

AD-A268 070



MENTATION PAGE

Form Approved
OMB No. 0704-0188

Estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering the necessary data, reviewing the collection of information, Send comments regarding this burden estimate or any other aspect of this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Avenue, Washington, DC 20540, and the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE 27 May 93	3. REPORT TYPE AND DATES COVERED THESIS/DISSERTATION											
4. TITLE AND SUBTITLE System Performance Modeling and Analysis of a Fault-Tolerant, Real Time Parallel Processor			5. FUNDING NUMBERS											
6. AUTHOR(S) Capt Robert J. Clasen														
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) AFIT Student Attending: Northeastern University			8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/CI/CIA- 93-112											
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) DEPARTMENT OF THE AIR FORCE AFIT/CI 2950 P STREET WRIGHT-PATTERSON AFB OH 45433-7765			10. SPONSORING/MONITORING AGENCY REPORT NUMBER											
11. SUPPLEMENTARY NOTES														
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for Public Release IAW 190-1 Distribution Unlimited MICHAEL M. BRICKER, SMSgt, USAF Chief Administration			<table border="1"><tr><td colspan="2">ACQUISITION CODE</td></tr><tr><td>NTIS CRA&I</td><td>☒</td></tr><tr><td>DTIC TAB</td><td>☐</td></tr><tr><td>Unannounced</td><td>☐</td></tr><tr><td>Justification</td><td></td></tr></table>		ACQUISITION CODE		NTIS CRA&I	☒	DTIC TAB	☐	Unannounced	☐	Justification	
ACQUISITION CODE														
NTIS CRA&I	☒													
DTIC TAB	☐													
Unannounced	☐													
Justification														
13. ABSTRACT (Maximum 200 words)			<table border="1"><tr><td colspan="2">By _____</td></tr><tr><td colspan="2">Distribution / _____</td></tr><tr><td colspan="2">Availability Codes</td></tr><tr><td>Dist</td><td>Avail and/or Special</td></tr><tr><td>A-1</td><td></td></tr></table>		By _____		Distribution / _____		Availability Codes		Dist	Avail and/or Special	A-1	
By _____														
Distribution / _____														
Availability Codes														
Dist	Avail and/or Special													
A-1														
14. SUBJECT TERMS			15. NUMBER OF PAGES 136											
			16. PRICE CODE											
17. SECURITY CLASSIFICATION OF REPORT			18. SECURITY CLASSIFICATION OF THIS PAGE											
19. SECURITY CLASSIFICATION OF ABSTRACT			20. LIMITATION OF ABSTRACT											

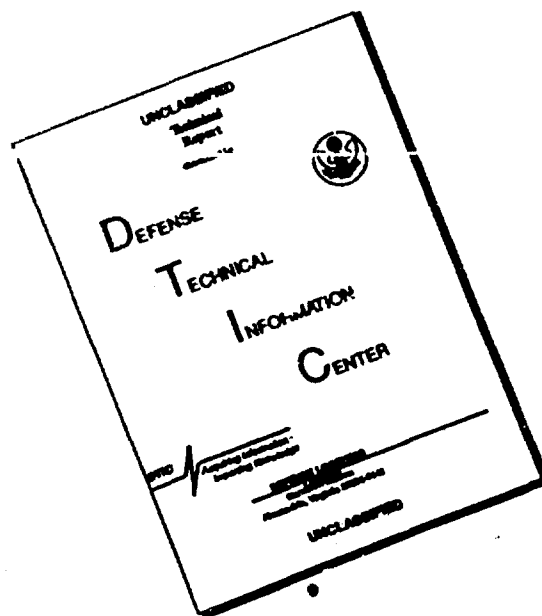
DTIC
ELECTE
AUG 17 1993
S E D

DTIC QUALITY INSPECTED 3

93-18974



DISCLAIMER NOTICE



**THIS DOCUMENT IS BEST
QUALITY AVAILABLE. THE COPY
FURNISHED TO DTIC CONTAINED
A SIGNIFICANT NUMBER OF
PAGES WHICH DO NOT
REPRODUCE LEGIBLY.**

System Performance Modeling and Analysis of a Fault-Tolerant, Real-Time Parallel Processor

by

Robert J. Clasen, Captain, USAF

Department of Electrical and Computer Engineering
Northeastern University, Boston, MA

Master of Science degree, 1993

Abstract

The use of fault-tolerant, real-time systems for the control of life-critical processes is becoming increasingly common, with examples including flight and nuclear reactor control systems. In such systems, the overhead associated with managing redundancy, communication, and task scheduling is critical due to real-time constraints imposed by the application; missed time deadlines can be viewed as system failures, with results as consequential as hardware failures.

The Fault-Tolerant Parallel Processor (FTPP) was developed by Draper Laboratory as a fault-tolerant, real-time computing platform. This thesis analyzes the FTPP prototype operating system overhead through the use of empirical performance measurement and two performance models based on these measurements. One model is developed to predict the operating system overhead under various configurations and workloads; accurate prediction of overhead provides confidence that real-time constraints can be satisfied. Because the system communication overhead may vary depending upon the amount of contention by Processing Elements for service by the Network Element, a second model is developed to account for performance delays that may result from this contention. When such performance analysis and modeling are an integral part of a concurrent build-analyze-improve methodology, performance bottlenecks can be cost-effectively removed at an early stage in development.

Empirical performance data show that the prototype FTPP Ada operating system (OS) overhead accounts for 22% of each 10 msec minor frame, excluding IO, for a given system configuration. The majority of the total overhead is due to communication overheads. The OS overhead model predicted the total overhead within an eight percent error. Simulation of the contention model showed that the most critical parameter in reducing the effect of contention is the time needed to transfer the packets from the local memory of the Processing Element to the Network Element.

Bibliography

- [Abl88] T. Abler, *A Network Element Based Fault Tolerant Processor*, MS Thesis, Massachusetts Institute of Technology, Cambridge, MA, May 1988.
- [Bab90] C. Babikyan, "The Fault Tolerant Parallel Processor Operating System Concepts and Performance Measurement Overview", *Proceedings of the 9th Digital Avionics Systems Conference*, October 1990, pp. 366-371.
- [Dol82] D. Dolev, "The Byzantine Generals Strike Again", *Journal of Algorithms*, 1982, pp. 14-30.
- [Dol84] D. Dolev, C. Dwork, and L. Stockmeyer, "On the Minimal Synchronism Needed for Distributed Consensus", IBM Research Report RJ 4294 (46990), 8 May 1984.
- [Fis82] M. Fischer and N. Lynch, "A Lower Bound for the Time to Assure Interactive Consistency", *Information Processing Letters*, 13 June 1982, pp. 183-186.
- [Har87] R. Harper, *Critical Issues in Ultra-Reliable Parallel Processing*, PhD Thesis, Massachusetts Institute of Technology, Cambridge, MA, June 1987.
- [Har88a] R. Harper, J. Lala, and J. Deyst, "Fault Tolerant Parallel Processor Overview", *18th International Symposium on Fault Tolerant Computing*, June 1988, pp. 252-257.
- [Har88b] R. Harper, "Reliability Analysis of Parallel Processing Systems", *Proceedings of the 8th Digital Avionics Systems Conference*, October 1988, pp. 213-219.
- [Har91] R. Harper and J. Lala, "Fault Tolerant Parallel Processor", *Journal of Guidance, Control, and Dynamics*, May-June 1991, pp. 554-563.
- [Joh89] B. Johnson, *Design and Analysis of Fault Tolerant Digital Systems*. Addison-Wesley, 1989.
- [Lam82] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem", *ACM Transactions on Programming Languages and Systems*, July 1982, pp. 382-401.
- [Leh89] T. Lehr, et. al., "Visualizing Performance Debugging", *Computer*, October 1989, pp. 38-51.
- [Pal85] D. Palumbo and R. Butler, "Measurement of SIFT Overhead", NASA Technical Memorandum 86722, Langley Research Center, Hampton, VA, April 1985.
- [Pea80] M. Pease, R. Shostak, and L. Lamport, "Reaching Agreement in the Presence of Faults", *Journal of the ACM*, April 1980, pp. 228-234.
- [Tre93] S. Treadwell, *Estimating Task Execution Delay in a Real-Time System via Static Source Code Analysis*, MS Thesis, Massachusetts Institute of Technology, Cambridge, MA, May 1993.

System Performance Modeling and Analysis of a Fault-Tolerant, Real-Time Parallel Processor

by

Robert J. Clasen, Captain, USAF

Department of Electrical and Computer Engineering
Northeastern University, Boston, MA

Master of Science degree, 1993

Abstract

The use of fault-tolerant, real-time systems for the control of life-critical processes is becoming increasingly common, with examples including flight and nuclear reactor control systems. In such systems, the overhead associated with managing redundancy, communication, and task scheduling is critical due to real-time constraints imposed by the application; missed time deadlines can be viewed as system failures, with results as consequential as hardware failures.

The Fault-Tolerant Parallel Processor (FTPP) was developed by Draper Laboratory as a fault-tolerant, real-time computing platform. This thesis analyzes the FTPP prototype operating system overhead through the use of empirical performance measurement and two performance models based on these measurements. One model is developed to predict the operating system overhead under various configurations and workloads; accurate prediction of overhead provides confidence that real-time constraints can be satisfied. Because the system communication overhead may vary depending upon the amount of contention by Processing Elements for service by the Network Element, a second model is developed to account for performance delays that may result from this contention. When such performance analysis and modeling are an integral part of a concurrent build-analyze-improve methodology, performance bottlenecks can be cost-effectively removed at an early stage in development.

Empirical performance data show that the prototype FTPP Ada operating system (OS) overhead accounts for 22% of each 10 msec minor frame, excluding IO, for a given system configuration. The majority of the total overhead is due to communication overheads. The OS overhead model predicted the total overhead within an eight percent error. Simulation of the contention model showed that the most critical parameter in reducing the effect of contention is the time needed to transfer the packets from the local memory of the Processing Element to the Network Element.

Bibliography

- [Abl88] T. Abler, *A Network Element Based Fault Tolerant Processor*, MS Thesis, Massachusetts Institute of Technology, Cambridge, MA, May 1988.
- [Bab90] C. Babikyan, "The Fault Tolerant Parallel Processor Operating System Concepts and Performance Measurement Overview", *Proceedings of the 9th Digital Avionics Systems Conference*, October 1990, pp. 366-371.
- [Dol82] D. Dolev, "The Byzantine Generals Strike Again", *Journal of Algorithms*, 1982, pp. 14-30.
- [Dol84] D. Dolev, C. Dwork, and L. Stockmeyer, "On the Minimal Synchronism Needed for Distributed Consensus", IBM Research Report RJ 4294 (46990), 8 May 1984.
- [Fis82] M. Fischer and N. Lynch, "A Lower Bound for the Time to Assure Interactive Consistency", *Information Processing Letters*, 13 June 1982, pp. 183-186.
- [Har87] R. Harper, *Critical Issues in Ultra-Reliable Parallel Processing*, PhD Thesis, Massachusetts Institute of Technology, Cambridge, MA, June 1987.
- [Har88a] R. Harper, J. Lala, and J. Deyst, "Fault Tolerant Parallel Processor Overview", *18th International Symposium on Fault Tolerant Computing*, June 1988, pp. 252-257.
- [Har88b] R. Harper, "Reliability Analysis of Parallel Processing Systems", *Proceedings of the 8th Digital Avionics Systems Conference*, October 1988, pp. 213-219.
- [Har91] R. Harper and J. Lala, "Fault Tolerant Parallel Processor", *Journal of Guidance, Control, and Dynamics*, May-June 1991, pp. 554-563.
- [Joh89] B. Johnson, *Design and Analysis of Fault Tolerant Digital Systems*, Addison-Wesley, 1989.
- [Lam82] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem", *ACM Transactions on Programming Languages and Systems*, July 1982, pp. 382-401.
- [Leh89] T. Lehr, et. al., "Visualizing Performance Debugging", *Computer*, October 1989, pp. 38-51.
- [Pal85] D. Palumbo and R. Butler, "Measurement of SIFT Overhead", NASA Technical Memorandum 86722, Langley Research Center, Hampton, VA, April 1985.
- [Pea80] M. Pease, R. Shostak, and L. Lamport, "Reaching Agreement in the Presence of Faults", *Journal of the ACM*, April 1980, pp. 228-234.
- [Tre93] S. Treadwell, *Estimating Task Execution Delay in a Real-Time System via Static Source Code Analysis*, MS Thesis, Massachusetts Institute of Technology, Cambridge, MA, May 1993.

System Performance Modeling and Analysis of a Fault-Tolerant, Real-Time Parallel Processor

by

Robert J. Clasen, Captain, USAF

Department of Electrical and Computer Engineering
Northeastern University, Boston, MA

Master of Science degree, 1993

Abstract

The use of fault-tolerant, real-time systems for the control of life-critical processes is becoming increasingly common, with examples including flight and nuclear reactor control systems. In such systems, the overhead associated with managing redundancy, communication, and task scheduling is critical due to real-time constraints imposed by the application; missed time deadlines can be viewed as system failures, with results as consequential as hardware failures.

The Fault-Tolerant Parallel Processor (FTPP) was developed by Draper Laboratory as a fault-tolerant, real-time computing platform. This thesis analyzes the FTPP prototype operating system overhead through the use of empirical performance measurement and two performance models based on these measurements. One model is developed to predict the operating system overhead under various configurations and workloads; accurate prediction of overhead provides confidence that real-time constraints can be satisfied. Because the system communication overhead may vary depending upon the amount of contention by Processing Elements for service by the Network Element, a second model is developed to account for performance delays that may result from this contention. When such performance analysis and modeling are an integral part of a concurrent build-analyze-improve methodology, performance bottlenecks can be cost-effectively removed at an early stage in development.

Empirical performance data show that the prototype FTPP Ada operating system (OS) overhead accounts for 22% of each 10 msec minor frame, excluding IO, for a given system configuration. The majority of the total overhead is due to communication overheads. The OS overhead model predicted the total overhead within an eight percent error. Simulation of the contention model showed that the most critical parameter in reducing the effect of contention is the time needed to transfer the packets from the local memory of the Processing Element to the Network Element.

Bibliography

- [Abl88] T. Abler, *A Network Element Based Fault Tolerant Processor*, MS Thesis, Massachusetts Institute of Technology, Cambridge, MA, May 1988.
- [Bab90] C. Babikyan, "The Fault Tolerant Parallel Processor Operating System Concepts and Performance Measurement Overview", *Proceedings of the 9th Digital Avionics Systems Conference*, October 1990, pp. 366-371.
- [Dol82] D. Dolev, "The Byzantine Generals Strike Again", *Journal of Algorithms*, 1982, pp. 14-30.
- [Dol84] D. Dolev, C. Dwork, and L. Stockmeyer, "On the Minimal Synchronism Needed for Distributed Consensus", IBM Research Report RJ 4294 (46990), 8 May 1984.
- [Fis82] M. Fischer and N. Lynch, "A Lower Bound for the Time to Assure Interactive Consistency", *Information Processing Letters*, 13 June 1982, pp. 183-186.
- [Har87] R. Harper, *Critical Issues in Ultra-Reliable Parallel Processing*, PhD Thesis, Massachusetts Institute of Technology, Cambridge, MA, June 1987.
- [Har88a] R. Harper, J. Lala, and J. Deyst, "Fault Tolerant Parallel Processor Overview", *18th International Symposium on Fault Tolerant Computing*, June 1988, pp. 252-257.
- [Har88b] R. Harper, "Reliability Analysis of Parallel Processing Systems", *Proceedings of the 8th Digital Avionics Systems Conference*, October 1988, pp. 213-219.
- [Har91] R. Harper and J. Lala, "Fault Tolerant Parallel Processor", *Journal of Guidance, Control, and Dynamics*, May-June 1991, pp. 554-563.
- [Joh89] B. Johnson, *Design and Analysis of Fault Tolerant Digital Systems*, Addison-Wesley, 1989.
- [Lam82] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem", *ACM Transactions on Programming Languages and Systems*, July 1982, pp. 382-401.
- [Leh89] T. Lehr, et. al., "Visualizing Performance Debugging", *Computer*, October 1989, pp. 38-51.
- [Pal85] D. Palumbo and R. Butler, "Measurement of SIFT Overhead", NASA Technical Memorandum 86722, Langley Research Center, Hampton, VA, April 1985.
- [Pea80] M. Pease, R. Shostak, and L. Lamport, "Reaching Agreement in the Presence of Faults", *Journal of the ACM*, April 1980, pp. 228-234.
- [Tre93] S. Treadwell, *Estimating Task Execution Delay in a Real-Time System via Static Source Code Analysis*, MS Thesis, Massachusetts Institute of Technology, Cambridge, MA, May 1993.

System Performance Modeling and Analysis of a Fault-Tolerant, Real-Time Parallel Processor

A Thesis Presented

by

Robert John Clasen

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements
for the degree of

Master of Science

in the field of

Computer Engineering

Northeastern University
Boston, Massachusetts

May 27, 1993

© Robert J. Clasen, 1993.

System Performance Modeling and Analysis of a Fault-Tolerant, Real-Time Parallel Processor

by

Robert J. Clasen

Submitted to the Department of Electrical and Computer Engineering
in partial fulfillment of the requirements for the degree of

Master of Science

Abstract

The use of fault-tolerant, real-time systems for the control of life-critical processes is becoming increasingly common, with examples including flight and nuclear reactor control systems. In such systems, the overhead associated with managing redundancy, communication, and task scheduling is critical due to real-time constraints imposed by the application; missed time deadlines can be viewed as system failures, with results as consequential as hardware failures.

The Fault-Tolerant Parallel Processor (FTPP) was developed by Draper Laboratory as a fault-tolerant, real-time computing platform. This thesis analyzes the FTPP prototype operating system overhead through the use of empirical performance measurement and two performance models based on these measurements. One model is developed to predict the operating system overhead under various configurations and workloads; accurate prediction of overhead provides confidence that real-time constraints can be satisfied. Because the system communication overhead may vary depending upon the amount of contention by Processing Elements for service by the Network Element, a second model is developed to account for performance delays that may result from this contention. When such performance analysis and modeling are an integral part of a concurrent build-analyze-improve methodology, performance bottlenecks can be cost-effectively removed at an early stage in development.

Empirical performance data show that the prototype FTPP Ada operating system (OS) overhead accounts for 22% of each 10 msec minor frame, excluding IO, for a given system configuration. The majority of the total overhead is due to communication overheads. The OS overhead model predicted the total overhead within an eight percent error. Simulation of the contention model showed that the most critical parameter in reducing the effect of contention is the time needed to transfer the packets from the local memory of the Processing Element to the Network Element.

Acknowledgments

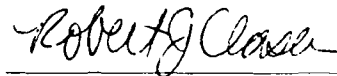
I would like to thank everyone at Draper Laboratory who made my fellowship here so rewarding, and special thanks are in order for Rick Harper and Carol Babikyan. As my thesis supervisor, Rick gave me the freedom to pursue a topic that interested me, and he was always helpful and encouraging. It was through Carol that I began to unravel the intricacies of the FTPP operating system. Her patience with my endless stream of questions was amazing, and she never once seemed annoyed with me, not even when I asked her to explain "scoop message" to me for the fifteenth time.

I also thank my thesis advisor at Northeastern University, Professor Ed Czeck. Professor Czeck took an early and active interest in my thesis, and his involvement dramatically improved its quality. I also thought it was great that he often played a 10,000 Maniacs compact disc during our weekly meetings.

On a more personal note, I thank my mom and dad. They gave me the most important gift that any child can receive: a sense of wonder and curiosity about the world we live in. As Vincent van Gogh once wrote, "One must always try to know deeper, better, and more." Finally, and most importantly, I thank my clever and witty wife, Sharon. I wouldn't have been able to complete my graduate studies with my sanity intact were it not for her unending support, and it is to her that I dedicate this thesis.

This thesis was prepared at the Charles Stark Draper Laboratory, under NASA contract NAS1-18565. Publication of this report does not constitute approval by Draper Laboratory or Northeastern University of the findings or conclusions contained within. It is published solely for the exchange and stimulation of ideas.

I hereby assign my copyright of this thesis to the Charles Stark Draper Laboratory, Incorporated, of Cambridge, Massachusetts.



Robert J. Clasen

Permission is hereby granted to the Charles Stark Draper Laboratory and to Northeastern University to reproduce and to distribute copies of this thesis in whole or in part.

