (5) into (1), and we have

```
ASSERTED(MATRIXINV)
  = ( i ← 1;
    if i ≠ 1 then ERROR;
    while i ≤ n do
        if i > n then ERROR
        i0 ← i;   (* save previous value *)
        B ← I[i-th column];
        MI[i-th column] ← compute X such that MX = B;
        i ← increment i;
        if (M · MI[i-th column] ≠ I[i-th column]) then ERROR;
        if (i ≠ i0+1) then ERROR;
      enddo
      if (i ≤ n) then ERROR; )
```

The assertions in (5) guarantee that after the execution of the WHILE..DO construct, we must have

$$\forall i \leq n,\ M \cdot MI[\text{i-th column}] = I[\text{i-th column}]$$

However, the assertion ASSERT(1) ensures that i was initialized to "1" before entering the loop. We therefore have

$$\forall i\ 1 \leq i \leq n,\ M \cdot MI[\text{i-th column}] = I[\text{i-th column}]$$

In other words, the assertions ensure that the computed result satisfies the condition M · MI = I, or errors would have been detected. These assertions provide a "complete check" for the program. To improve the run-time overhead, one may want to use the following less rigorous assertion for statement 3

ASSERT(3) = (M[i-th row] · MI[i-th column] = B[i])

We then have

ASSERT(2) ∘ ASSERT(3) ∘ ASSERT(4)
= (M[i_row] · MI[i_column] = [1]) ∧ (i = i_0 + 1)

and the final design becomes

```
ASSERTED(MATRIXINV)
  = ( i ← 1;
    if i ≠ 1 then ERROR;
    while i ≤ n do
        if i > n then ERROR
        i0 ← i;   (* save previous value *)
        B ← I[i-th column];
        MI[i-th column] ← compute X such that MX = B;
        i ← increment i;
        if (M[i_row] · MI[i-th column] ≠ [1]) then ERROR;
        if (i ≠ i0-1) then ERROR;
      enddo
      if (i ≤ n) then ERROR; )
```

We see that this approach can reduce the run-time overhead by about n (number of rows in the matrix M) times compared to the design with a complete check. Although its assertions do not perform a complete check, every element of the output matrix MI is involved in the checking process. These assertions can therefore provide powerful error detection at a much less cost. Another form of less rigorous checks using data encoding schemes is discussed in [18], in which a program is designed to process information encoded in some error detecting code and error detection is performed by observing the encoded output.

The transformation process described in Example 1 may seem to involve many steps. Actually, many of them were shown for the purpose of illustration. In practice, the obvious steps could have been skipped, and the process would have appeared much shorter.

The transformation rules described earlier can also be applied to programs at the target programming language

level. Computer programs, however, are extremely low-level. They seldom involve concepts related to the problem the program is trying to solve, but instead deal with strictly computer-oriented operations such as addition, subtraction, etc. Deriving assertions from the program code therefore can be tedious and error-prone. In contrast, program designs are written in terms of the concepts and functions of the discipline from which the problem is taken. Such a program would be little more than a restatement of the mathematical or other formulation of the problem, and derivation of assertions based on the program design would be correspondingly general and would not depend on the implementation details. Since this approach would employ the concepts and notations of the problem discipline, fundamental results from the problem discipline can also be used in designing the assertions. The preceding example demonstrates that deriving assertions from the program design is much simpler than deriving them from the program code. In the example, we dealt with only four design statements. In Pascal or Fortran for instance, we would have to deal with about a hundred of lines of code.

Another advantage of deriving assertions from the program design instead of the program code lies in the detection of errors due to coding faults and erroneous designs in the lower levels of the program. During the development of software, executable assertions can provide an extensive testing capability to locate software errors [9]. If the assertions are derived from the program design, it can be used to check whether the program code correctly implements the program design.

3. HANDLING PROCEDURE CALLS

A design may involve procedure and function calls. In order to show the generality of the transformation technique, it is extended to handle procedure and function calls. There are two approaches to these statements:

(1) We can consider a procedure or a function as a separate program and apply the transformation rules to transform it into a design with assertion statements. Assertions can also be inserted to check the correct passing of the arguments. For instance, we can pass to the called procedure a check variable which is the sum of the actual parameters. In the called procedure, we add assertions to verify that the check variable is indeed the checksum of the formal parameters. This scheme can protect against hardware and transient faults that affect the argument stack during the procedure calls.