Contents

Chapter 1. Introduction	12
1.1 Problem Statement	12
1.2 Objective	13
1.3 Approach	14
 Chapter 2. Fault Tolerance Fundamentals	 15
2.1 Hardware Fault Tolerance Concepts	15
2.1.1 Redundancy	15
2.1.2 Hardware Redundancy	16
2.1.2.1 Passive Redundancy	17
2.1.2.2 Active Redundancy	17
2.1.2.3 Hybrid Redundancy	19
2.1.3 Hardware Fault Tolerance in FTTP	20
2.2 Byzantine Resilience	21
2.2.1 Byzantine Faults	21
2.2.2 Requirements for Byzantine Resilience	22
2.2.3 Byzantine Resilience on FTTP	23
 Chapter 3. FTTP System Description	 25
3.1 FTTP Hardware	27
3.1.1 Network Element	29
3.1.1.1 Data Exchange Primitives	29
3.1.1.2 System Maintenance Primitives	31
3.2 Operating System Overview	31
3.2.1 Tasking Services	32
3.2.1.1 FTTP Scheduling Overview	33
3.2.1.2 Rate Group Dispatcher	35
3.2.1.3 Rate Group Tasks	36

3.2.2	Communication Services.....	37
3.2.2.1	Message and Packet Structure.....	38
3.2.2.2	Message Buffering.....	39
3.2.2.3	Message Transmission.....	40
3.2.2.4	Message Reception.....	41
3.2.3	Time Management.....	42
3.2.4	IO Services.....	42
3.2.4.1	IO User Interface.....	44
3.2.4.2	IO Communication Manager.....	45
3.2.5	Fault Detection, Identification and Recovery (FDIR).....	47
3.2.5.1	Local Fault Detection, Identification and Recovery.....	48
3.2.5.2	Global Fault Detection, Identification and Recovery.....	48
3.2.6	Overview of a Minor Frame.....	48
Chapter 4.	Performance Models	51
4.1	Operating System Overhead Model.....	51
4.1.1	Interrupt Handler (IH ₁) Overhead.....	52
4.1.2	Rate Group Dispatcher - Part One (RGD ₁) Overhead.....	53
4.1.3	IO Dispatcher (IOD) Overhead.....	53
4.1.4	Interrupt Handler (IH ₂) Overhead.....	54
4.1.5	Rate Group Dispatcher - part two (RGD ₂) Overhead.....	54
4.1.6	Fault Detection Identification and Recovery (FDIR) Overhead.....	55
4.1.7	IO Source Congruency Manager (IOSC) Overhead	55
4.1.8	IO Processing Task (IOP) Overhead.....	55
4.1.9	OS Overhead Summary.....	56
4.2	Contention Model.....	57
4.2.1	The Model.....	57
4.2.1.1	Processing of Message Packets.....	58
4.2.1.2	Contention for NE Services Among Two or More PEs.....	60
4.2.1.3	Simplifying Assumptions.....	61
4.2.2	Contention Simulation.....	64
4.2.3	Results of the Simulation.....	65

Chapter 5. Performance Measurement Methodology	70
5.1 Software Probes.....	71
5.1.1 Description of Data Recorded by Software Probes.....	71
5.1.2 Example of Software Probe Use.....	72
5.2 Transfer of Data from the FTPP to the Host VAX.....	74
5.3 Data Analysis.....	74
5.3.1 Determination of Time Interval.....	74
5.3.2 Statistical Analysis of Time Data.....	76
 Chapter 6. Performance Measurement Results	 77
6.1 Interrupt Handler Overhead	78
6.1.1 Scoop Message.....	78
6.2 Rate Group Dispatcher (Part One) Overhead.....	79
6.2.1 Record Congruent Time Value, Check for RGD2 Overrun.....	80
6.2.2 Check for RG Task Overruns.....	80
6.2.3 Set Up Next RG Interval, Schedule IO Dispatcher.....	80
6.3 IO Dispatcher (IOD) Overhead.....	81
6.4 Rate Group Dispatcher (Part Two) Overhead.....	82
6.4.1 Update Congruent Time Value, Check for RGD1 and IOD Overrun.....	83
6.4.2 Send Queue.....	83
6.4.3 Update Queue.....	84
6.4.4 Schedule Rate Group Tasks.....	86
6.4.5 Increment Frame Number, Set Up IO Interval for Next Frame.....	86
6.4.6 RGD2 Summary.....	87
6.5 Fault Detection, Identification, and Recovery (FDIR) Overhead.....	87
6.6 IO Source Congruency Manager (IOSC) Overhead	87
6.7 IO Processing Task (IOP) Overhead.....	88
6.8 Other Overheads.....	88
6.8.1 Queue Message.....	88
6.8.2 Retrieve Message.....	89
6.8.3 Context Switch Overhead.....	91
6.9 Performance Data Summary	91

Chapter 7. Detailed OS Overhead Model	94
7.1 OS Overhead Model with Empirical Data	94
7.1.1 Interrupt Handler (IH ₁) Overhead.....	94
7.1.2 Rate Group Dispatcher - Part One (RGD ₁) Overhead.....	95
7.1.3 IO Dispatcher (IOD) Overhead.....	96
7.1.4 Interrupt Handler (IH ₂) Overhead.....	96
7.1.5 Rate Group Dispatcher - Part Two (RGD ₂) Overhead.....	97
7.1.6 Fault Detection Identification and Recovery (FDIR) Overhead.....	99
7.1.7 IO Source Congruency Manager (IOSC) Overhead	99
7.1.8 IO Processing Task (IOP) Overhead.....	99
7.1.9 Total OS Overhead.....	100
7.2 Example of Overhead Model Use.....	101
7.2.1 Description of Example System Configuration	101
7.2.2 Predicted Overheads.....	102
7.2.3 Comparison of Predicted and Actual Overheads.....	104
7.2.4 Other Uses of OS Overhead Model.....	106
Chapter 8. Conclusions	107
8.1 Major Contributions.....	107
8.2 Suggested Further Research.....	109
Appendix A. Acronyms	111
Appendix B. Contention Model Source Code	113
Appendix C. Statistical Analysis Source Code	120
Appendix D. References	135

Figures

Figure 1-1. Use of Performance Analysis in Removing Performance Bottlenecks.....	13
Figure 1-2. Thesis Structure.....	14
Figure 2-1. Passive Redundancy Using Triple Modular Redundancy (TMR).....	17
Figure 2-2. Active Redundancy Using Duplication with Comparison.....	18
Figure 2-3. Active Redundancy Using Standby Sparing	19
Figure 2-4. Hybrid Redundancy Using TMR with Spares.....	20
Figure 2-5. Example of the Byzantine Generals' Problem.....	22
Figure 2-6. FCR Interconnections to Achieve Byzantine Resilience.....	23
Figure 3-1. FPHP Abstract Layered Structure.....	26
Figure 3-2. FPHP Physical Architecture	28
Figure 3-3. FPHP Virtual Configuration	29
Figure 3-4. FPHP Operating System Structure.....	32
Figure 3-5. Architecture of RG Frames on a Single VG.....	34
Figure 3-6. Rate Group Frame - Programming Model.....	35
Figure 3-7. Overview of Task Communication Process	38
Figure 3-8. Transmit and Receive Queues (PE Local Memory).....	40
Figure 3-9. FPHP IO Services	43
Figure 3-10. The IO User Interface and IO Communication Manager.....	44
Figure 3-11. Overview of Minor Frame.....	49
Figure 4-1. Minor Frame Overview	52
Figure 4-2. Message Packet Processing	59
Figure 4-3. Contention Timeline.....	61
Figure 4-4. Phasing Among PEs.....	63
Figure 4-5. Contention Model Timeline	64
Figure 4-6. Effect of Varying Number of Packets and Phasing on Time to Send Message Packets	66
Figure 4-7. Effect of Varying Number of PEs and Phasing on Time to Send Message Packets	67

Figure 4-8. Effect of Varying Process Packet Time and Transfer Packet Time on Time to Send Message Packets.....	67
Figure 4-9. Effect of Varying Process SERP Time and Transfer Packet Time on Time to Send Message Packets.....	68
Figure 4-10. Effect of Reducing Each Default Parameter by 50% on Time to Send Message Packets.....	69
Figure 5-1. Performance Measurement Overview.....	71
Figure 5-2. Placement of Software Probes.....	72
Figure 6-1. Graphical Representation of Send Queue and Update Queue Execution Time as a Function of Number of Packets.....	85
Figure 6-2. Graphical Representation of Queue Message and Retrieve Message Execution Time as a Function of Number of Packets.....	90
Figure 7-1. Comparison of Time to Schedule Tasks (Measured) with Least Square Line Approximation.....	105

Tables

Table 5-1. Representative Use of Debug Log Data Fields.....	73
Table 5-2. Overheads Associated with debug_log Procedure Calls.....	75
Table 6-1. Scoop Message Execution Time as a Function of Number of Packets	78
Table 6-2. Overall Rate Group Dispatcher (Part One) Execution Time as a Function of Minor Frame Number.....	79
Table 6-3. RGD ₁ Update Congruent Time Value and Check for RGD ₂ Overrun Execution Time.....	80
Table 6-4. RGD ₁ Check for Rate Group Task Overruns Execution Time as a Function of Number of Rate Group Tasks.....	80
Table 6-5. RGD ₁ Set Up RG Interval and Schedule IO Dispatcher Execution Time	81
Table 6-8. Overall Rate Group Dispatcher (Part Two) Execution Time as a Function of Minor Frame Number.....	82
Table 6-9. RGD ₂ Update Congruent Time Values and Check for RGD ₁ and IOD Overrun.....	83
Table 6-10. RGD ₂ Send Queue (Per Task) Execution Time as a Function of Number of Packets	84
Table 6-11. RGD ₂ Update Queue (Per Task) Execution Time as a Function of Number of Packets.....	85
Table 6-12. RGD ₂ Schedule Rate Group Tasks Execution Time as a Function of Number of Tasks Per Rate Group	86
Table 6-13. RGD ₂ Increment Frame Number and Set IO Interval Execution Time	86
Table 6-14. Local FDIR Execution Time.....	87
Table 6-15. Minimal IO Source Congruency Manager Execution Time.....	88
Table 6-15. Minimal IO Processing Task Execution Time.....	88

Table 6-16. Queue Message Execution Time as a Function of Message Size.....	89
Table 6-17. Retrieve Message Execution Time as a Function of Message Size.....	90
Table 6-18. Context Switch Execution Time.....	91
Table 6-19. OS Overhead Due to Communication, Scheduling and Fault Detection (Average Vaules for a Minor Frame).....	92
Table 7-1. System Parameters for Each Minor Frame.....	102
Table 7-2. Comparison of Predicted and Measured Overheads.....	104

Chapter 1

Introduction

1.1 Problem Statement

The use of fault-tolerant, real-time systems for the control of life-critical processes is becoming increasingly common. Examples include flight control systems and nuclear reactor control systems. These types of applications have two common requirements. First, they have tasks which must meet hard deadlines; a missed deadline is as disastrous as total system failure. Second, they must be extremely reliable. This requires the use of massive redundancy to achieve high reliability goals.

The Fault-Tolerant Parallel Processor (FTPP), developed by Draper Laboratory, was designed to tolerate hardware faults and to perform within the real-time constraints required by an application. FTPP achieves high reliability by combining Byzantine resilience with the use of redundant, concurrently executing processors. The FTPP scheduler uses a Rate Group paradigm to ensure that task execution times are predictable within guaranteeable worst-case limits.

As with other fault-tolerant, real-time systems, FTPP obtains high reliability and predictable timing characteristics by paying a price. The added cost is not only due to the redundant processor boards, it also is a result of the computing resources needed to manage the system's redundancy and real-time operation. Operating system activities such as voting the output of a redundant processing group, monitoring the execution times of tasks, and reconfiguring the system due to a failed processor, require computing time that could otherwise be used by application tasks.

The amount of operating system overhead determines the amount of time available for application tasks. This overhead should be minimal and as predictable as possible to maintain real-time requirements. FTPP requires a mechanism that allows accurate prediction of the amount of operating system overhead under various configurations and workloads. Such a mechanism could be used to determine the amount of time available to execute application tasks.

1.2 Objective

This thesis aims to develop two models (based on empirical performance data) that can be used to accurately predict the operating system overhead. The first model, called the operating system overhead model, uses performance data to estimate the overhead of various operating system tasks. This model assumes that processing elements do not have to contend for the shared system resource known as the Network Element. The second model, called the contention model, determines the effect of contention on the Rate Group Dispatcher, which is the only operating system task that is affected by contention.

In addition to their use in the models to predict available processing time, the collected performance data have the useful side effect of drawing attention to potential performance bottlenecks in the operating system. When performance measurements are collected concurrently with operating system development, these bottlenecks can be removed at an early and cost-effective stage of development. Figure 1-1 shows the use of performance analysis in eliminating performance bottlenecks.

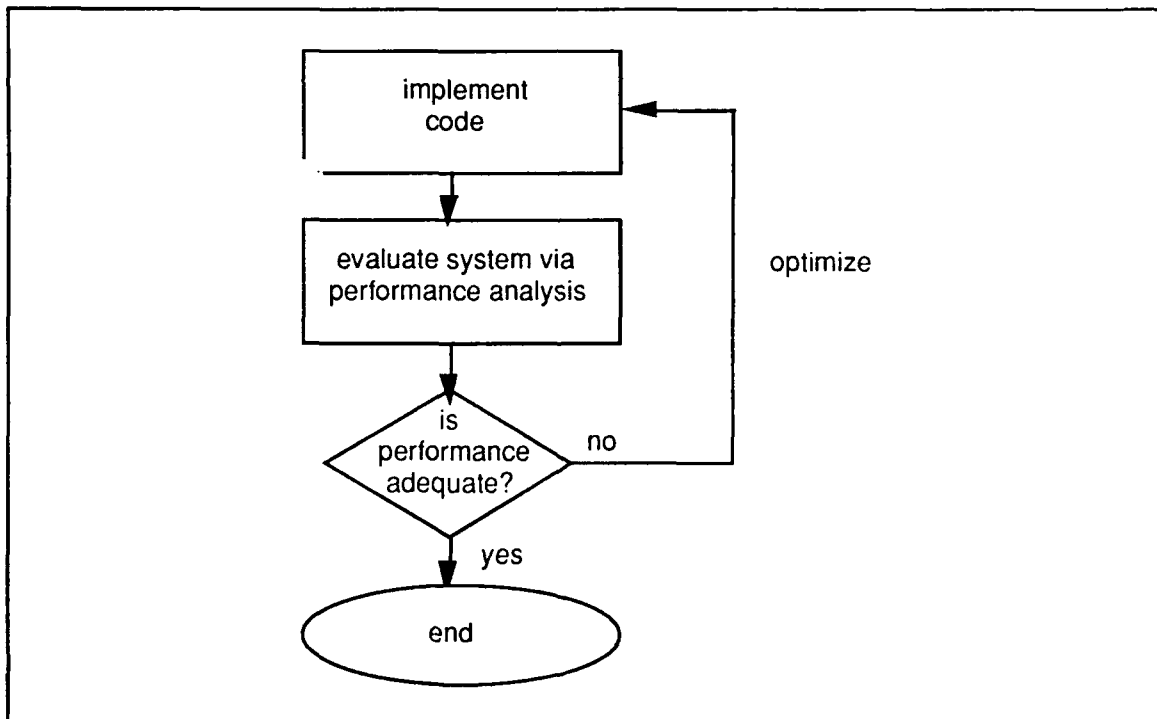


Figure 1-1. Use of Performance Analysis in Removing Performance Bottlenecks

1.3 Approach

This study begins with an overview of some fault tolerance fundamentals, including a discussion of Byzantine resilience, in Chapter 2. Chapter 3 describes both the FTPP hardware and operating system. In Chapter 4, the overhead model and the contention model are described; in addition, the results of the contention model simulation are presented. A description of the methodology used to collect performance data is given in Chapter 5, and Chapter 6 summarizes the data collected using this methodology. The application of the performance data to the operating system overhead model results in the detailed overhead model of Chapter 7. Finally, a summary of significant results and suggestions for future research are given in Chapter 8. Figure 1-2 shows the structure of the main body of this thesis. In the appendices, an acronym list and source code for the contention model simulation and the statistical analysis program are given.

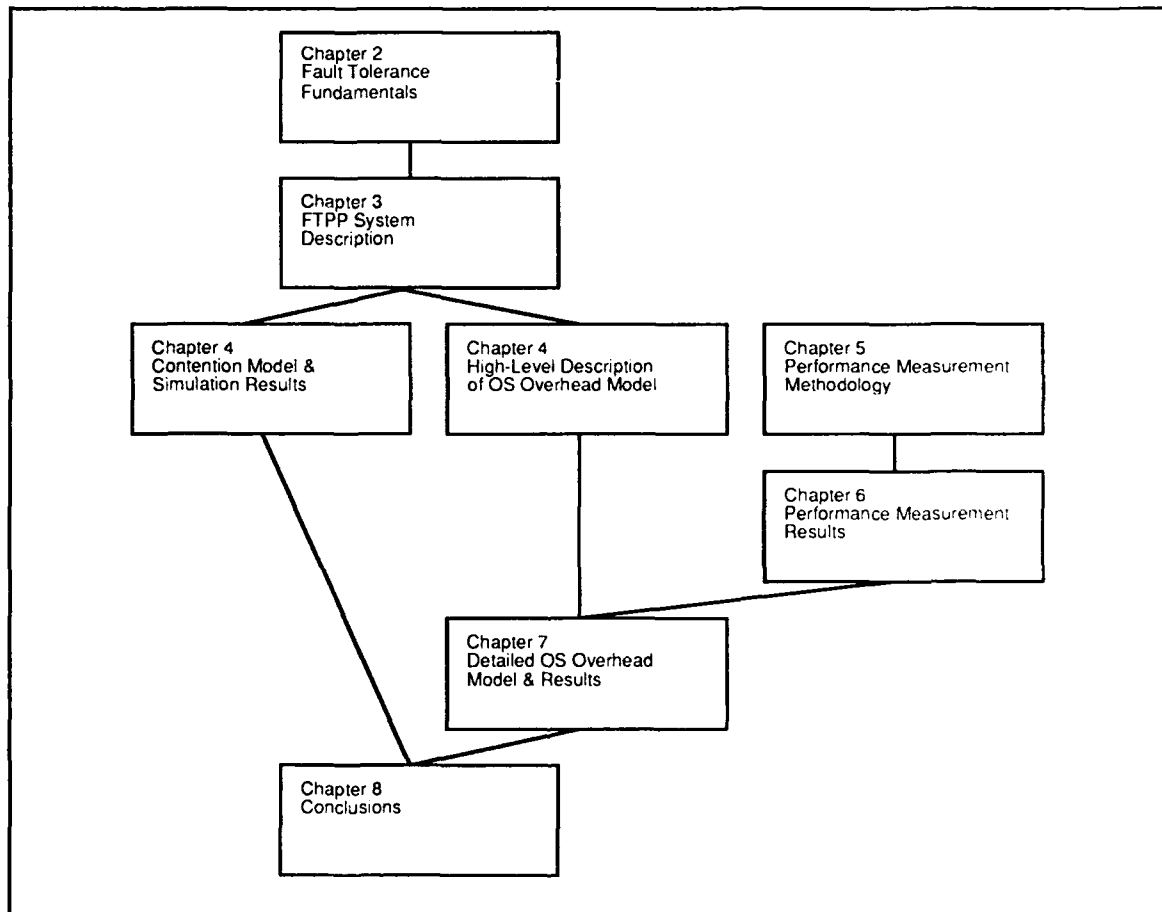


Figure 1-2. Thesis Structure

Chapter 2

Fault Tolerance Fundamentals

Applications having a high cost of failure, such as an aircraft control system, require the use of highly-reliable computing systems. Since it is impossible to guarantee that failures will not occur within a system, such systems need to be designed to tolerate any component failure. The discipline of fault-tolerant computing focuses on effective and efficient methods to achieve dependability in the presence of faults. This chapter provides a general discussion of some of the fundamentals of fault tolerance. Section 2.1 gives an overview of hardware fault tolerance techniques and describes the implementation of some of these techniques on the Fault-Tolerant Parallel Processor (FTPP). While the methods discussed in Section 2.1 are adequate for tolerating the most commonly expected faults, ultra-reliable computer systems, like FTPP, need to adopt a more conservative fault model which includes Byzantine faults. Section 2.2 discusses the Byzantine Resilience approach used to tolerate these faults and its implementation on FTPP.

2.1 Hardware Fault Tolerance Concepts

FTPP uses hardware redundancy to tolerate hardware failures and to achieve high reliability. An overview of some of the basic concepts associated with hardware fault tolerance is presented here. This section begins with a discussion of the use of redundancy to tolerate faults and then elaborates on some hardware redundancy techniques. Finally, the implementation of these techniques on FTPP is described.

2.1.1 Redundancy

All forms of fault tolerance rely on some type of redundancy. This redundancy can take one of several forms [Joh89]:

- *Hardware Redundancy*

Extra hardware is used to provide fault detection, masking, or diagnosis. This is the most common type of fault tolerance technique.

- *Software Redundancy*

Additional software, beyond that which is needed for normal use, is provided to detect and possibly tolerate faults. Typically, each copy of the software is designed and implemented by members of independent teams.

- *Information Redundancy*

Extra bits are appended to instructions or data to detect and possibly correct errors. Error detecting and correcting codes (such as Hamming codes) are examples of the use of information redundancy.

- *Time Redundancy*

Operations are repeated to allow recovery from transient or intermittent faults. Time redundancy trades poorer performance for a reduction in the amount of extra hardware. In many applications, time is less important than hardware due to size or weight constraints.

The addition of redundancy to a system is costly. For hardware redundancy, the additional expense is the increased size, weight, and cost of redundant components. Redundant software takes longer to develop and requires more memory. Coding techniques require extra hardware and/or software to generate and interpret redundant data information, and as mentioned above, time redundancy techniques are costly in terms of performance.

2.1.2 Hardware Redundancy

Despite the cost of using redundancy, the high reliability requirements of many applications necessitate its use. The most common type of redundancy used in fault-tolerant systems is the physical replication of hardware. Because digital hardware has become smaller, faster, and less expensive in recent years, the costs of implementing hardware redundancy have decreased.

There are three general types of hardware redundancy [Joh89]: passive, active, and hybrid. Passive techniques use masking to hide the occurrence of faults. Active techniques detect the presence of a fault and then perform some action to remove it. The hybrid approach combines features of the passive and active methods. Each of these three techniques is examined more closely in the following paragraphs.

2.1.2.1 Passive Redundancy

Passive hardware redundancy masks the presence of faults by using a majority voting mechanism. Explicit location of a faulty module is not necessary to achieve fault tolerance because the erroneous data is masked, regardless of location.

The most common type of passive redundancy is called Triple Modular Redundancy (TMR). In TMR, the hardware module is triplicated, and the three outputs of each module are given to a voter to determine the correct value. If one of the modules is faulty, the other two modules will mask the fault, and the voter will provide the correct output. The structure of a TMR system is given in Figure 2-1.

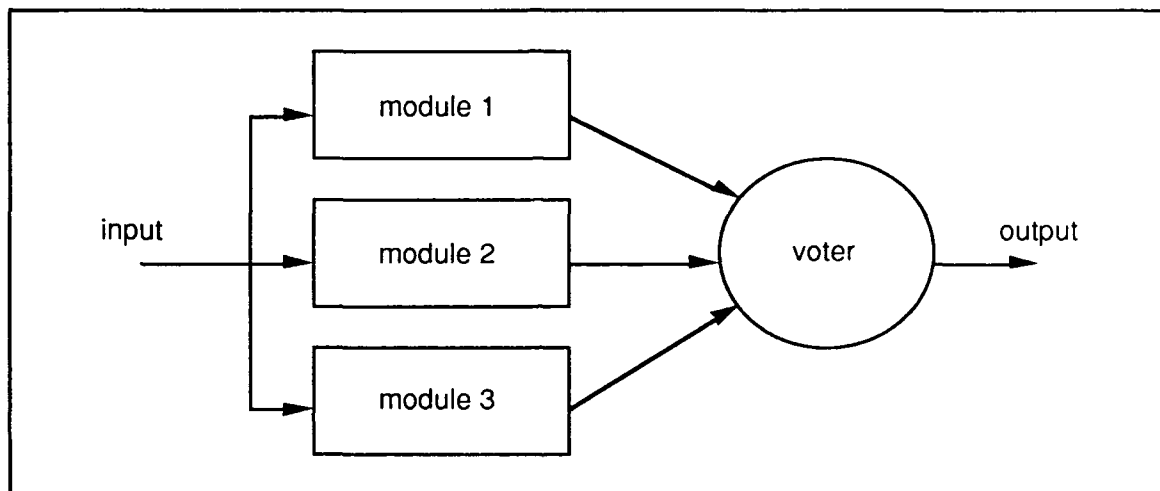


Figure 2-1. Passive Redundancy Using Triple Modular Redundancy (TMR)

The overhead for TMR is 200%, not including the voter, which can be very complex. The voter is the weak link in the TMR scheme; if the voter fails, the system fails. Great effort must be taken to ensure the high reliability of the voter. However, despite its high cost, TMR is very general and can be applied at any level of a parallel system.

2.1.2.2 Active Redundancy

Unlike passive redundancy which immediately masks a fault, active hardware redundancy techniques must first detect and locate a fault before correcting it. Therefore, strictly active methods are most commonly used in applications that can allow temporarily erroneous results, as long as reconfiguration occurs within a reasonable amount of time. The two most popular implementations of active redundancy are duplication with comparison and standby sparing.

Duplication with Comparison

In duplication with comparison (Figure 2-2), two identical copies of a module operate on the same data and supply their results to a comparator to check for equality. If the comparator detects a disagreement, it generates an error signal. Upon receipt of the error signal, the decision unit causes both modules to perform self-tests to determine the faulty module. Based on the results of the self-tests, the decision unit determines which module's output to use as the system output, and the faulty module can later be replaced or switched-out of the system.

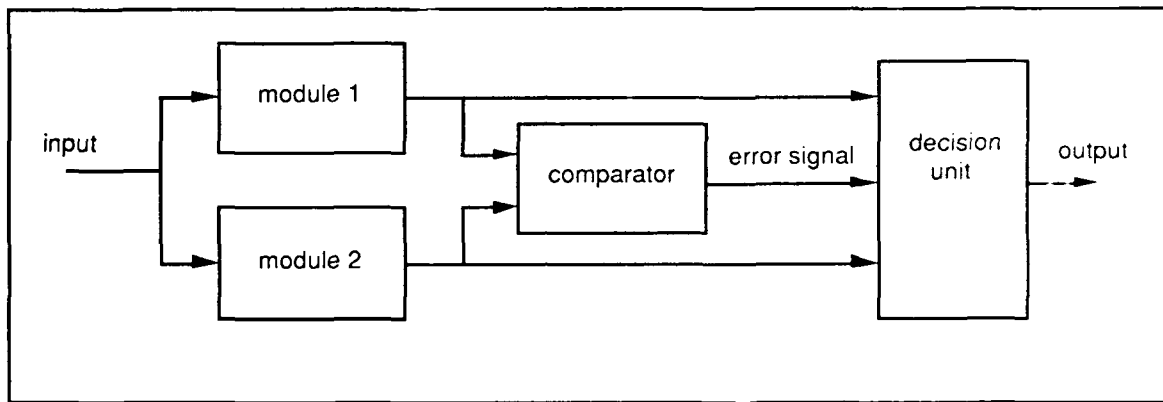


Figure 2-2. Active Redundancy Using Duplication with Comparison

This technique requires 100% overhead, not counting the comparator. The circuitry of the comparator is much less complex than that of the voter used in TMR, so the comparator is less expensive and typically more reliable. However, a comparator failure results in system failure, just as a voter failure results in TMR system failure.

Standby Sparing

A second example of active hardware redundancy is standby sparing (Figure 2-3). In standby sparing, one module (known as the primary) is operational, and another module serves as a standby or spare. The primary module employs some type of error detection scheme (perhaps self-testing) to determine the validity of its output. If the output of the primary is faulty, the spare becomes operational, and the faulty module is switched out of the system. Some implementations of this technique use more than one spare module, and therefore tolerate more than one fault.

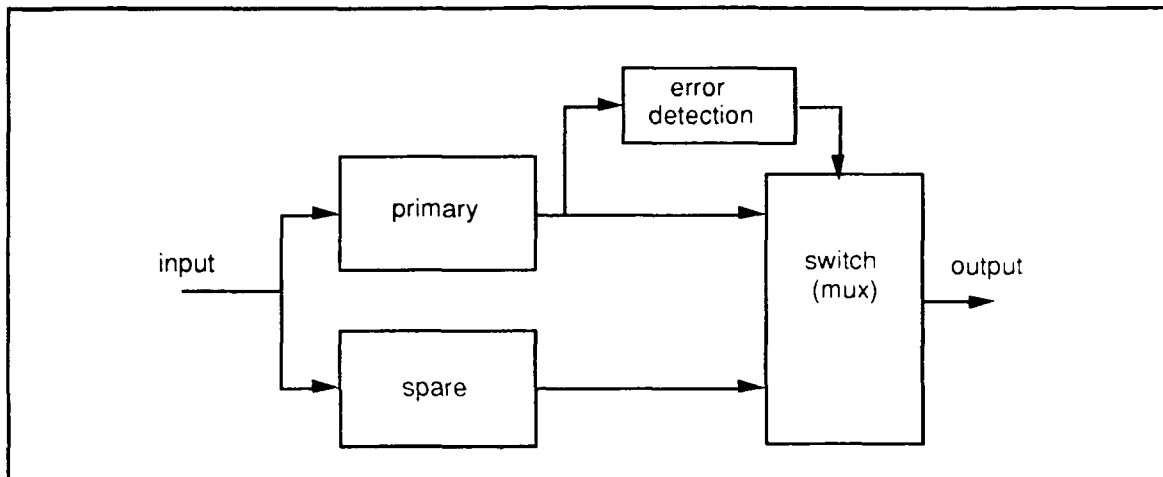


Figure 2-3. Active Redundancy Using Standby Sparring

The success of the standby sparing scheme is dependent upon the reliability of the error detection process used by the primary module.

2.1.2.3 Hybrid Redundancy

The third type of hardware redundancy is known as hybrid redundancy. Hybrid redundancy combines features of both the passive and active approaches. Fault masking is used to prevent the system from producing erroneous results, and fault detection and location information is used to reconfigure the system after a fault has occurred. Of the three hardware redundancy techniques, the hybrid approach is the most expensive to implement in terms of hardware.

An example of hybrid redundancy is the TMR with spares scheme, shown in Figure 2-4. The technique provides a core of three modules arranged in a voting configuration. Spare modules provide replacements to faulty modules in the TMR core, while the disagreement detector locates any module whose output does not agree with the majority. The faulty module is then removed from the TMR core, and one of the spare modules is switched in to replace it.

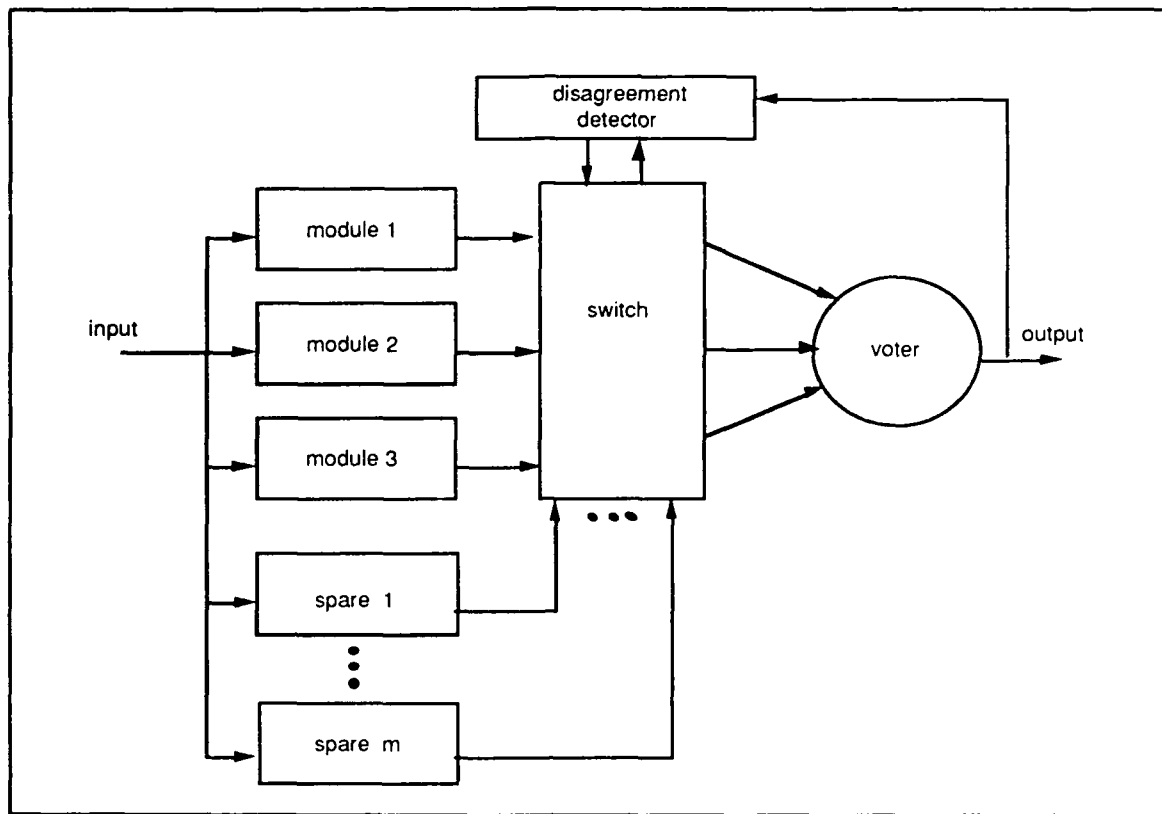


Figure 2-4. Hybrid Redundancy Using TMR with Spares

If m spare modules are available, the system can tolerate $m + 1$ failed modules. For example, if one spare module is used, the system can tolerate two module failures. In a purely passive scheme, five modules would be needed to tolerate two failures.

2.1.3 Hardware Fault Tolerance in FTTP

FTTP uses a hybrid redundancy approach to achieve fault tolerance. As will be described in more detail in Chapter 3, Processing Elements (PEs) can be grouped virtually as triplexes (three PE modules) or quads (four PE modules). Results from each replicated PE are exchanged and voted using messages. In addition, syndrome information indicating any discrepancies is attached to the message. The majority version of the message is delivered to each destination. Up to this point, the system has performed passive fault tolerance by masking any errors.

Active redundancy is used to identify faulty PEs and to reconfigure the system to avoid their use. This is accomplished through the use of Fault Detection, Identification and Recovery (FDIR) software task. FDIR uses syndrome information to analyze the health of each PE. Upon detection of a fault, several options exist. The faulty PE may be

given a reset signal in an attempt to recover from a transient fault, or the system may be reconfigured by replacing the faulty PE with a spare. The application programmer is responsible for choosing appropriate FDIR options.

2.2 Byzantine Resilience

The hardware redundancy techniques described earlier in this chapter can be used to tolerate the types of faults that are most likely to occur in a digital system. However, ultra-reliable computer systems need to adopt a more conservative fault model, one that does not rely on any *a priori* assumptions about component behavior. For example, if a module fails in such a way that it produces conflicting outputs, traditional hardware redundancy techniques may not be able to reach agreement on the correct output since no clear majority exists. Byzantine resilience is a fault tolerance technique that guarantees a system can tolerate this type of malicious fault. Since the concept of Byzantine resilience is central to the theory, design, and operation of the FTPP [Harp91], it is discussed in some detail in this section. First, a description of Byzantine faults is given, and then a list of requirements to tolerate these faults is provided. Finally, an overview of the implementation of Byzantine resilience on FTPP is given.

2.2.1 Byzantine Faults

As mentioned in the preceding paragraph, a faulty module may fail in such a way as to display seemingly malicious behavior. This type of fault, known as a Byzantine fault, could include behavior such as starting and then restarting execution, sending conflicting information to different destinations, and any other action a failed component may do which can corrupt the system. The source of this terminology can be found in [Lam82]:

"Reliable computer systems must handle malfunctioning components that give conflicting information to different parts of the system. This situation can be expressed abstractly in terms of a group of generals of the Byzantine army camped with their troops around an enemy city. Communicating only by messenger, the generals must agree upon a common battle plan. However, one or more of them may be traitors who will try to confuse the others. The problem is to find an algorithm to ensure that the loyal generals will reach agreement."

An example of the Byzantine generals' problem is shown in Figure 2-5. In this figure, General #1 is loyal and orders the other two generals to attack. However, General #2 is a

traitor, and he reports to General #3 that he received a "retreat" order. General #3 now has conflicting information and may not be able to correctly decide the proper course of action.

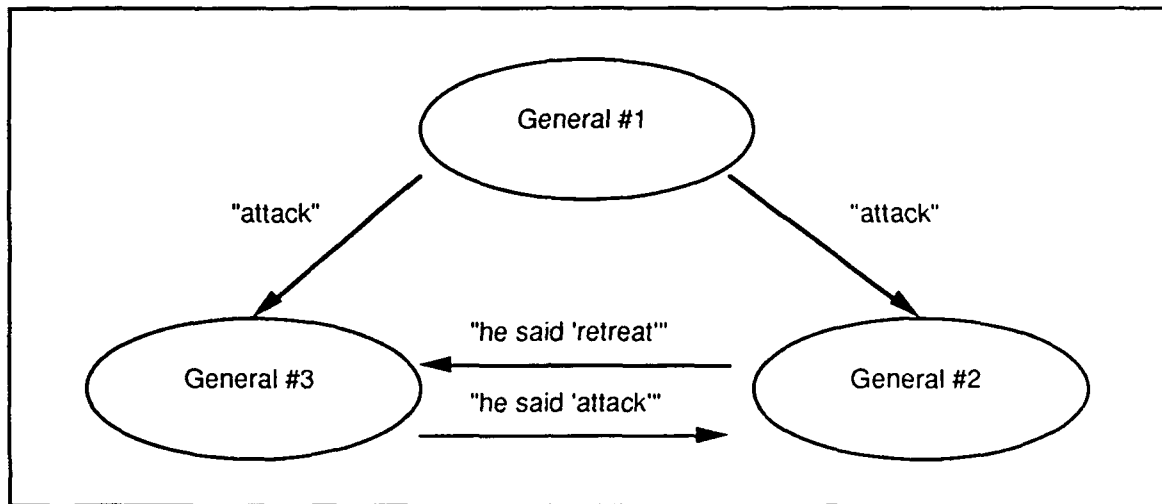


Figure 2-5. Example of the Byzantine Generals' Problem

In this analogy, the generals correspond to Processing Elements, the traitors to faulty PEs, and the messages to inter-processor communication links.

2.2.2 Requirements for Byzantine Resilience

It is obvious that a system designed to tolerate Byzantine faults is more complex than one that is designed to handle more traditional faults. Simple redundancy techniques are no longer sufficient when Byzantine faults are included in the fault model. To ensure a protocol can tolerate f arbitrarily-failed members, four prerequisites must be satisfied. These requirements are summarized below:

1. There must be at least $3f + 1$ participants in the protocol [Pea80].
2. Each participant must be connected to each other participant through at least $2f + 1$ disjoint communication paths [Dol82].
3. The protocol must consist of a minimum of $f + 1$ rounds of communication among the participants. [Fis82].
4. The participants must be synchronized to within a known skew of each other [Dol84].

A system that meets these requirements is called f -Byzantine resilient. For example, a 1-Byzantine resilient fault masking group would have four participants, each of which is

connected to each other participant by three disjoint communication paths. Communication among the participants would consist of a synchronous, two-round exchange.

2.2.3 Byzantine Resilience on FTPP

FTPP is designed as a *1*-Byzantine resilient system; therefore, it satisfies each of the four prerequisites listed above. This section gives a simplified overview of how FTPP fulfills each of these requirements.

The first requirement -- at least 4 participants -- is easily satisfied since FTPP allows PEs to be grouped as simplexes (1 member), triplexes (3 members), and quads (4 members). By choosing virtual groups of processors configured as quads, the requirement of four members is met.

The second requirement for Byzantine resilience calls for each member of the group to communicate with each other by at least three disjoint communication paths. FTPP PEs are contained in one of up to five Fault Containment Regions (FCRs) that are present in the system. Each of the four members of a quad are physically located in separate FCRs, and each FCR is connected to each other via an optical link network. Figure 2-5 shows the connections of a four-FCR system. By picturing each member of a quad residing in a different FCR, it is easy to see that three disjoint communication paths exist for any PE to communicate to any other PE in the quad.

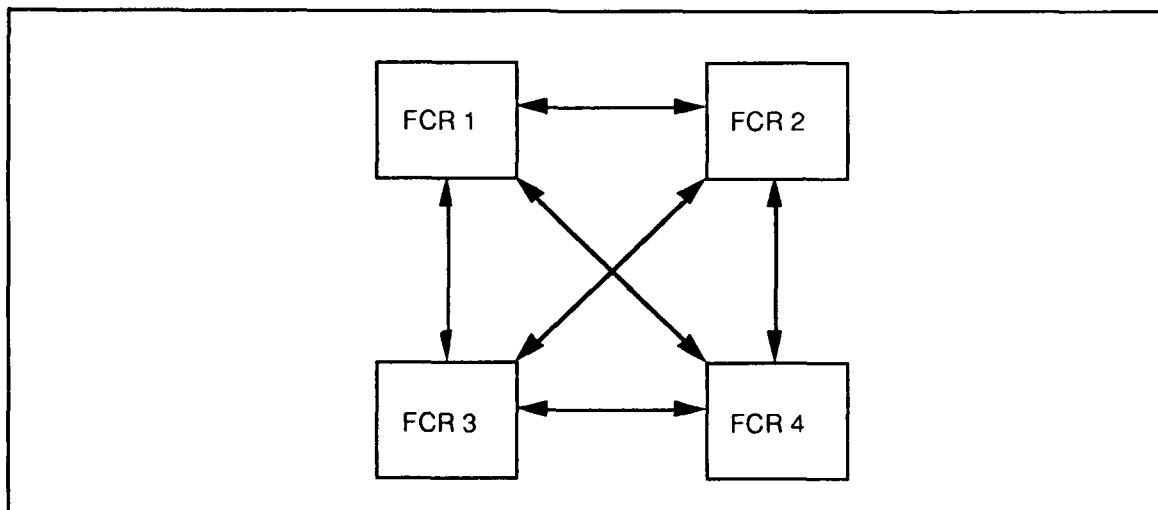


Figure 2-6. FCR Interconnections to Achieve Byzantine Resilience

A *1*-Byzantine resilient system must have at least two rounds of communication among its members to satisfy the third prerequisite. FTPP provides a two-round source

congruency exchange to distribute data. In the first phase of this exchange, the source PE sends its message to the three other members of its quad. In the second phase, each PE in the quad sends a copy of the message it received to the other three members. Each PE in the quad then votes the copies of the message to arrive at a consistent result.