(2) Viewing a procedure call as an instance of the called procedure, we can consider the called procedure as if it is part of the calling procedure. The transformation then can be carried out normally after the following "modification" - the following steps merely describes a mental process; non of the assignment statements should actually be added, or any call statements be removed from the original program:

(a) At the beginning of the called procedure, we add assignment statements to assign the values of the actual parameters to the corresponding formal parameters.
(b) At the end of the called procedure, we add assignment statements to assign the values of those formal parameters that are "called by reference" to the corresponding actual parameters.
(c) Replace the procedure call statement by the new body of the called procedure.

For function calls, the rules are similar. They are given in the following:

(a) At the beginning of the called function, we add assignment statements to assign the values of the actual parameters to the corresponding formal parameters.
(b) Add before the statement that involves the function call, the new body of the called function.
(c) Replace the function call by the function name, (i.e., the function name is used as a variable).

Example 2: This example illustrates these two approaches. The procedure MATRIXINV shown in Example 1 can be rewritten to include a call to procedure SOLVE that computes the solution of a linear system MX = B. The solution is returned in the matrix X.

```
procedure MATRIXINV(n: integer;  (* dimension of M *)
    M: array [1..10,1..10] of real;  (* matrix to invert *)
    var MI: array [1..10,1..10] of real);  (* inverted matrix *)
var
    i: integer;  (* loop index *)
    B,X: array [1..10] of real;
begin
    i←1;
    while i ≤ n do
        B←i-th column of the identity matrix, I[i-th column];
        call SOLVE(M,X,B);
        MI[i-th column]← X;
        i←increment i;
    enddo
end;  (* MATRIXINV *)
```

If the first approach is used, then the transformed version of MATRIXINV is as follows:

```
procedure MATRIXINV(n: integer;  (* dimension of M *)
    M: array [1..10,1..10] of real;  (* matrix to invert *)
    var MI: array [1..10,1..10] of real);  (* inverted matrix *)
var
    i: integer;  (* loop index *)
    B,X: array [1..10] of real;
begin
    i←1;
    if (i ≠ 1) then ERROR;
    while i ≤ n do
        if i>n then ERROR;
        i0 ← i;  (* save previous value *)
        B←i-th column of the identity matrix, I[i-th column];
        call SOLVE(M,X,B);
        MI[i-th column]← X;
        i←increment i;
        if (MI[i-th column] ≠ X) then ERROR;
        if (i ≠ i0+1) then ERROR;
    enddo
    if (i ≤ n) then ERROR;
end;  (* MATRIXINV *)
```

Note that the correct operation of the procedure SOLVE is assumed by the calling procedure. MATRIXINV assumes that the solution returned in X has been checked to be correct by SOLVE, and it only checks to make sure that X is assigned to the appropriate column of the matrix MI. In other words, it assumes the responsibility of the procedure SOLVE to include assertions to assure that X satisfies the condition MX = B. If the second approach were used, the assertion statements for MATRIXINV would have been the same as those shown in Example 1. Consider the procedure SOLVE as having only one statement:

X ← compute X such that MX = B;

it is then obvious why the second approach would have yielded the same assertion statements as those derived in Example 1.

4. EXPERIMENTAL STUDY

4.1. Methodology

To study the effectiveness of assertion statements, we must be able to insert hardware faults into the system to see their effects on the execution of the programs. Simulating faults in hardware, however, is very expensive. A less expensive approach that simulates hardware failures in software is described in [18]. The central idea of this technique is to introduce simulated faults by applying mutation transformations [18] to produce mutant programs. These mutants are then executed to measure the ability of the set of assertions to distinguish the program from its mutants. This technique employs an error-based testing strategy. It requires the definition of the classes of faults that are considered. The set of faults that constitutes the fault model must correctly characterize possible physical failures in the hardware system. Our fault model is based on the earlier functional fault model for complex processors [19]. We extended the existing fault model to include faults in the memory system. The fault model is given in the following:

(A) **Addressing faults:** Faulty decoding circuits can cause the following faults:
- no storage location is selected
- a wrong storage location is selected
- more than one storage location is selected.

(B) **Instruction decoding faults:** Failures in instruction decoding can cause the following faults:
- no instruction is executed
- a wrong instruction is executed
- more than one instruction is executed.

(C) **Faults in storage elements:** Some of the bits in the storage elements are stuck-at-0 or stuck-at-1. Transient failures can also change the contents of registers or memory locations arbitrarily.