The final requirement for Byzantine resilience calls for the participants to be synchronized to within a known skew of each other. In FTPP, functional synchronization occurs twice during each ten-millisecond time interval. Each PE sends a special message which is distributed, voted and delivered to each member of the quad. The operating system blocks awaiting return of the synchronization message, thus allowing the slowest member of the quad to "catch up" with the others and reduce the skew among them.

Chapter 3 discusses in more detail the application of these Byzantine resilience techniques to FTPP.

Chapter 3

FTPP System Description

The Fault-Tolerant Parallel Processor (FTPP) architecture was developed by Draper Laboratory to satisfy the dual requirements of ultra-high reliability and high throughput. For reliability, the FTPP is designed to be resilient to Byzantine faults. For throughput, the architecture includes multiple processing elements providing parallel processing capability.

The FTPP architecture is described in references [Abl88], [Bab90], [Har87], [Har88a], [Har88b], and [Har91]. It is composed of Processing Elements (PEs) and specially designed hardware components referred to as Network Elements (NEs). The multiple Processing Elements provide a parallel processing environment as well as components for hardware redundancy. The group of Network Elements acts as the intercomputer communications network and the redundancy management hardware. As with most computing systems, FTPP can be viewed as a layered system (Figure 3-1).

The top layer consists of the applications programs themselves. In an ideal world, applications are constructed by the applications engineers without regard for the parallel and redundant nature of the FTPP system. With this view in mind, FTPP supports a virtual architecture of Ada tasks which executing in parallel, subject to preemption, and data and control flow dependencies. In reality, the applications engineers must assist in the selection of appropriate task-to-processing site mappings, processing site redundancy levels, fault recovery strategies, and other parameters available through the FTPP architecture.

The next lower layer consists of the FTPP System Services; this layer is intended to mask the complexity of the FTPP's lower layers from the programmer. Certain services are visible and may be invoked by the applications programmer; these include input/output, task scheduling, and intertask communication services. Other important functions of the FTPP System Services are not directly accessible by the applications programmer and are performed in a manner which is largely transparent. These include the functions of mapping of tasks to processing sites, routing intertask messages to remote tasks, disassembling and reassembling long messages, performing input/output functions.

and fault detection, identification, and recovery (FDIR). The application tasks and FTPP System Services execute on the FTPP Processing Elements, as indicated in Figure 3-1.

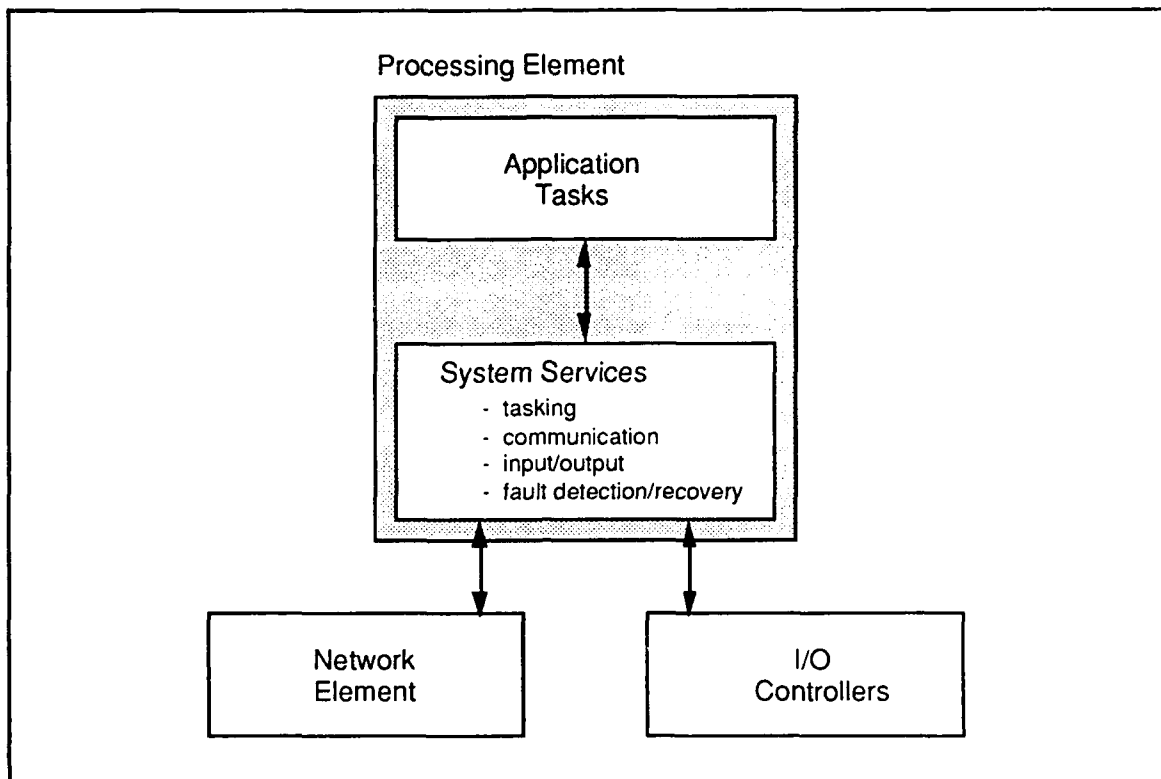


Figure 3-1. FTPP Abstract Layered Structure

The next lower layer of FTPP consists of the interprocessor communication network hardware, known as Network Elements. This hardware implements the interprocessor message passing functions of FTPP. In addition, it implements throughput-critical, fault-tolerant functions such as: voting of messages from redundant processing sites; providing error indications; assisting in synchronizing redundant processing sites; and assisting in arranging the non-redundant processing resources of FTPP into redundant processing sites based on the needs of the application and the fault state of FTPP.

FTPP Communication Services run on the Processing Elements and interface to the Network Elements over a standard backplane bus. FTPP Input/Output (IO) Services also run on the Processing Elements and communicate with the Input/Output Controllers over a standard backplane bus; this may be separate from the bus hosting the Processing Element-Network Element interface.

This chapter continues the description of FTPP in two sections. Section 3.1 describes the FTPP hardware, and Section 3.2 presents an overview of the FTPP operating system.

3.1 FTPP Hardware

The FTPP is composed of Processing Elements (PEs), Input/Output Controllers (IOCs), Power Conditioners (PCs), backplane/chassis assemblies, and specially designed hardware components referred to as Network Elements (NEs).

A diagram of the physical FTPP configuration is shown in Figure 3-2. The FTPP cluster consists of four or five Fault Containment Regions (FCRs). A fault occurring in one FCR can not cause another FCR to malfunction. Fault containment is achieved by providing each FCR with independent sources of power, clocking, as well as dielectric and physical isolation. FCRs may either be distributed for damage tolerance or integrated if damage tolerance is not an issue. Each FCR contains an NE, between zero and eight PEs, a PC, and zero or more IOCs. A minimal FTPP system configuration consists of four NEs and three PEs; a maximal system would consist of five NEs and 40 PEs. Selection of the number of NEs and PEs for a given application is made according to performance, reliability, availability, and other engineering requirements. Devices in an FCR are interconnected using one or more standardized backplane buses.

The NEs provide communication between PEs, maintain synchronization among the FCRs, maintain data consensus among FCRs, and provide dielectric isolation between the FCRs via fiber optic links. The NE also implements the protocol requirements for Byzantine resilience [Lam82]. Each PE consists of a processor, private RAM and ROM, and miscellaneous support devices, such as a periodic timer. The PEs may optionally have private IO devices. The PE may be either a general-purpose processor or a special-purpose processor for signal or image processing. The prototype FTPP uses Motorola 68030-based processor boards as PEs. The IOCs connect FTPP to the outside world, and they can be any module that is compatible with the FCR standard backplane bus.

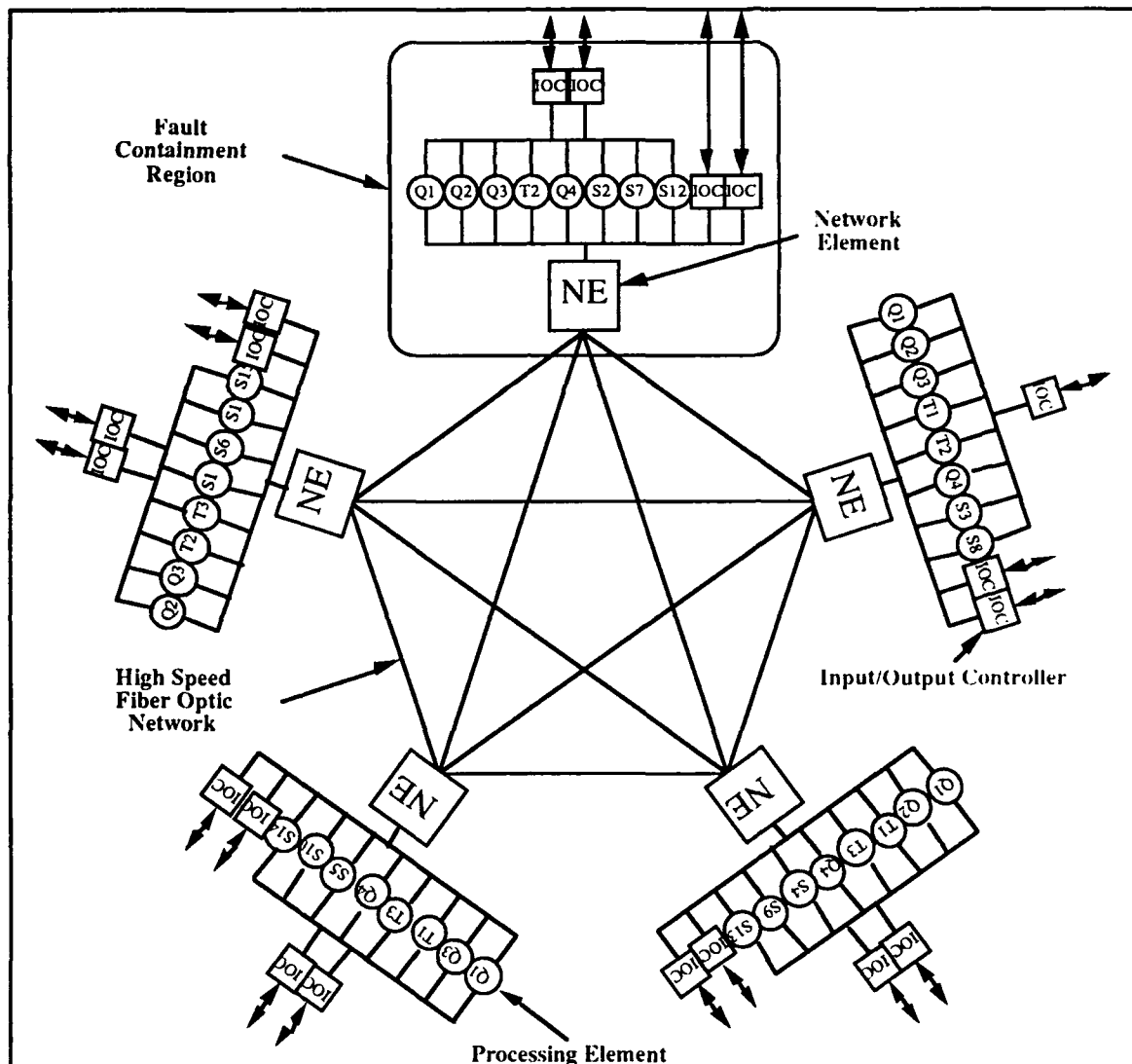


Figure 3-2. FTTP Physical Architecture

To achieve fault tolerance, individual PEs are grouped into Virtual Groups (VGs). Byzantine-resilient triplexes and quadruplex VGs consist of three and four PEs, respectively, with no more than one PE taken from any FCR. Virtual groups consisting of only one processing site are called simplexes. Arbitrary mixes of redundancy levels can be supported. The ensemble of Network Elements provides a virtual bus abstraction connecting the VGs (Figure 3-3). This abstraction conceals the multiple NEs and their interconnect, by replacing it with a simple bus-oriented abstraction.

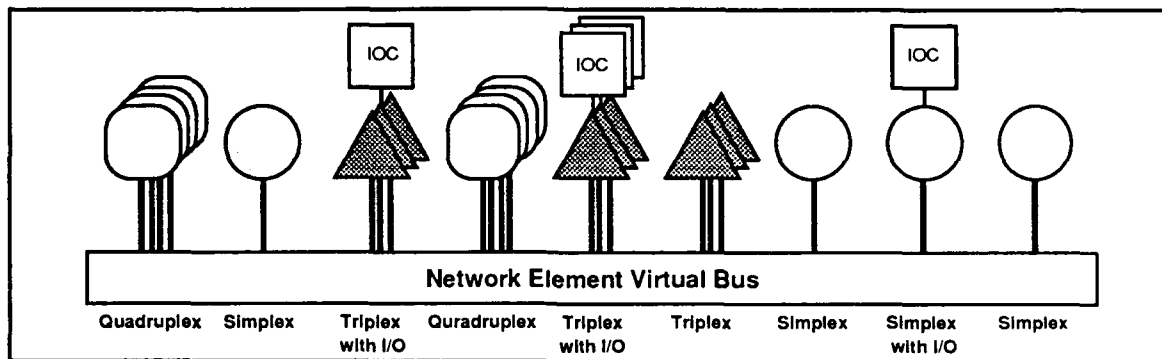


Figure 3-3. FTPP Virtual Configuration

3.1.1 Network Element

The Network Element is the core of an FTPP cluster. The Network Element connects on one side to a number of processing sites, and on the other side to the other Network Elements in the cluster. The ensemble of Network Elements forms a virtual bus network through which the processors communicate. The following paragraphs describe the functions provided by the Network Element. These functions can be grouped as data exchange primitives and system maintenance primitives.

3.1.1.1 Data Exchange Primitives

The Network Element provides a number of data exchange primitives. The primary use of these primitives is to transfer data from one virtual processing site to another. However, some primitives are used to vote common-source data or distribute single-source data within a virtual processing site. Most of the primitives are solely used by the FTPP operating system. When the application requests inter-VG communication, the operating system transparently maps the communication request to the appropriate data exchange primitive.

Data is transferred from a Processing Element to the associated Network Element through the processor's output buffers, which contain message packets. Each buffer contains 64 bytes of data, and each buffer has a descriptor which specifies the primitive to be executed, the destination Virtual Group, and the location of the data in the output data block. The Network Element is responsible for delivering the buffers to the destination processor via the processor's input buffers.

The following paragraphs describe the four types of data exchange primitives: class 0, class 1, class 2, and broadcasts.

Class 0 Data Exchange Primitive

The class 0 primitive is used when only the side effects of a data primitive are needed; it does not exchange any data. Examples of some side effects include synchronization, timestamping, and syndrome reporting. When a Virtual Group executes a class 0 primitive, all of the descriptor information is valid, except for the vote syndrome, which is undefined. The data in the output data block do not need to be defined. As a result, the data are not guaranteed to be congruent among members of the destination Virtual Group and must be ignored.

Class 1 Data Exchange Primitive

The class 1 primitive performs a single round of exchange and vote on data from a fault-masking Virtual Group (FMG). FMGs include triplex and quad VGs. Only FMGs are allowed to execute the class 1 primitive, since at least three independent copies of data are required for an unambiguous bitwise majority vote. In a class 1 data exchange, the independent copies of data are voted, and the voted result is delivered to each member of the destination Virtual Group.

Class 2 Data Exchange Primitive

The class 2 primitive performs a two-round, or source congruency, exchange on data from any Virtual Group. The source of the data may be a simplex VG or a single member of a triplex or quad. The class 2 primitive is the only mechanism by which simplexes are allowed to communicate with other VGs. The class 2 data exchange is performed in two phases. During the first phase, the source VG sends its data to each member of the destination VG. For the second phase, each member of the destination VG sends a copy of the data it received from the source VG in the first phase to the other members of the destination VG. Then, each member of the destination VG votes the copies of the message data to arrive at a consistent result.

Broadcasts

Broadcasts are a useful means of transmitting data to all active Virtual Groups in the cluster. Since broadcasts are more of a drain on system resources than the standard point-to-point communication primitives, only FMGs are allowed to use them. The broadcast primitive is invoked as a modifier to the existing exchange primitives. Any of the primitives, including the data exchange and the system maintenance primitives, can be delivered as a broadcast. A VG can determine that a received packet was delivered as part of a broadcast by examining the broadcast modifier bit in the descriptor.

3.1.2.2 System Maintenance Primitives

In addition to providing primitives for data exchange, the Network Element also allows for periodic system maintenance through the use of primitives. A special set of primitives produce side effects that directly affect the state of the Network Element. One such primitive is a configuration table update, which allows the mapping of physical processors to Virtual Groups to be changed. Another primitive allows transient recovery of a failed NE or PE which has been repaired or restarted. Another system maintenance primitive initially synchronizes the NEs at power-up or after a system reset.

3.2 Operating System Overview

The foundation of the prototype operating system for FTPP consists of the vendor-supplied Ada Run-Time System and Draper-supplied extensions. FTPP processing is distributed by task, and intertask communication is provided by message passing. High reliability is provided by redundantly executing the tasks on replicated processors. The FTPP hardware and software have been designed to hide the hardware redundancy, hardware faults, and the distributed processing details from the applications programmer. A system configuration specifies the mapping from tasks to VGs and from VGs to processors. This mapping is maintained by the operating system (in the configuration table) and is used to isolate the applications programmer from the underlying redundancy and distributed processing mapping.

The functional structure of the operating system is shown in Figure 3-4. The operating system consists of two layers: one for services that application tasks access directly, and a lower layer that does not provide direct service for application tasks. Services that may be invoked by the applications programmer include task scheduling, intertask communication, and input/output. This layer is intended to mask the complexity of FTPP's lower layers from the programmer. Other important functions of the FTPP System Services are not directly accessible by the applications programmer and are performed in a manner which is largely transparent. These functions include Fault Detection, Identification and Recovery (FDIR) and Time Management. The following paragraphs describe these five major services provided by the FTPP operating system.

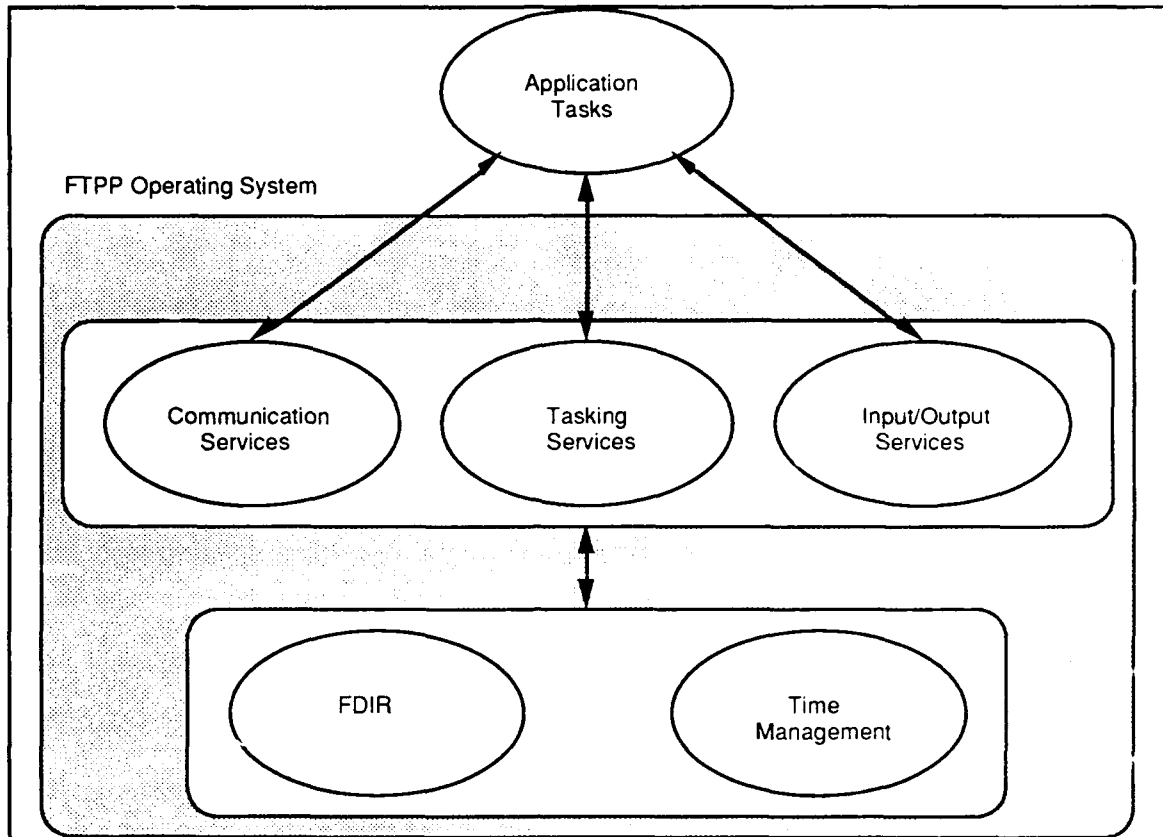


Figure 3-4. FTPP Operating System Structure

3.2.1 Tasking Services

Within a Virtual Group, multiple tasks require the use of the message passing resource. To maintain congruent use of this resource across the members of a VG, it is necessary to ensure there is no competition for its use. This is done by limiting the preemption allowed in the system and by limiting the use of the message passing resource. A Rate Group tasking paradigm was developed to fulfill these requirements. The paradigm consists of the Rate Group tasking services (Section 3.2.1), the communication services (Section 3.2.2), and the time management services (Section 3.2.3).

Within the Rate Group tasking services, tasks are assigned to execute as either RG1, RG2, RG3, or RG4 tasks; a Rate Group Dispatcher controls their execution. The tasks in each Rate Group must be cyclic and execute one complete iteration within their Rate Group frame. The communication services prevent preemptible tasks from using the message passing resource directly. Instead, their messages are buffered on queues controlled by the Rate Group Dispatcher. This removes the possibility of contention for the

message passing resource. The time management services establish the Rate Group frame boundaries.

The Rate Group tasking services are described in the next three sections. An overview of FTPP scheduling is given in Section 3.2.1.1. The Rate Group Dispatcher is described in Section 3.2.1.2, and the structure of Rate Group tasks is described in Section 3.2.1.3.

3.2.1.1 FTPP Scheduling Overview

The FTPP supports two different paradigms for scheduling tasks on a single Virtual Group. The first, known as *Rate Group scheduling*, is suitable for task suites in which each task has a well-defined iteration rate and can be validated to have an execution time which is guaranteed to not exceed its iteration frame. Flight control is an example of such a task. A modification of Rate Group scheduling discussed below also allows aperiodic hard real-time events to be processed. The second style of scheduling, known as *aperiodic non-real-time scheduling*, is available when the iteration rate of a particular non-real-time task is unknown or undefined. A mission planning algorithm is an example of such a task. Validation of the temporal behavior of such tasks may be difficult. In FTPP, non-real-time aperiodic tasks are not allowed to perturb the critical timing behavior of hard real-time tasks. In this thesis, only Rate Group scheduling is used and considered.

In the Rate Group paradigm, tasks executing on each VG in the FTPP are characterized by an iteration rate. In FTPP, these rates are nominally 100, 50, 25, and 12.5 Hz, corresponding to Rate Group identifiers RG4, RG3, RG2, and RG1, respectively. A Rate Group frame duration is the inverse of the Rate Group iteration rate; thus the RG4, RG3, RG2, and RG1 frames are 10, 20, 40, and 80 msec in duration, respectively (Figure 3-5). All frame boundaries are determined by crystal oscillator-controlled interrupts. The frequencies and number of Rate Group frames are readily changed as the application dictates. Frames executing on different VGs in FTPP do not need to have a particular phase relationship with each other.

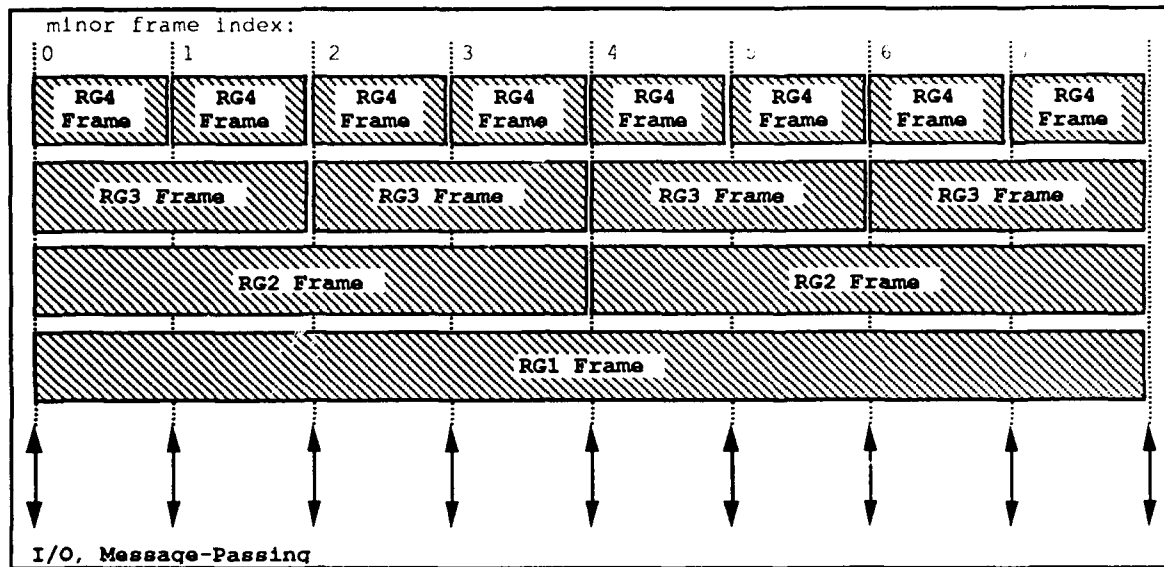


Figure 3-5. Architecture of RG Frames on a Single VG

To achieve multi-Rate Group execution on a VG, lower frequency Rate Group tasks are executed in the “margin” time left from the higher Rate Groups. Since these tasks have a lower frequency of execution, the higher frequency task scheduler periodically interrupts the lower frequency tasks to allow for the higher frequency tasks to execute. The interrupt and task switching process is transparent to the application programmer.

Within a particular Rate Group frame, tasks are scheduled using a nonpreemptive static schedule. When scheduled, a task executes to self-suspension. The exact time of execution of a particular task in the Rate Group frame will be in general unknown to the application programmer, and interactions between RG tasks and other entities occur only at RG boundaries. Instead, FTPP guarantees that all tasks within a Rate Group will be executed in the order specified by the application programmer sometime within the appropriate Rate Group frame. Figure 3-6 gives a programming model of a single Rate Group frame.

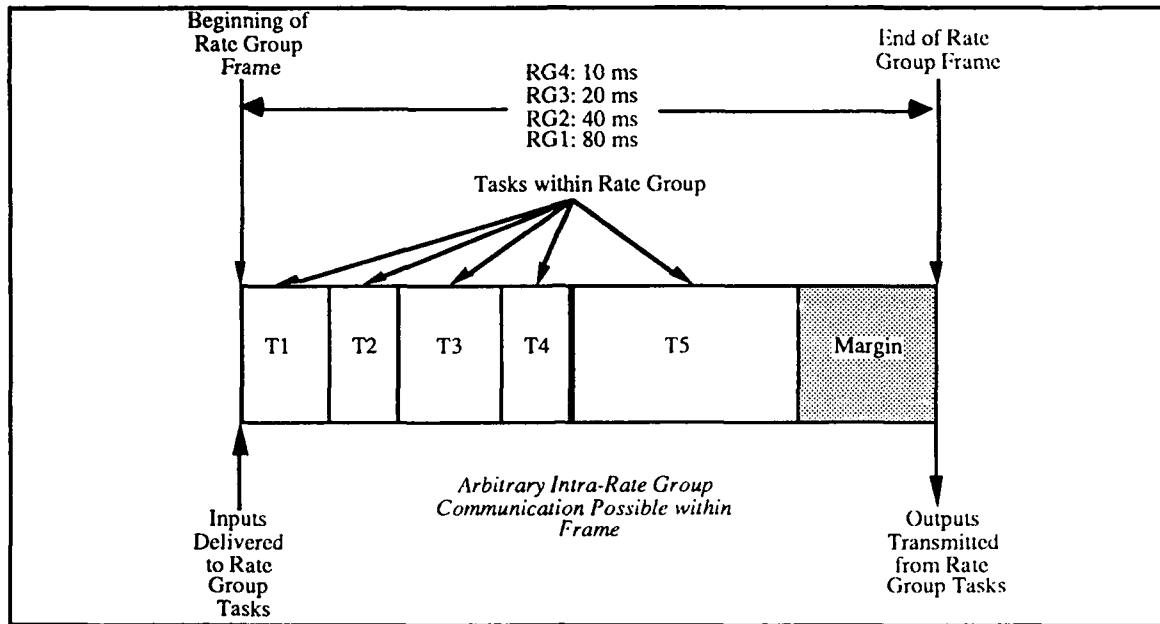


Figure 3-6. Rate Group Frame - Programming Model

Task overruns are detected by the Rate Group Dispatcher at the beginning of each frame. Since all tasks within a frame nominally execute to self-suspension, the Rate Group Dispatcher can detect a frame overrun by checking the suspension status of tasks which should have completed an iteration in the preceding Rate Group frame. Note that since the task which caused the overrun may itself have completed in the frame yet caused a subsequently-scheduled task to overrun, a verification technique must conclusively identify the task responsible for the overrun. Identification of the culprit is achieved by comparison of the actual measurement of each task's execution time with its predicted execution time. Note that this information is already needed for construction of the task schedule. Several overrun handling options exist and must be selected on a task-specific basis. Examples include aborting or restarting the culprit, or resuming the preempted task from its preemption point at the start of its next RG frame.

3.2.1.2 Rate Group Dispatcher

The Rate Group Dispatcher is a special RG4 task responsible for controlling the execution of the Rate Group tasks and providing reliable communication between Rate Group tasks throughout the system. It executes as two different parts. *Part one* (RGD₁) runs at the start of each minor frame and, based upon the minor frame index, determines the corresponding Rate Group frame boundaries. It checks that the tasks in these Rate Groups have completed an iteration of their execution cycle. After RGD₁ schedules the

IO Dispatcher task for execution, it suspends itself until the IO Completion interrupt occurs. Part two of the RG Dispatcher (RGD₂) will then execute.

After the IO Completion interrupt, *part two* of the Rate Group Dispatcher records a congruent value of the current time. It then determines the slowest Rate Group whose frame boundary corresponds to the start of this minor frame. Because of the mapping of Rate Group frames to minor frames, all faster Rate Groups will also be at a frame boundary and the identifier of the slowest Rate Group is used to indicate the entire set of Rate Groups at a frame boundary.

The `send_queue` and `update_queue` communication services are then called and passed the identifier of the slowest Rate Group at the frame boundary. `send_queue` transmits all the messages enqueued by the tasks of the corresponding Rate Groups in their previous frame. `update_queue` updates the communication service pointers to provide a congruent set of received messages and free buffers to the Rate Group tasks throughout their frame.

RGD₂ checks the overrun condition of each of the tasks which completed its execution cycle during the previous minor frame. If a task has overrun, the condition is logged in the Rate Group Dispatcher log. The log can be examined from the terminal display. RGD₂ then schedules the tasks in the appropriate Rate Groups for their next execution cycle.

RGD₂ then increments the minor frame index and suspends itself until the start of the next minor frame, which is when part one will run again. This allows the lower priority tasks which were previously executing or which were readied for execution by setting the Rate Group events to begin execution based upon their priority and precedence.

3.2.1.3 Rate Group Tasks

Rate Group tasks must be uniquely associated with a communication identification number (CID) and a corresponding task configuration table entry. The table entry must be initialized to specify whether the task is executing on one VG or executing on all VGs. System service tasks normally execute on all VGs. If a task executes on all VGs then broadcast messages can be used to send a message to all instantiations of the task. Otherwise the task instantiation must be identified by specifying the hosting VG identification number. If a task will execute on only one VG then the task's CID is sufficient to uniquely identify the task. The task's Rate Group and precedence within the Rate Group must also be specified; this determines how often the task will execute and the order in which tasks in the same Rate Group and on the same VG will execute. In addition, the maximum number and maximum size of messages that each task will queue

for transmission and that may be queued for its reception must be specified; these values are used to allocate packet buffers for the task's messages.

The task itself must have a well-defined cyclic execution behavior. The task and all the other tasks specified to execute on the VG must complete their execution cycle within their Rate Group frame. If they do not, then the Rate Group Dispatcher will detect an overrun condition for those tasks which did not complete within their frame. As explained in Section 3.2.1.1, these tasks are not necessarily the ones that caused the overrun condition to occur. Rate Group tasks use the `queue_message` and `retrieve_message` system calls to communicate between tasks.

3.2.2 Communication Services

The communication services are used to communicate among Rate Group tasks. Each Rate Group task has a communication identification number which can be used as its logical address. Other tasks in the system can send messages to this address, and the communication services will map the logical address to the VG executing the task. The communication is in the form of messages enqueued by the sender for transmission at the start of the next Rate Group frame and dequeued for reading by the recipient task within the next Rate Group frame after it is received. Messages are delivered in the order in which they were sent.

Figure 3-7 gives an overview of the task communication process. The sending task uses the `queue_message` procedure to decompose the message into 64-byte data packets. The Network Element hardware processes the packets and delivers them to the destination Processing Element. For redundant Virtual Groups (triplexes and quads), the NE processing consists of voting the redundant copies of each message packet and then delivering the voted result. The `scoop` procedure transfers the voted message packets from the dual-port RAM shared by the NE and PE to the local memory of the PE. The receiving task can then reconstruct the message by using the `retrieve_message` procedure.

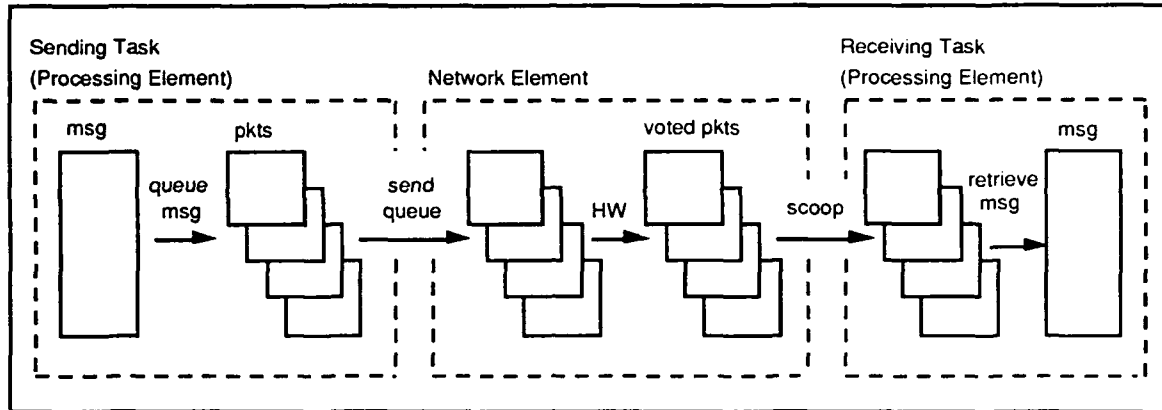


Figure 3-7. Overview of Task Communication Process

The following four sections provide a detailed discussion of the task communication process. The message and corresponding packet structures are described in Section 3.2.2.1. The message buffering process is described in Section 3.2.2.2. The message transmission procedures are described in Section 3.2.2.3, and the message reception procedures are described in Section 3.2.2.4.

3.2.2.1 Message and Packet Structure

The Rate Group tasks have a message-based interface to the communication services. The message itself is a contiguous block of data that is transferred from the sender to the receiver. Associated with the message are descriptor fields describing the sender, receiver, type of message, and how the message is to be exchanged. The message and message descriptor fields are supplied to the communication services by the task sending the message. The communication services then perform the exchange, and they deliver the message and descriptor information to the receiving task when it requests delivery of its messages.

Internally, the communication services store and manipulate the message as a set of fixed-size packets. A packet is the exchange unit used by the Network Elements, and each packet consists of a 64-byte block of data and a packet descriptor. The packet descriptors contain much of the same information found in the message descriptor. In addition, each packet descriptor has a boolean indicator signifying whether this is the last packet of the message. The packet descriptors are included with every packet and are voted along with the packet data by the Network Elements.

In addition to the packet descriptors mentioned above, a packet syndrome and packet timestamp are also provided by the NE for each delivered packet. These fields are main-

tained in the queue of received packets and are used by system services, but they are not delivered to the receiving task.

3.2.2.2 Message Buffering

A transmit packet queue and a receive packet queue are maintained for each task, and these queues are located in the Processing Element's local memory. They are the buffers between the underlying packet-based communication primitives which directly access the Network Elements and the message-based communication services which are used by the Rate Group tasks. The transmit queues are used to guarantee that the packets written to the NEs by the members of a VG have a consistent ordering. The receive queues are used to guarantee that Rate Group tasks see a consistent set of available messages. Both these conditions are necessary to guarantee that the members of a VG do not diverge.

Each queue is portioned into a set of active packets followed by a set of free packets. The active transmit packets contain data waiting to be written to the NE. The active receive packets contain data waiting to be read by a task. During initialization all the packets allocated for a task are placed in the free portion of the respective transmit or receive queue. The queues are maintained as linked lists with pointers to the entry at the head of the active portion, to the entry at the head of the free portion, and to the entry at the tail of the free portion. The structure of a Processing Element's transmit and receive queues is shown in Figure 3-8.

In the *transmit queues*, entries are moved from the free portion to the active portion when a message is enqueued by a task. Entries are removed from the active portion and replaced in the free portion when the stored packets are written to the Network Element. The transmit queues are maintained as singly-linked lists.

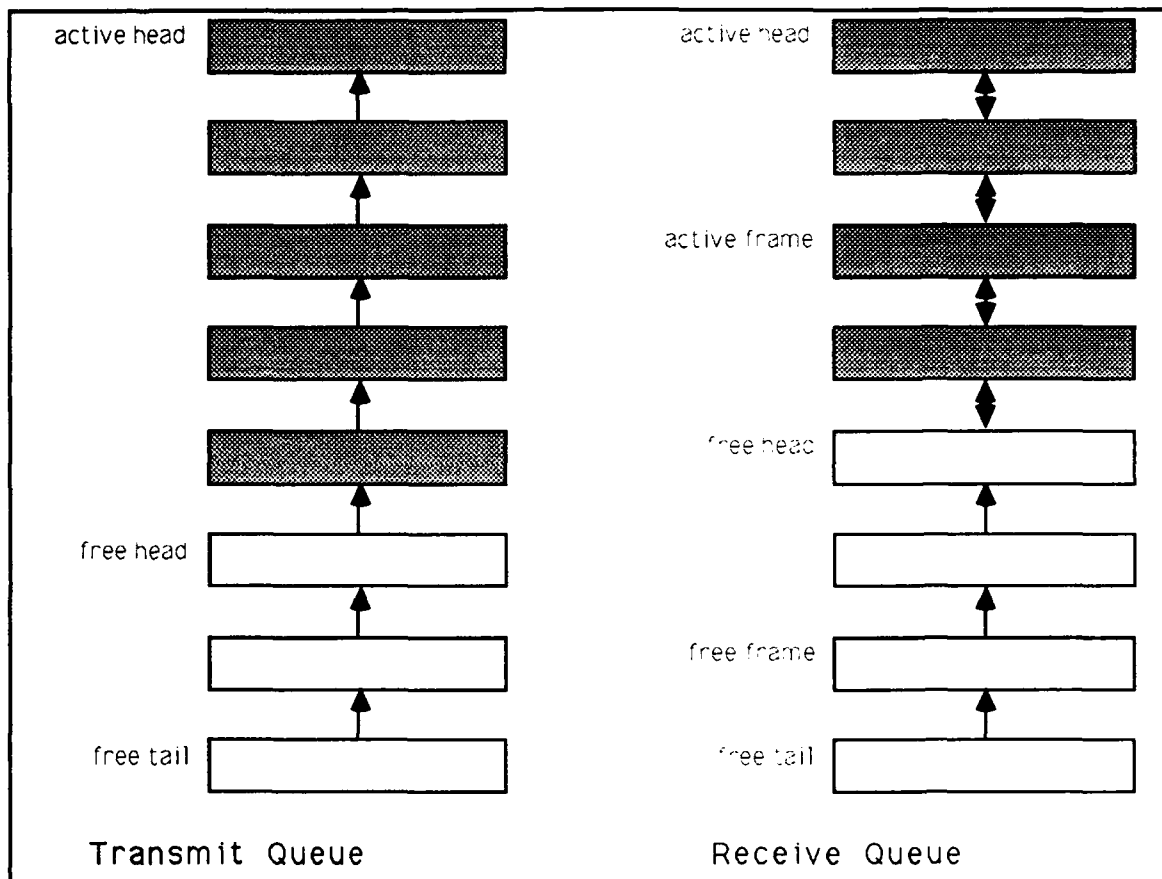


Figure 3-8. Transmit and Receive Queues (PE Local Memory)

In the *receive queues*, entries are moved from the free portion to the active portion when a packet is read from the Network Element. Entries are removed from the active portion and replaced in the free portion when a task retrieves a message. Because the packets of messages may be interleaved, entries may be removed from anywhere within the active portion. Therefore, the entries in this portion of the queue are doubly-linked.