Since a hardware fault can affect many statements in the assembly code, the insertion of a simulated hardware fault into the system can be very tedious. For instance, a stuck-at fault in a register will affect all the assembly instructions that involve the faulty register. To produce the mutant program, one must add additional instructions throughout the assembly code to set (stuck-at 1) or reset (stuck-at 0) the stuck-at bit properly. The assembly code is usually very long and this task is therefore very tedious and error-prone. In order to be able to insert hardware faults quickly and reliably, we implemented a program that can perform this task automatically. Basically, for each test case we specify a mutant operator; in response, the fault insertion program asks for relevant information, and then performs the mutation transformation automatically by scanning the assembly code and modifying the affected instructions accordingly.

Another class of faults that must be considered is transient faults. Transient faults are due to temporary environmental conditions. They can occur even in the absence of all physical defects. To insert these faults, we used the debugger ADB [20] available on the VAX. ADB is a general purpose debugging program. It provides capabilities to examine files and a control environment for the execution of UNIX programs. We ran the assembly code of the program under the control of ADB, and transient faults were injected by setting break points and modifying the object files.

4.2. Experiments

The experimental results reported in [18] show that the use of run-time assertions yielded a substantial improvement in system reliability. The programs selected for the experiments, however, were small. We are also interested in assessing the reliability and the cost effectiveness of this form of CED when it is used in larger programs. In this section we describe another experiment based on a nontrivial program, namely the Redundant Strapped Down Inertial Measurement Unit (RSDIMU). This experiment will help us understand better the effectiveness of the CED technique when it is used in real applications.

The RSDIMU is part of a navigation system in aircraft and spacecraft [21]. We selected the RSDIMU for this study for two reasons. First, it is the kind of application that would require very high system reliability. Secondly, the problem has been used in a study of N-version programming that was carried out by the Research Triangle Institute for the National Aeronautics and Space Administration. We were able to use a version of the program for our project.

The program is written in Pascal. We added executable assertions to the original program to provide the CED capability. Both the transformation approach described in this paper and the data encoding technique discussed in [18] were used to design the assertion statements. The software redundancy due to the assertion statements is about 14%, and the run-time overhead is 9.8%.

In this study, we assume that assertions that distinguish mutant programs differing from a correct one by only simple errors are so sensitive that they also implicitly detect more complex errors. In other words, complex errors are coupled to simple errors. The coupling effect has also been used in mutation testing in software [22, 23, 24]. Based on this assumption, we included only single faults in our experiments. All three types of faults described in the fault model were considered in our study. We tried 539 cases of hardware faults and about 600 cases of transient faults. Every register, most memory locations used for data storage, and essentially every type of VAX instruction used by the program were subjected to inserted faults during some phase of the experiment. The experimental results are summarized in Table 1 that gives the types of faults and the resulting errors detected during the experiments.

We found that essentially all errors that affected the data manipulation were detected. Undetected errors were due to permanent hardware faults that caused infinite execution. A fault may cause an unintended loop to execute indefinitely [25], and if there is no assertion in the loop then the errors will not be detected. Inexpensive time-out mechanisms [26] can be used to handle this problem.

There were a few cases where the errors were detected by the operating system, for example, errors which caused the program to try to read pass end of file, floating exceptions, when bad data was found on real read, memory faults, illegal instructions, bus error, etc., before the program had a chance to check its output. These were counted as detected in the experiments. In particular, permanent s-a-1 faults, especially those at the more

significant bits, tend to cause floating exception. Errors due to this class of faults are generally detected by the operating system.

If a transient or software fault affects only the input operations, it effectively changes the input data, and the assertion statements cannot recognize valid (satisfies the input specifications) but faulty (unintended) data, unless they were originally stored in some encoded format. Nor can the technique detect errors caused by faults that affect the output only after the correctness of the output has been verified by the assertions. These classes of faults are beyond the scope of the CED technique and are not included in our study. In fact, many faults that occur during the I/O operations are likely to cause memory faults, bus error, etc., and will be detected by most operating systems. We also did not consider those faults (mostly transient, less than 5% of the test cases) that have no effect on the outcomes of the computations because the proposed technique is intended for error detection. For instance, a faulty decoding circuit may cause a wrong register to be selected every time some intended register is used. If the wrong register is not designed to be used anywhere in the program segment, then the wrong register can replace the intended one without causing any error in the computation of the program segment.

Although it is not our primary concern, it is interesting to note that the CED technique located two software bugs committed by the programmers in the original program. These software faults were overlooked by the acceptance test (used to test the system to the requirements) done by the Research Triangle Institute. The program was run on 1,000 sets of test data, and the two bugs were not discovered.

6. CONCLUSIONS

This paper has presented a programming technique that utilizes software redundancy to accomplish concurrent error detection. We have formalized the use of executable assertions for the purpose of concurrent error detection, and presented a transformation approach to the design of assertion statements.