3.2.2.3 Message Transmission

Messages are enqueued by Rate Group tasks for transmission at the end of a Rate Group frame. The message is packetized and enqueued using `queue_message` and is transmitted by the Rate Group Dispatcher (Section 3.2.3.2) using `send_queue`. When `queue_message` is called by a task, the source task is examined to determine where to queue the message. The message size is then examined to determine if there are enough free transmit packet buffers to enqueue the message. If there are, then the message descriptor and packet descriptors are constructed, and the message packets are written

into the free packet buffers. If there are insufficient buffers, a failure condition is returned.

At the end of each frame, the Rate Group Dispatcher (part two) determines the corresponding Rate Group frame boundaries. The dispatcher calls `send_queue` with the slowest Rate Group at a frame boundary as its parameter. Because of the mapping of Rate Group frames to minor frames, all faster Rate Groups are also at their frame boundary. `send_queue` examines the message queue in each task's local memory and transmits all the enqueued message packets to the transmit queue located in the dual-port RAM memory shared by the PE and NE. Any incomplete messages in the queue are flushed; this indicates a task overrun or other error condition and is logged.

3.2.2.4 Message Reception

Once the Network Element has processed a task's message packets, the packets need to be transferred from the receive queue (located in the dual-port RAM shared by the NE and its PEs) to the PE's local memory. This is accomplished through the use of the scoop procedure. The scooping of message packets occurs twice during each minor frame, once after the interrupt marking the beginning of a minor frame and once after the interrupt signifying the end of the IO interval. The PE performing a scoop simply sends a message to itself and awaits its reception. The scoop ensures that all members of a Virtual Group receive an identical set of identically ordered messages before delivery of its own scoop message [Har87].

Once the scoop has been performed, the pointers used by the receive queue need to be updated. The `update_queue` procedure is called by the Rate Group Dispatcher (part two) at a Rate Group frame boundary. It updates the set of packets usable by the tasks in that Rate Group within their next frame. When it is called it sets the receive free queue frame marker to the receive free queue tail for all the tasks in the Rate Group. Because the tasks in this Rate Group have completed their execution at the frame boundary, this provides the same set of free buffers for use when reading packets within the frame. `update_queue` then sets the receive active queue frame marker to the receive active queue tail for all the tasks in the Rate Group.

When the delivered message packets arrive in the PE's local memory, these individual packets need to be reassembled into complete messages. This is done through the `retrieve_message` procedure. `retrieve_message` returns the next available message to the calling task which has been read prior to the last frame marker. `retrieve_message` unpacketizes and reconstructs the next message at the message address specified in the `retrieve_message` call. The message descriptor fields are

then updated, and the freed buffers are placed at the tail of the receive free queue. Otherwise, an error condition is returned.

3.2.3 Time Management

The time management service is executed on each Processing Element to maintain congruent execution between the members of a VG and to provide a consistent system time for all the VGs. The system time is maintained by the Network Element aggregate as the elapsed time since system start up. When a packet is received by a Network Element, the system time is written to the packet's corresponding descriptor field. This time information is used by the time management service to define absolute time. Each PE locally maintains a timer to measure the elapsed time since the absolute time was updated. This timer is used to generate periodic interrupts to define the start of each minor Rate Group frame and to trigger the update of the absolute time. The interrupt causes preemption of the currently executing task by the Rate Group Dispatcher. The execution state of the system must be well-defined at these points to maintain congruent execution. This is provided by the Rate Group tasking implementation.

A chiming model is used to maintain the local value of system on each processor. *The local timer is used to generate the chime every minor frame period on each member of a VG. When the chime is generated, the VG members resynchronize and congruently update the local value of system time with the system time maintained by the Network Element. The updated time may not agree with the expected time of the chime and this difference is used to adjust the next chime interval and maintain a constant frame phasing among the VGs in the system. The local value of system time is only updated at the chime interrupt.*

3.2.4 IO Services

The Input/Output (IO) services provide communications between the application program and external devices (sensors and actuators). IO services execute on any VG responsible for IO, and provide distribution of input data and voting of output data. IO activity is slaved to timer-based interrupts on the VG to reduce jitter, the temporal variance with which IO devices are accessed.

The IO services are logically segmented into two functional modules: the IO User Interface and the IO Communication Manager (Figure 3-9). Applications engineers use the IO User Interface to define the required IO activity during the specifications phase.

During the execution phase, the IO Communication Manager controls the processing of the IO requests.

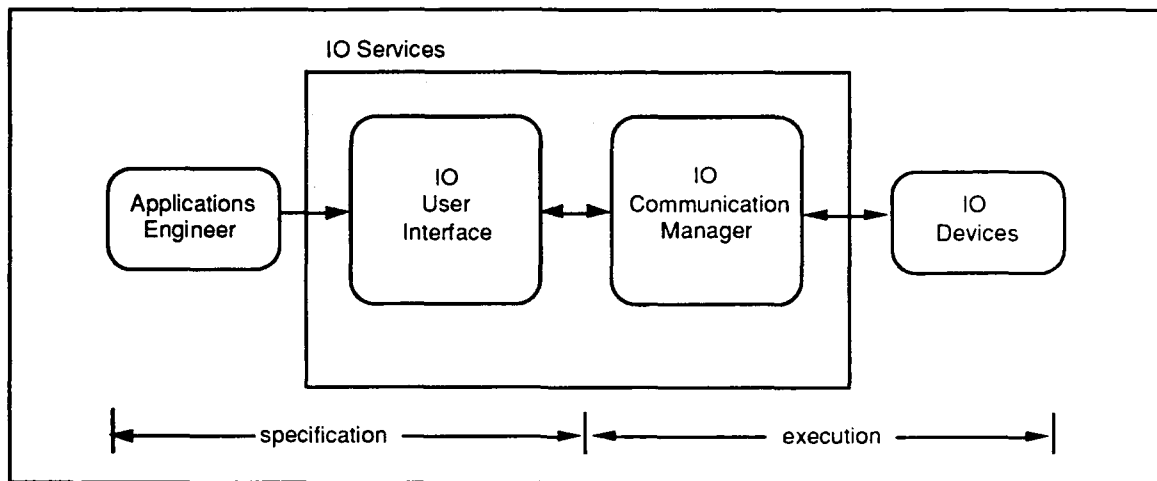


Figure 3-9. FTTP IO Services

The IO User Interface and IO Communication Manager are dependent processes, as shown in Figure 3-10. The Interface interacts with the application tasks to create an IO request database. Further, the Interface and Communication Manager exchange control and status information; the output data and control commands are destined for the IO devices while the input and status data are sent to the application tasks. Additionally, the Communication Manager retrieves information from the IO request and IO device databases and interchanges data with the IO devices.

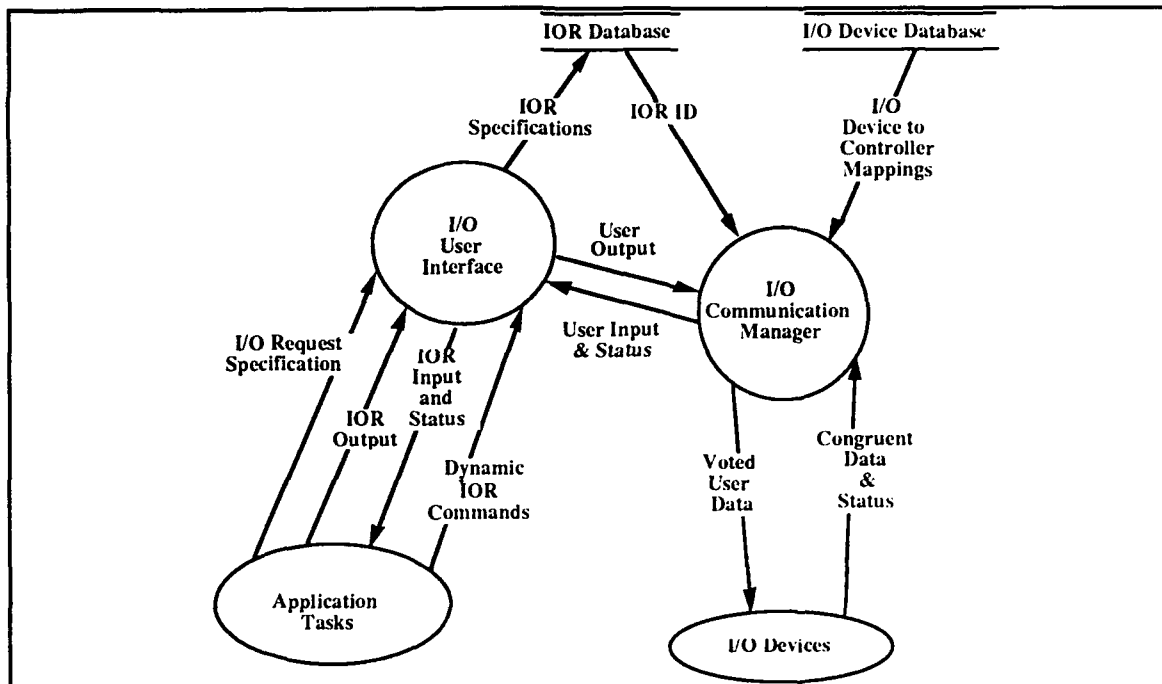


Figure 3-10. The IO User Interface and IO Communication Manager

The IO User Interface is detailed in Section 3.2.4.1 while the IO Communication Manager is described in Section 3.2.4.2.

3.2.4.1 IO User Interface

The FTPP IO User Interface allows the applications designer to specify the IO activity in a straightforward manner. The IO Services can either communicate directly or indirectly with IO devices (sensors and actuators). Direct communication is achieved by sending data and command information immediately to the device. Indirect communication utilizes an IO controller to access a device. This intervening mechanism accepts data and control commands from the VG and then manages the IO operation.

The IO Services support two general types of IO activity: sequential and concurrent. *Sequential IO* requires that the VG completely supervise the activity; that is, it must block itself until the IO operation has finished. Accordingly, the VG and the IO devices are tightly synchronized during the IO activity. This is necessary to communicate with IO controllers or devices that have limited processing capabilities such as A/D converters or "dumb terminals".

Alternatively, *concurrent IO* allows the VG to perform other tasks while the IO is being processed. The VG downloads data to the controller, sends an "start" command, and then executes another process. After the IO has completed, the VG collects the

resultant input data. The concurrent IO capability is provided to maximize FTPP's processing throughput. To permit this parallel IO-VG processing, smart hardware such as an Ethernet or 1553 controller is necessary.

The applications engineer defines the required IO activity. This is accomplished by specifying one or more IO requests.

3.2.4.2 IO Communication Manager

The FTPP IO Communication Manager supervises the execution and processing of the IO requests. It involves three key components: the IO Dispatcher, the IO Source Congruency Manager task, and the IO Processing tasks.

The IO Dispatcher manages the execution of the IO requests whereas the IO Source Congruency Manager and the IO Processing tasks exchange data among all members of the Virtual Group (VG), perform the error detection processing and return the data and status information to the application tasks. The three IO Communication Manager processes are discussed in the following paragraphs.

IO Dispatcher

The IO Dispatcher is a task on the VG that manages the execution of the IO instructions that cannot be interrupted. For the FTPP, two types of nonpreemptible instruction sequences exist: (1) the execution and reading of sequential IO; and (2) the execution of concurrent IO.

Sequential IO must be carefully controlled by the VG, because the associated destination IO devices have limited processing and storage abilities. Furthermore, applications that utilize sequential IO often require that data be sent or received quickly and in autonomous batches. If the VG is interrupted, then the IO operation could be delayed considerably. Thus, the execution of sequential IO can not be preempted. Additionally, since these IO devices have minimal memory capabilities, the input and status data for each transaction must be read before a subsequent transaction can be executed. Therefore, the reading of sequential IO also cannot be interrupted.

In contrast to sequential IO, concurrent IO is managed by an intelligent IO Controller (IOC), permitting the IOC and VG to run in parallel. The VG, however, must initiate the IO activity by sending a sequence of "start" instructions to the IOC. This sequence can not be interrupted if the IO requests are to execute correctly. Accordingly, the IO Dispatcher must initiate all concurrent IO.

To ensure nonpreemption, the IO Dispatcher must complete in less than 10 msec, which is the length of a minor frame. Thus, the application must design and organize its

nonpreemptible IO activity such that the IO Dispatcher does not exceed this constraint. In addition, the Dispatcher cannot be interrupted by other IO activity (because it would be delayed and then possibly preempted); thus, it must have the highest priority of the IO tasks.

The IO Dispatcher is scheduled by the Rate Group Dispatcher every minor frame. Since the type and amount of IO activity typically varies with each frame, the minor frame number must be determined every time the task is executed.

Once the frame number is identified, the IO Dispatcher executes the associated IO requests. The concurrent IO is executed before the sequential IO. This allows the VG to execute and process the sequential IO while the associated IO controllers are processing the concurrent requests. Some IO requests may be comprised of both sequential and concurrent IO chains (referred to as "mixed IO requests"). They are executed by the IO Dispatcher after the concurrent IO requests but before the sequential IO requests. This allows the mixed IO chains to be executed nearly simultaneously while not blocking the execution of the concurrent IO requests.

IO Source Congruency Manager

The IO Source Congruency Manager (IOSC) ensures that each member of a fault-masking group receives a copy of any input IO data read by any other member of that VG. It exchanges this information among each member of that VG via messages sent through the FTPP communication services. If the Virtual Group performing IO is a simplex, then the data does not need to be exchanged (since there is no one to exchange it with), and the execution time of the IOSC task will be minimal. The IOSC executes as an RG4 task.

IO Processing Tasks

The IO Processing (IOP) tasks are used by redundant IO applications. The IOP tasks have two primary responsibilities: (1) filtering the multiple copies of input data to come up with a single value, which is returned to the application; and (2) performing error detection routines and returning error status information to the application.

Data filtering is necessary when multiple input values are not expected to be precisely equivalent. This can occur when reading the output of an analog device. Since multiple readings will result in similar (but not necessarily exact) input values, some method for determining what value to use is needed. Many algorithms exist (average value, midpoint value, etc.), and the algorithm implemented is obviously very dependent upon the application.

The IOP tasks are also responsible for detecting any errors that may be present in a redundant IO Controller. If one of the multiple copies of input data greatly varies from the others, the IOP tasks will report this information to the Fault Detection, Identification and Recovery system which will perform any necessary reconfiguration.

There are four IOP tasks, one for each Rate Group.

Though the IOP task design is complete, the current implementation is only skeletal. Neither the filtering algorithm or the error detection code has been implemented.

3.2.5 Fault Detection, Identification and Recovery (FDIR)

The FTTP uses hardware redundancy with fault detection and masking capabilities to provide fault tolerance. This inherent fault detection capability is supplemented with traditional self-test methods to increase the coverage of faults.

The fault tolerance provided by the hardware is enhanced by the Fault Detection, Identification and Recovery (FDIR) functions which are part of the operating system. While the hardware alone in the FTTP could sustain one fault, the FDIR software allows it to sustain multiple successive faults by identifying a faulty component and masking it from system operations. Consequently, the primary purpose of FDIR is to maintain correct operation in the presence of hardware faults. To achieve this, FDIR has four main functions:

- testing of FTTP components, i.e., initiating various test procedures in order to uncover hardware failures.
- identifying a failed component, i.e., detecting a fault, isolating it to a single component and disabling the faulty component.
- performing a remedial operation, i.e., initiating a recovery operation commensurate with system requirements.
- performing transient fault analysis, i.e., determining whether the error was due to a transient fault.

FDIR for the FTTP is composed of two tasks: Local FDIR which executes on each VG, and Global FDIR which executes on a specially designated VG. *Local FDIR* has the responsibility for detecting and identifying hardware faults in the PEs of its VG and disabling their outputs. *Global FDIR* is responsible for the collection of status from the Local FDIR, and it is also responsible for the detection, identification and masking of NE faults and optical link faults. It resolves conflicting local fault identification decisions, disambiguates unresolved faults, correlates transient faults, and handles VG failures. When a faulty component has been identified, Global FDIR initiates an appropriate

recovery strategy which attempts to compensate for the loss of a component. Both of these FDIR tasks are described in the following paragraphs.

3.2.5.1 Local Fault Detection, Identification and Recovery

After the Network Elements have synchronized with each other the FTPP operates as a fault tolerant system which provides fault tolerant communication mechanisms to processing entities referred to as Virtual Groups. Using these communication mechanisms each Virtual Group will exercise some level of fault detection and identification capabilities for identification of failures among its processors. Simplex Virtual Groups may perform only processor self-testing. Fault masking groups (triplex or quad VGs) can not only perform various levels of testing (unlike simplexes), but can also unequivocally diagnose a failure in a constituent processor. The fault masking group maintains correct operation even when one of its members has failed. Furthermore, it may initiate certain recovery options.

3.2.5.2 Global Fault Detection, Identification and Recovery

The Local FDIR function executing in a Virtual Group monitors itself and performs some recovery operation which directly affects itself. However, to monitor the FTPP system globally and also to determine the health of shared components such as the Network Elements, a Global FDIR is necessary. The Global FDIR executes on a single fault masking group and is responsible for high-level testing of the FTPP, such as a poll of all Virtual Groups within the system. This is particularly important when a simplex Virtual Group exhibits faulty behavior. Since a simplex cannot mask itself out of the system configuration via configuration table updates, the Global FDIR assumes this responsibility. In addition, some recovery options require global information regarding system resources; this information is unavailable to the Local FDIR functions.

3.2.6 Overview of a Minor Frame

A simplified description of the sequence of events occurring within a minor frame (a single frame within one Rate Group) of a single VG is depicted in Figure 3-11. The frame begins with a Frame Timer interrupt which is generated by a crystal oscillator resident on each member of the VG. Immediately after the Frame Timer interrupt, the VG synchronizes its members using a synchronizing act as described in [Har87], and sets up the Frame Timer interrupt for the next minor frame.

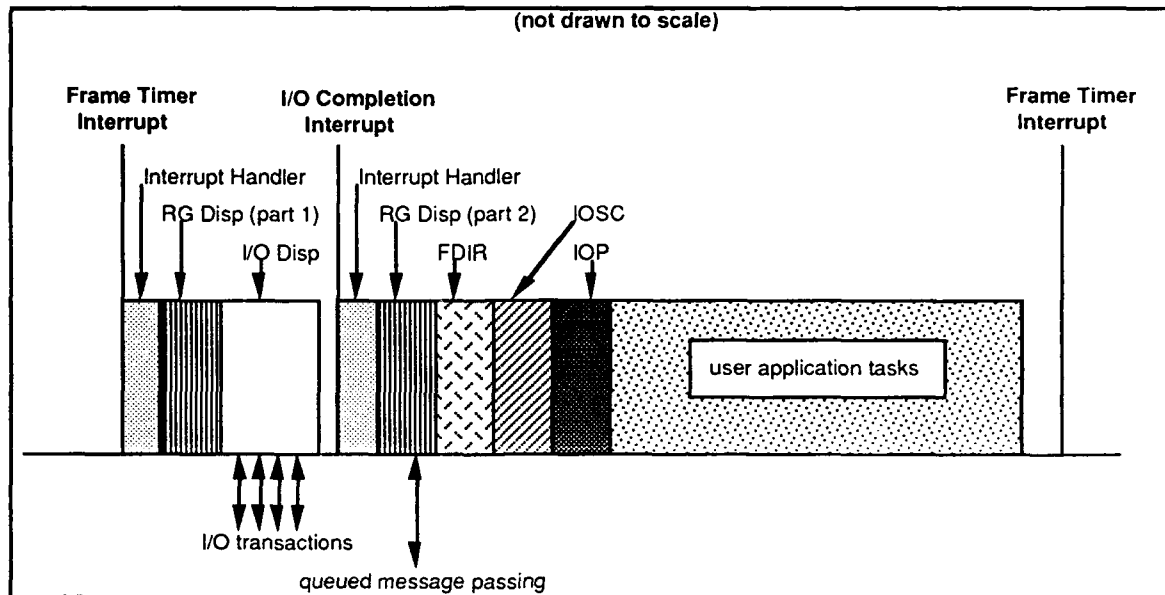


Figure 3-11. Overview of Minor Frame

The RG Dispatcher runs in two parts. The first part executes immediately after the Frame Timer interrupt and is responsible for checking for task overruns, scheduling the IO Dispatcher, and other minor activities. The second part executes after the IO Completion interrupt and is responsible for checking for IO Dispatcher overruns, sending queued messages to the Network Element, and scheduling RG tasks which are to be started in the current frame.

The IO Dispatcher performs all IO activity as close as possible to the synchronizing act to minimize IO jitter. For IO performance reasons, it is possible for each member of a VG to perform different IO transactions and thus not be synchronous after performing such operations. Therefore an IO Completion interrupt is scheduled on all VG members at a user-definable time after the Frame Timer interrupt in order to snap them back into synchronization. The Frame Timer - IO Completion interrupt interval may vary for each frame based on the IO transactions performed in that frame, and it is determined by the most lengthy set of transactions the VG's members must perform. This interrupt is also generated by a crystal oscillator on each VG member.

After the IO Completion interrupt another VG synchronization is performed by the RG Dispatcher, and messages previously queued for transmission by Rate Group tasks which completed in the prior frame are transmitted to the Network Element. Messages are also read from the NE to the VG at this time. The FDIR task is scheduled after message passing, followed by the IO Source Congruency (IOSC) and Redundancy Manager and IO Processing (IOP) tasks. These IO tasks are responsible for transmitting single-source

input data from one member of the VG to the others, IO error processing, and deriving and formatting a known good copy of redundant input data for delivery to the destination application task. The IO Processing task is also responsible for transmitting predetermined input data from one VG to another. After the IO tasks execute, the application tasks are scheduled and execute according to the Rate Group scheduling paradigm until the next Frame Timer interrupt occurs.

Chapter 4

Performance Models

Two aspects of FPHP system performance are of special importance. The first of these is the operating system overhead. Due to FPHP's real-time constraints, the overhead associated with the operating system (OS) tasks needs to be accurately predicted to ensure sufficient time exists for the execution of user application tasks. The second area of concern is contention for Network Element services by the Processing Elements (PEs). Since up to eight PEs may be served by one NE, the PEs have to contend with each other for NE service. This contention results in decreased performance, as well as variable execution time.

Because of their importance to FPHP system performance, analytical models for both the operating system overhead and Network Element contention are developed. This chapter presents descriptions of each model. A high-level description of the OS overhead model is given in Section 4.1. Because the OS overhead model uses empirical performance data, which is presented in Chapter 6, a detailed description of the overhead model is not given until Chapter 7. Section 4.2 describes the Network Element contention model and presents simulation results.

4.1 Operating System Overhead Model

This section gives a general overview of the model for the overhead associated with FPHP operating system tasks. Figure 4-1 shows the operating system tasks associated with each minor frame.

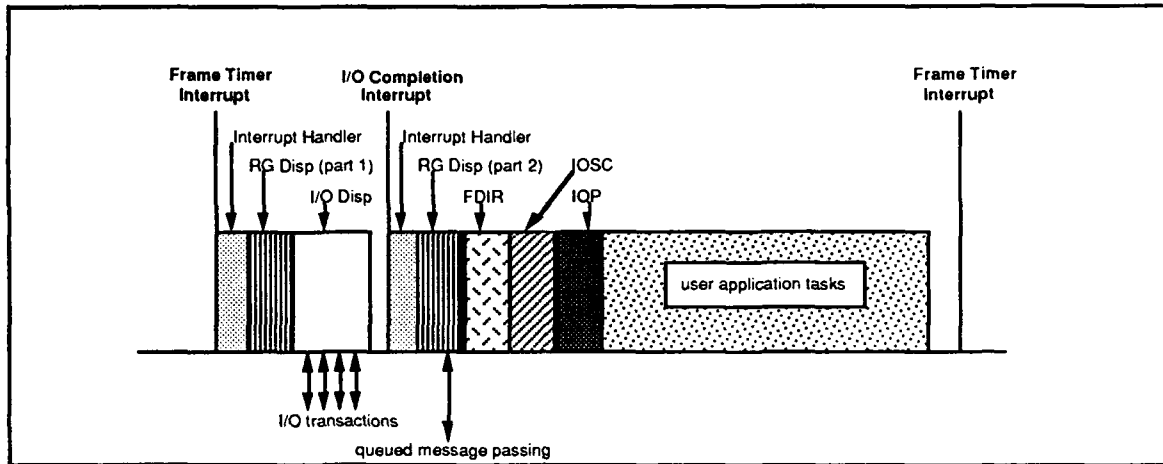


Figure 4-1. Minor Frame Overview

The overhead required by system resources within each minor frame is the sum of the execution times for each of the following operating system tasks: interrupt handler (IH), Rate Group Dispatcher (RGD), IO Dispatcher (IOD), Fault Detection Identification and Recovery (FDIR), IO Source Congruency Manager (IOSC), and IO Processing (IOP). This overhead is represented by the following equation:

$$OH = IH_1 + RGD_1 + IOD + IH_2 + RGD_2 + FDIR + IOSC + IOP$$

A description of each of these eight overheads follows.

4.1.1 Interrupt Handler (IH₁) Overhead

The overhead associated with the first interrupt handler (IH₁) is given by the following equation:

$$IH_1 = (\text{time to update clock}) + (\text{time to schedule next interrupt}) + (\text{time to scoop messages})$$

The time needed to update the system clock and to schedule the I/O Completion Interrupt is constant, and should be relatively small. Both these events are executed in assembly language routines. The time to scoop messages (Section 3.2.2.4) is a function of the number of packets that arrived in the processor's receive queue since the last time a scoop was executed.

4.1.2 Rate Group Dispatcher - Part One (RGD₁) Overhead

The time needed to execute the first part of the Rate Group Dispatcher (RGD₁) can be summarized with the following equation:

$$\begin{aligned} RGD_1 = & \text{(time to update congruent time)} + \text{(time to check for RGD}_2 \text{ overrun)} + \\ & \text{(time to check for task overruns)} + \text{(time to set up next RG interval)} + \\ & \text{(time to schedule IOD)} \end{aligned}$$

With the exception of checking for task overruns, all the components of RGD₁ are constant. Checking for task overruns is a function of the number of tasks which were scheduled to suspend themselves during the previous minor frame.

4.1.3 IO Dispatcher (IOD) Overhead

The overhead associated with the IO Dispatcher task is given below:

$$\begin{aligned} IOD = & \text{(time to increment frame counter)} + \text{(time to start IOR execution)} + \\ & \text{(time to wait for IO to complete)} + \text{(time to read input data)} \end{aligned}$$

The time to increment the frame counter is constant and is negligible (one 'add' statement in Ada). The other constant is the time to wait for IO to complete. This is simply a wait of a duration chosen by the application programmer to ensure that any outward-bound IO is finished before any attempt is made to read incoming IO data. If IO is strictly incoming or strictly outgoing, this wait can be minimal. The wait is really only necessary for IO that sends out data to some device and then awaits a reply (in the form of incoming data) from that device.

The two remaining constituents of the IOD overhead are variable and depend on the type and amount of IO activity to be performed during a given minor frame. The time to start the execution of IO requests depends on the number of IO requests scheduled to run this minor frame that have outgoing data, and it also depends on the amount of data each IO request sends. Finally, the time to read input data depends on the number of IO requests which have incoming data and on the amount of incoming data.

4.1.4 Interrupt Handler (IH₂) Overhead

The overhead associated with the second interrupt handler is the same as that given for the first interrupt handler (Section 4.1.1) and is repeated below:

$$IH_2 = (\text{time to update clock}) + (\text{time to schedule next interrupt}) + (\text{time to scoop messages})$$

Even though both instances of the interrupt handler are modeled by the same equation, in general the overheads associated with IH₁ and IH₂ are different. This is because the time to scoop messages will vary with the number of packets present in the receive queue for the processor. Typically, the time interval between the occurrence of IH₁ and IH₂ is less than the time duration from IH₂ to the next occurrence of IH₁. This implies that more packets may arrive in the receive queue during the interval from IH₂ to IH₁, and therefore the message scoop time should generally be longer for IH₁ than IH₂.

4.1.5 Rate Group Dispatcher - part two (RGD₂) Overhead

The execution time for the second part of the Rate Group Dispatcher (RGD₂) can be quantified as follows:

$$RGD_2 = (\text{time to update congruent time}) + (\text{time to check for } RGD_1 \text{ overrun}) + (\text{time to check for IOD overrun}) + (\text{time to send queued messages}) + (\text{time to update queues}) + (\text{time to schedule RG tasks}) + (\text{time to increment frame count}) + (\text{time to set up IO interval})$$

Most of the constituents of RGD₂ listed above involve simple housekeeping chores and have constant execution times. The three areas of interest are the time to send queued messages, the time to update queues, and the time to schedule Rate Group tasks. The *time to send queued messages* is a function of the number of tasks that suspended themselves during the previous minor frame and the number of message packets that each task had enqueued since the last time its queue was sent. The time to send queued messages also varies with the amount of contention for NE service. The OS overhead model assumes no contention; the effect of contention on the `send_queue` time is explored in Section 4.2. The *time to update a task's queue* is a function of the number of packets received and the number of packets read since the last time the queue was

updated. The *time to schedule the RG tasks* is a function of the number of RG tasks that are to be scheduled this minor frame.

4.1.6 Fault Detection Identification and Recovery (FDIR) Overhead

The overhead of running the Local FDIR task is the same as enqueueing a one-packet message; this is the entirety of the Local FDIR task.

$$FDIR = (\text{time to enqueue message to System FDIR task})$$

The Local FDIR task simply sends a message to the System FDIR task, and the time needed to enqueue a one-packet message is constant.

4.1.7 IO Source Congruency Manager (IOSC) Overhead

The IO Source Congruency Manager ensures all members of a redundant virtual group receive a copy of any input read by another member. The overhead associated with the IOSC task is given below:

$$IOSC = (\text{time to exchange input data among VG members})$$

The time to exchange the input data depends on several factors. The most important factor is whether or not any input data was read at all. If no data were read in, there is none to exchange, and the IOSC overhead will be minimal. The IOSC overhead increases as the amount of incoming data increases. Also important in determining the execution time of the IOSC task is the number of IO requests executed during the current frame that involved incoming IO data.

4.1.8 IO Processing Task (IOP) Overhead

The IO Processing task is responsible for ensuring that all members of a VG performing redundant IO agree with one redundant input value. This usually involves some data smoothing or averaging. For instance, the average of three sensor values could be used as the single input value. This processing or smoothing of the input data is specific to the application, and can vary widely as far as execution time is concerned. The IOP overhead is given below:

$$IOP = (\text{time to process input data})$$

Note that there are four IOP tasks, one for each Rate Group.

4.1.9 OS Overhead Summary

In summary, the total OS overhead for a minor frame is given by:

$$OH = IH_1 + RGD_1 + IOD + IH_2 + RGD_2 + FDIR + IOSC + IOP$$

Many components of this equation have execution times that are constant. Other components are variable and depend upon such factors as the system configuration or amount of message traffic. Looking at the overhead in this manner, the total OS overhead can be written as a constant value plus some functions of different system parameters. This equation is given below:

$$OH = \text{Constant} + f(\text{number_of_tasks}) + g(\text{number_of_message_packets}) \\ + h(\text{amount_and_type_of_IO})$$

The total overhead is a function of the number of tasks and of the distribution of these tasks among the four Rate Groups. It is also a function of the number of message packets that each task sends. In addition, the amount of overhead is a function of the type and quantity of IO activity.

One aspect of system performance that is not accounted for in the OS overhead model presented in this section is contention for NE service by PEs. This occurs when more than one PE is serviced by a particular NE. The OS overhead model was developed using a simplex processor which did not have to contend for NE service; only one prototype NE and a limited number of PE boards were available when the overhead model was developed. In this regard, the OS overhead model provides a lower bound on the amount of system overhead. The effect of contention on system performance is examined by the model presented in the following section.

4.2 Contention Model

The second model developed to analyze FPHP system performance examines the contention among PEs for NE services. Each processor sends its queued message packets during the second part of the Rate Group dispatcher. If several PEs are sending packets at the same time, they must wait for the NE, which services the PEs in round-robin fashion. This contention results in performance delays because the PEs busy-wait (i.e., continuously poll the NE to see if it is ready) for each packet to be serviced before enqueueing the next one. Since performance delays can be critical in real-time systems, it is important to understand the effects of this contention on system performance by developing an analytical model.

This section describes the model developed to analyze the delay times associated with contention. This contention model can be used to determine the busy-wait delay for each PE as a function of the phasing of the eight PEs. Section 4.2.1 gives an overview of the contention model, and Section 4.2.2 describes the simulation of the system, based on the contention model. Finally, the results of the simulation and some concluding comments are presented in Section 4.2.3.

4.2.1 The Model

The contention model is developed without using traditional queueing theory concepts. Due to the periodic, real-time characteristics of the operating system, NE contention can not be modeled using a simple Markovian birth-death queueing model; the PEs send their message packets once during each Rate Group frame, so the assumption that packet arrivals are exponentially distributed is not valid for FPHP. Queueing models with generalized distributions could be used, but the mathematical complexity of these models quickly becomes excessive. Instead, we have developed a contention model based on empirical performance data.

The following three sections describe the model used to demonstrate how contention affects the amount of time needed by a PE to send its message packets to the NE. First, the PEs' use of the NE to vote and deliver messages is described. Then, an example is given demonstrating how contention arises when more than one PE is sending packets at the same time. Finally, a description of the assumptions used to simplify the model is given.

4.2.1.1 Processing of Message Packets

Tasks that wish to send messages must first decompose each message into 64-byte packets and place them in a queue in the PE's local memory. When the Rate Group Dispatcher (part two) executes during the following minor frame, these message packets are sent to the NE one at a time during execution of the `send_queue` procedure. The queued packets are sent to the PE's transmit queue, which is located in a dual-port RAM memory shared between the PE and the NE. This is shown in Figure 4-2. Each of the possible eight PEs connected to a NE has its own transmit queue. The packets are sent one at a time because the capacity of the queue is only one packet. The PE can not send a second packet to the transmit queue until the first one has been removed by the NE. If the PE has more than one packet to send and the transmit queue is full, the PE must wait until the NE empties the transmit queue before the PE can transfer the next packet to the transmit queue.

The NE is notified of a packet arrival from the PE in the transmit queue via a System Exchange Request Pattern (SERP). The SERP is a string of bytes describing the current state of the transmit and receive queues for each processor in the system. When a packet arrives in a transmit queue, a status bit is set, and the next SERP will indicate the arrival of the packet. The NE is not aware of the presence of the packet until the SERP is processed. Therefore, there will be a delay from the time when the packet arrives in the transmit queue until the NE has processed the SERP. If the status bit is set immediately before the SERP is exchanged, the delay will be minimal and will equal the amount of time needed to process a SERP (approximately 16 μ sec). If the status bit is set just after a SERP was sent, the delay will be maximal and will equal the time needed to process two SERPs (approximately 32 μ sec).

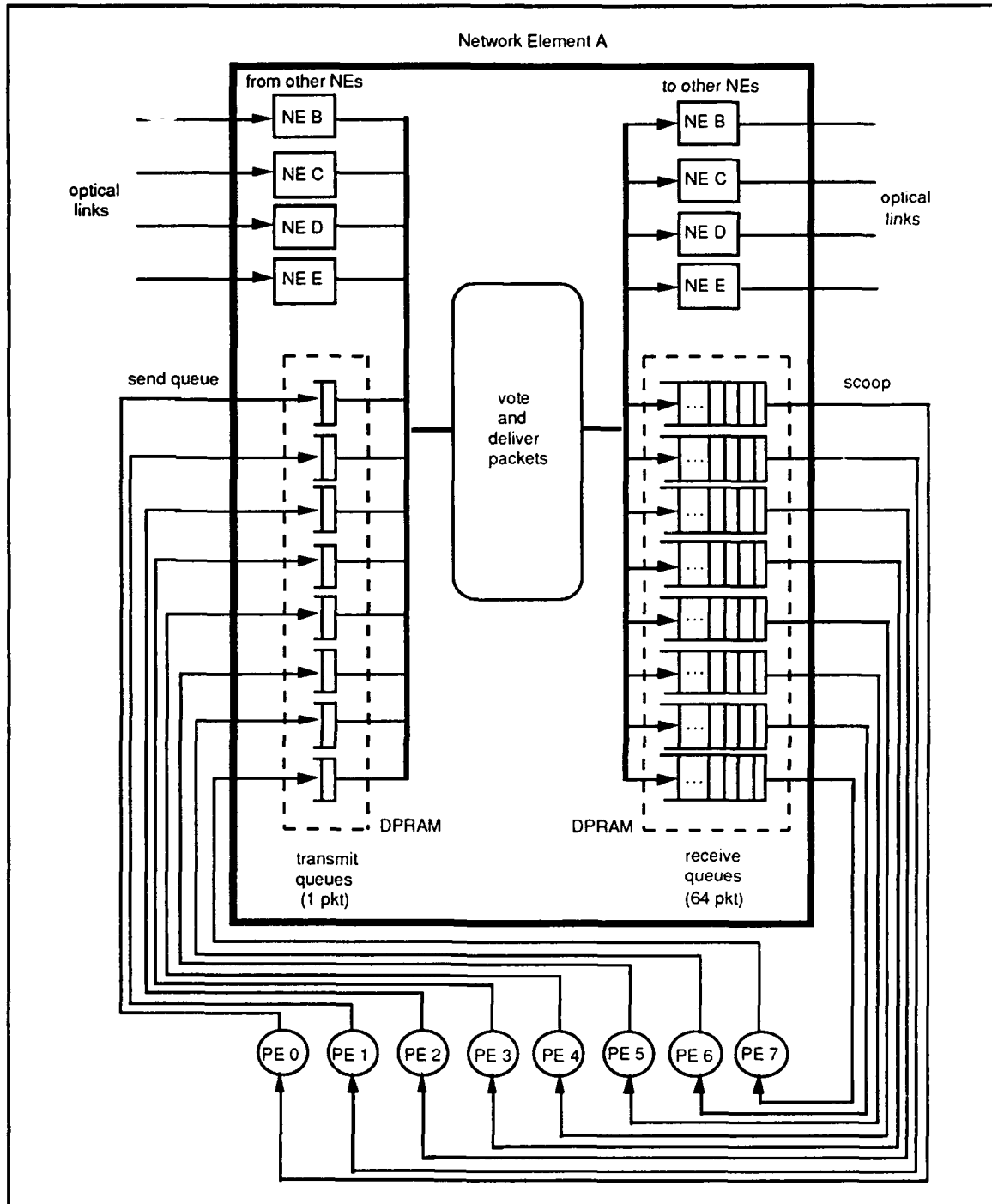


Figure 4-2. Message Packet Processing

Once the NE is aware of the arrival of a packet, it can begin to process it. The processing done by the NE depends on the class of the packet being transmitted (Section 3.1.1.1). A class 0 message requires minimal processing time since no data is involved. Class 1 messages (voted messages) are typically the most common type of message.

Processing a class 1 packet involves receiving redundant copies of the packet from the other PEs (connected to different NEs) in the virtual group. These copies are voted, syndrome information is attached, and the voted copy is delivered to the destination PE. A class 2 packet (source congruency message) undergoes a two-round exchange with the other NEs before voting and delivery.

After processing a packet, the NE delivers the packet by placing it in the appropriate receive queue, as shown in Figure 4-2. Each PE has its own receive queue located in the dual-port RAM shared among the NE and its eight PEs. The capacity of each receive queue is 64 packets. Packets are transferred to the destination PE via a `scoop` call, and the packets are reassembled into messages when the destination VG executes a `retrieve_message` system call (Section 3.2.2.4).

4.2.1.2 Contention for NE Services Among Two or More PEs

As mentioned earlier, contention during message packet transmission occurs if more than one PE is sending packets at the same time. The PE must wait for the NE to clear its transmit queue before the next packet can be transferred to the queue. If only one PE is attached to the NE, there is no delay. If more than one PE is assigned to the NE, the delay is a function of how many other PEs are sending packets at the same time.

An example demonstrating how the busy-wait time can vary is given in Figure 4-3. The time needed for a PE to transfer a packet from its local memory to the transmit queue is constant (approximately 57 μ sec). As shown in Figure 4-3, the NE will be informed that PE_0's transmit queue is full once the NE has processed the SERP containing this information. In the figure, the transmit queue was filled at time t1, and the SERP processing was completed at time t2. Once the SERP is processed, the NE is able to process the packet, and the PE is then able to transfer the next packet when the packet processing is finished and the transmit queue is cleared at time t3. Thus, PE_0 had to wait from time t1 to time t3. Notice that once PE_0 has filled the transmit queue a second time, it has a much longer delay before it can transfer a third packet. This is because when the queue is filled at time t4, the NE is busy processing a packet from PE_2 and thus can not immediately empty PE_0's transmit queue. It is not until time t6 that the NE finishes processing the SERP that indicates PE_0's queue is full. Since the NE services the PEs in round-robin fashion, PE_0 will have its queue emptied at time t7. Figure 4-3 shows that the t4-t7 time interval is greater than the t1-t3 time interval. The amount of time spent by PE_0 waiting for NE access increased because it had to contend with other PEs for NE service. Figure 4-3 also indicates the phasing in the system for this example.