The second part of the paper dealt with the reliability of the concurrent error detection technique. A fault simulation method has been developed for the evaluation of the reliability of assertion statements. This scheme was used in our experiments to study the effectiveness of a nontrivial program. The experiments showed that errors due to most inserted faults were detectable.

The CED technique described in this paper depends for its reliability improvement on the assumption that errors occurring during the execution of the assertion statements will not mask out the errors in the normal computation. This is a reasonable assumption. An assertion statement usually has the following form:

if expression1 $\neq$ expression2 *then* ERROR

It is unlikely that errors occurred during the execution of expression1 and/or expression2, yet their results are accidentally equal (except for faults in the comparator). Our experimental results seem to support the validity of this assumption.

A disadvantage of using executable assertions for concurrent error detection is the extra time required to execute the assertion statements. A technique that uses a watchdog processor to execute the assertion statements concurrently with the main process in the main processor is discussed in [13]. This approach can improve the performance of such programs. Another disadvantage is the extra burden imposed on the software developer in designing the assertion statements. However it seems likely that the advantages of requiring checks, which are realized not only at run-time but also during the construction of the program, far outweigh any disadvantages [9].

We must point out that the technique discussed in this paper is meant to complement rather than replace other forms of concurrent error detection. This technique has its limitations. It cannot detect errors that occur after the validation has been done. For instance, once the results have been verified, errors in the output process are not detectable. A stuck-at fault in the status bit may cause an assertion to be TRUE at all time, and thus effectively disables the error detection capability of the assertion statement. Lower level error detection schemes such as

Table 1. Experimental Results

Type of Faults		Errors Detected by CED	Errors Detected by O. S.	Undetected Errors	Error Coverage	Error Coverage (with time out)
Hardware	Faults in Storage Element	90 (58.58%)	66 (41.25%)	4 (0.17%)	98.33%	100%
	Instruction Decoding Faults	22 (25.00%)	64 (72.73%)	2 (2.27%)		
	Addressing Faults	127 (43.64%)	161 (55.33%)	3 (1.03%)		
Transient	Control Faults	45 (42.86%)	60 (57.14%)	0 (0%)	100%	100%
	Instruction Decoding Faults	136 (62.67%)	81 (37.33%)	0 (0%)		
	Addressing Faults	109 (51.17%)	104 (48.83%)	0 (0%)		

self-checking circuits can be used to protect against such problems.

Finally, since removal of all faults prior to operational use of the software system cannot be guaranteed using software validation techniques, concurrent error detection of errors caused by software bugs is as important as detection of errors due to physical failures in the hardware. Although the technique discussed in this paper can be used to detect errors due to coding faults during the construction of the software, there is evidence that more errors occur during the requirements and design stages than during coding [27]. Run-time assertions have also been proposed for the detection of run-time errors due to design faults in the software. These techniques are discussed in [28, 29, 30]. As more and more critical safety systems becoming computerized, there is incentive to include such checking mechanisms in the software to detect run-time errors due to software faults.

In summary, It has been shown that a reliable system level concurrent detection of errors due to hardware faults can be accomplished by introducing executable assertions into the program to monitor the correct operation of the system. The design technique discussed in this paper provides a systematic approach to the design of such assertion statements. Since this approach require only software redundancy, it offer an inexpensive approach to achieve concurrent error detection. This technique, however, also has its limitations; it is desirable that we also include other forms of concurrent error detection to rectify the situation, particularly for safty critical applications.

REFERENCES

[1] M. A. Breuer and A. D. Friedman, *Diagnosis and Reliable Design of Digital Systems*. Maryland: Comput. Sci. Press, 1976.

[2] D. P. Siewiorek and R. S. Swarz, *The Theory and Practice of Reliable System Design*. Massachusetts: Digital Press, 1982.

[3] J. Wakerly, *Error Detecting Codes, Self-Checking Circuits and Applications*, New York: North-Holland, 1978

[4] D. Johnson, "The Intel 432: A VLSI Architecture for Fault-Tolerant Computer Systems," *IEEE Computer*, pp. 40-48, August 1984.

[5] T. Sridhar and S. M. Thatte, "Concurrent Checking of Program Flow in VLSI Processors," *Proc. Int'l Test Conf.*, pp. 191-199, 1982.

[6] J. P. Shen and M. A. Schuette, "On-Line Self-Monitoring Using Signatured Instruction Streams," *Proc. Int'l Test Conf.*, pp. 275-282, 1983.

[7] S. S. Yau and F-C Chen, "An Approach to Concurrent Control Flow Checking," *IEEE Trans. on Software Eng.*, vol. se-6, pp126-137, Mar. 1980.

[8] M. E. Schuette, J. P. Shen, D. P. Siewiorek and Y. X. Zhu, "Experimental Evaluation of Two Concurrent Error Detection Schemes," *Proc. 16th Int'l Symp. on Fault Tolerant Computing*, pp. 138-143, June 1986.

[9] T. Anderson and P. A. Lee, *Fault Tolerance: Principles and Practice*. New Jersey: Prentice-Hall, 1981.

[10] A. Mili, "Towards a Theory of Forward Error Recovery," *Trans. on Software Eng.*, vol. SE-11, no. 8, pp. 735-748, Aug. 1985.

[11] S. S. Yau and R. C. Cheung, "Design of Self-Checking Software," *Proc. Int'l Conf. on Reliability Software*, pp. 405-457, April 1975.

[12] D. M. Andrews, "Using Executable Assertions for Testing and Fault Tolerance," *Proc. 9th Int'l Symp. on Fault Tolerant Computing*, pp. 102-105, June 1979.

[13] A. Mahmood, E. J. McCluskey and D. J. Lu, "Concurrent Fault Detection Using A Watchdog Processor and Assertions," *Proc. Int'l Test Conf.*, pp. 622-628, 1983.

[14] A. Mili, *Self-Checking Programs: An Axiomatic Approach to The Validation of Programs by The Use of Assertions*, Ph.D. dissertation, Dept. Comp. Sci., University of Illinois, Urbana, 1981.

[15] S. H. Caine and E. K. Gordon, "PDL - A Tool for Software Design," *Proc. of the National Computer Conf.*, pp. 271-276, 1975.

[16] R. T. Yeh Ed., *Current Trends in Programming Methodology*, vol. II, Program Validation, New Jersey: Prentice-Hall, chs. 6-10, 1977.

[17] H. D. Mills, "The New Math of Computer Programming," *Comm. ACM*, vol. 18, no. 1, pp. 43-48, Jan. 1975.

[18] K. A. Hua and J. A. Abraham, "Design of Systems with Concurrent Error Detection Using Software Redundancy," *Proc. FJCC '86*, pp. 826-835, Nov. 1986.

[19] D. Brahme and J. A. Abraham, "Functional Testing of Microprocessors," *IEEE Trans. Comput.*, vol. C-33, pp. 475-485, June 1984.

[20] J. F. Maranzano and S. R. Bourne, "A Tutorial Introduction to ADB," *UNIX Programmer's Manual*, 7th Edition, vol. 2A, New Jersey: Bell Laboratories, Jan. 1979. vol. C-33, pp. 475-485, June 1984.

[21] Research Triangle Institute, *Redundancy Management Software Requirements Specification for A Redundant Strapped Down Inertial Measurement Unit*, Version 2.0, May 1985.

[22] R. A. DeMillo and R. J. Lipton, "Hints on Test Data Selection: Help for the Practicing Programmer," *IEEE Computer*, pp. 34-41, April 1978.

[23] T. A. Budd, R. J. Lipton, F. G. Sayward and R. A. DeMillo, "The Design of A Prototype Mutation System for Program Testing," *Proc. National Computer Conf.*, pp 623-627, 1978.

[24] W. E. Howden, "Weak Mutation Testing and Completeness of Test Sets," *IEEE Trans. Software Eng.*, vol. SE-8, no. 4, pp.371-379, July 1982.

[25] R. E. Glaser and G. M. Masson, "Transient Upsets in Microprocessor Controllers," *Proc. 11th Int'l Symp. on Fault Tolerant Computing*, pp. 165-167, June 1981.

[26] R. H. Campbell, K. H. Horton, and G. G. Belford, "Simulations of a Fault-Tolerant Deadline Mechanism," *Proc. 9th Int'l Symp. on Fault Tolerant Computing*, pp. 95-101, June 1979.

[27] M. Lipow, "Prediction of Software Errors," *J. Syst. Software*, vol. 1, pp. 71-75, 1979.

[28] E. Best and F. Cristian, "Systematic Detection of Exception Occurrences," *Science of Computing Programming 1*, Amsterdam: North-Holland, pp. 115-144, 1981.

[29] F. Cristian, "Correct and Robust Programs," *IEEE Trans. on Software Eng.*, vol. SE-10, no. 2, pp. 163-174, March 1984.

[30] N. G. Leveson and P. R. Harvey, "Analyzing Software Safety," *IEEE Trans. on Software Eng.*, vol. SE-9, no. 5, pp. 569-579, Sept. 1983.