The phasing between two PEs is the difference in time between the start of each of their minor frames.

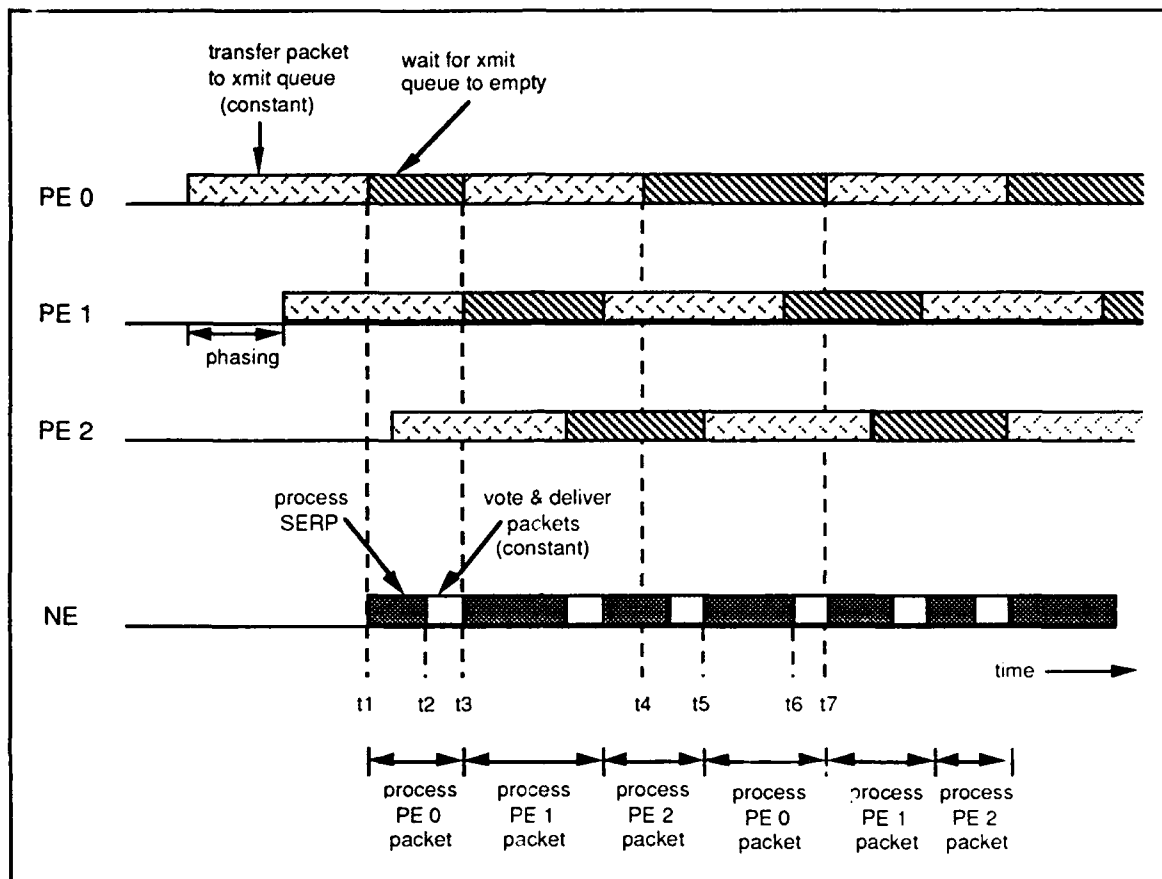


Figure 4-3. Contention Timeline

The result of this increased wait between packets is an increase in the amount of operating system overhead and a corresponding decrease in the amount of time available for executing application tasks. The OS overhead increase is a result of the increased time needed for the Rate Group Dispatcher (part two) to execute, which is a result of the increased amount of time spent executing the `send_queue` procedure.

4.2.1.3 Simplifying Assumptions

To facilitate the modeling and simulation of this system, some simplifying assumptions have been made. These assumptions are listed below, and a justification for each is given.

The time delay for the NE to realize that a transmit queue has been filled is constant. As mentioned in Section 4.2.1.1, the NE is made aware of a full transmit queue when it

processes a SERP. The time delay from when the queue has been filled to when the NE realizes it, varies as a function of when the queue was filled. If it was filled just before a SERP is exchanged, the time delay is the time to process one SERP (16 μ sec). If it was filled just after a SERP was exchanged, the time delay is the time to process two SERPs (32 μ sec). Thus, the time delay is always somewhere between 16 μ sec and 32 μ sec. To simplify the simulation, we will assume the time delay is a constant and equals 25 μ sec.

The amount of time needed for the NE to vote and deliver a packet is constant. This assumption is a combination of the assumption that only class 1 (voted messages) packets are transmitted and the assumption that the redundant members of the virtual group are tightly synchronized. Since the vast majority of system message traffic is expected to be class 1 messages (possibly over 90%), we will assume that all packets are class 1 in order to simplify the model. Then, we will also assume the NE processing time per class 1 packet is constant. The only variance that could exist is due to any time difference in the arrival of redundant copies of the packets to be voted. Because the processors are only loosely coupled, individual copies of the packets may arrive at different times. However, the processors are synchronized just before the Rate Group Dispatcher (part two) is executed, so the skew among the processors should be minimal and can be ignored. Therefore, the NE processing time for packets will only consist of the time needed to vote the packet, attach syndrome information, and deliver the packet. This time is assumed to be constant and equal to 10 μ sec.

Operating system overheads are identical for each PE for each frame. As will be shown in Chapter 7, the operating system overhead generally varies with the minor frame number. For example, minor frame 0 usually has the largest OS overhead since all tasks, regardless of their rate group, have suspended themselves and are ready to send queued messages. Minor frame 1 usually has a minimal overhead since only RG4 tasks can send their queues. We assume the OS overhead variance is negligible, and that it is identical for each minor frame for each PE. Therefore, each minor frame on each PE appears like every other minor frame. Without this assumption, the time within the minor frame when RGD₂ was executed (and thus when the queued packets can be sent) would vary from frame to frame and would be a function of the number of tasks, the distribution of tasks among the four rate groups, the number of packets enqueued by each task per minor frame, and the amount and type of IO performed by each task per frame.

The phasing among the eight PEs is constant. The phasing between two PEs is the difference in time between the start of each of their minor frames. In Figure 4-4, an example of phasing among eight PEs is given. The phasing between PE₀ and PE₁ is indicated in the figure. We assume that the phasing from one PE to its neighbor is the

same. The amount of phasing is important because it determines how much overlap there is when PEs are performing a `send_queue` call. The worst case, in terms of contention, would be zero phasing; then, all PEs would be sending their queues at the same time, and contention would be maximized. During simulation, the phasing is varied to note its effect on contention.

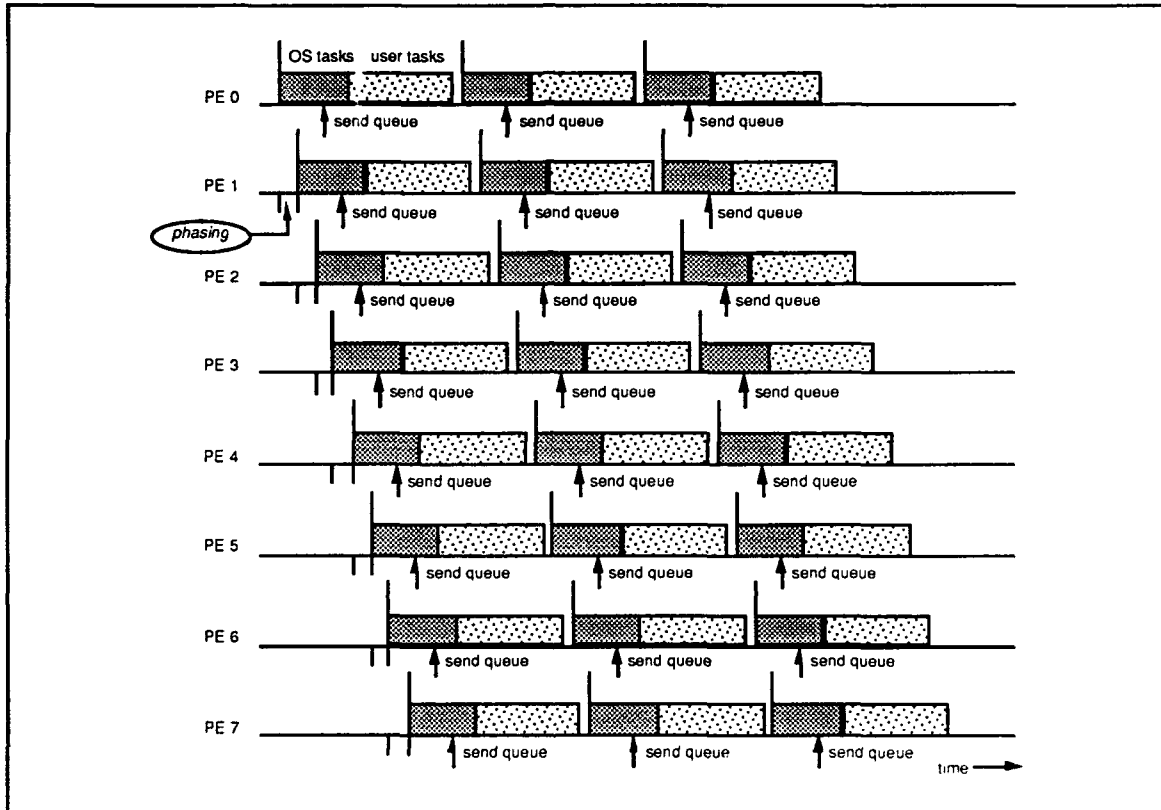


Figure 4-4. Phasing Among PEs

There is no contention for the VMEbus which connects the PEs to the NE. One detail that has been ignored so far is the bus connecting the PEs to their NE. The prototype FTPF uses VMEbus to connect the PEs and their NE, and it is possible that the PEs may have to contend for the VMEbus while transferring their packets from local memory to the transmit queue. We assume that there is no contention among PEs for use of the VMEbus. Consider the worst case scenario for data traffic over the VMEbus (zero phasing among the eight PEs). In this case, all eight PEs attempt to send a 64-byte packet over the VMEbus at the same time. Empirical performance data show it takes approximately 60 μ sec for a single PE to transfer a packet from local memory to the transmit queue. Therefore, at worst 512 bytes are being sent over the VMEbus in a 60- μ sec period, which corresponds to a data rate of 8.5 Mbytes/sec. The VMEbus has been

rated at 40 Mbytes/sec, so the worst case amount of VMEbus traffic only uses about 21% of the available bandwidth. As a result, we consider the assumption of no VMEbus contention valid.

4.2.2 Contention Simulation

To model the contention for NE services by the PEs, the system description of Section 4.2.1.2 is combined with the simplifying assumptions presented in Section 4.2.1.3 to form a simplified model used in simulations. An example timeline for the contention model is given in Figure 4-5. In this example, three PEs each send two packets to the NE for processing.

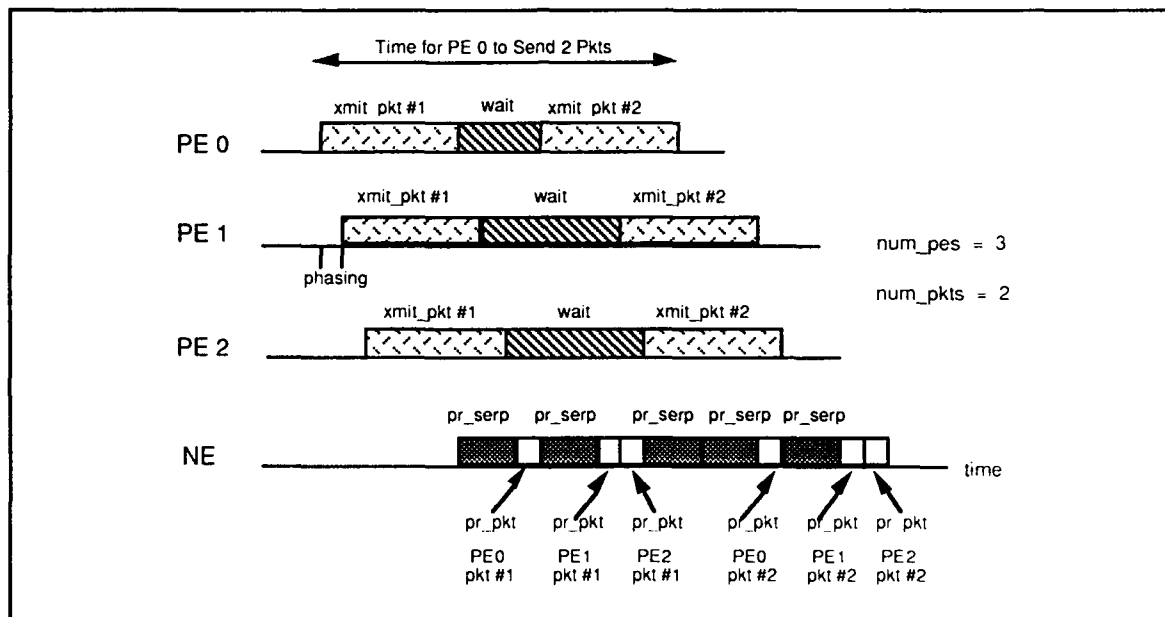


Figure 4-5. Contention Model Timeline

During simulation, a number of parameters can be varied to note their effect on the time it takes a PE to send its message packets. These parameters are listed below:

- | | |
|-----------------|--|
| <i>xfer_pkt</i> | This is the time it takes a PE to transfer a packet from its local memory space to the transmit queue located in the dual-port RAM shared by the PE and NE. The default value is 60 μ sec. |
| <i>pr_serp</i> | This is the time it takes a PE to process a SERP. This value is assumed to be constant, and the default value is 25 μ sec. |

<i>pr_pkt</i>	This is the time needed by the PE to process a packet. Processing a packet includes voting redundant copies of the packet and delivering the voted packet to its destination. The default value is 10 μ sec.
<i>num_pkts</i>	This is the number of packets each PE sends during each frame. For simulation purposes, all PEs send the same number of packets. The default value is 10 packets per PE per frame.
<i>num_pes</i>	This is the number of PEs connected to the NE. The default value is 8 PEs (the maximum possible).
<i>phasing</i>	The phasing between two different PEs is the difference in time between the start of each of their minor frames. It is assumed that the phasing among PEs is constant, as shown in Figure 4-4. The default value for phasing is 0 μ sec; this represents worst case contention.

The simulation software is written in C, and the source listing is given in Appendix B. It is menu-driven, and the user can change any of the simulation parameters he or she desires. The simulation provides the length of time needed by each PE to send the indicated number of packets under the given conditions.

4.2.3 Results of the Simulation

Of the parameters listed in the previous section, some are of more interest to application engineers than system designers. Application engineers are concerned with the number of PEs connected to a NE, the number of packets sent by each PE, and the phasing among the PEs; these are the parameters they control. Their goal is to minimize the time needed for a PE to send its packets within the time constraints of the application task. The effect of varying the number of packets sent by each PE is shown in Figure 4-6. By reducing the number of packets per PE the delay in sending the packets is reduced, and this is shown in the graph. For a given number of packets, different amounts of phasing can result in slight improvements in performance. However, the performance improvement is not very significant.

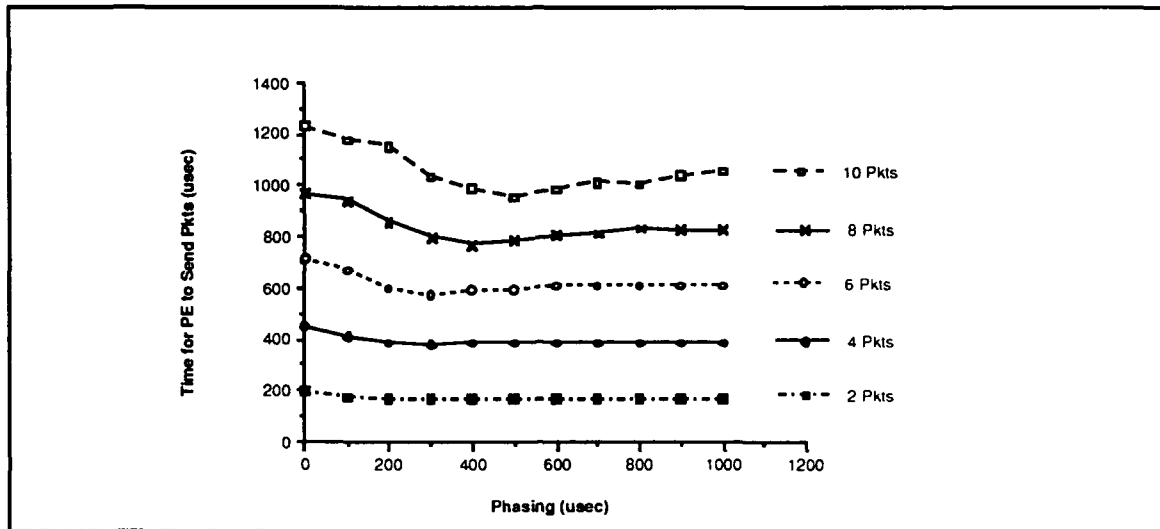


Figure 4-6. Effect of Varying Number of Packets and Phasing on Time to Send Message Packets

Another parameter of interest to the applications engineer is the effect of reducing the number of PEs connected to a NE. Simulation results showing the impact of varying the number of PEs on the time needed by each PE to send its packets are given in Figure 4-7. With small amounts of phasing, the number of PEs has some effect on the time needed to send packets. However, the improvement is not large. Consider the case of no phasing. With eight PEs, the time delay is 1230 μsec , but reducing the number of PEs to four only decreases the time delay to 1100 μsec . Reducing the number of PEs by 50% results in an improvement of only 10.6% in performance. It is also interesting to note that as the phasing increases, the effect of reducing the number of PEs becomes negligible.

Though not shown in Figure 4-7, the time to send packets for one PE is of interest because it indicates the extent to which contention can increase system overhead. With only one PE connected to a Network Element, no contention can occur; the simulation predicts a 1035 μsec time delay for one PE to send its queued packets. The worst-case contention occurs when eight PEs are connected to one NE, and the amount of time needed for a PE to send 10 packets in this configuration is 1230 μsec . Therefore, contention can increase the amount of overhead in sending packets by 18.8% compared with the case when no contention occurs.

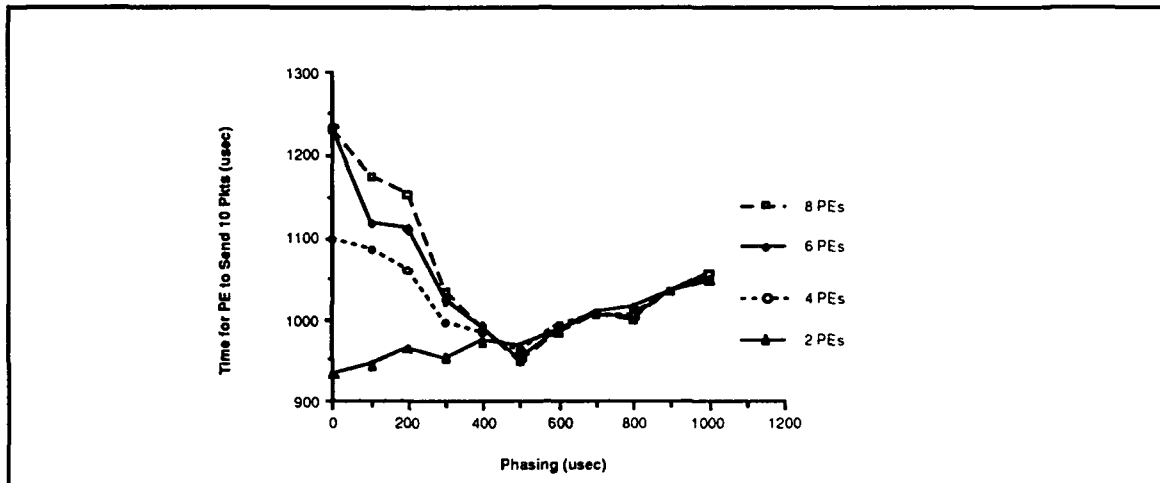


Figure 4-7. Effect of Varying Number of PEs and Phasing on Time to Send Message Packets

System designers are also interested in ways to reduce the amount of time spent sending packets. The parameters controlled by system designers include the time it takes the NE to process a packet, the time it takes for the NE to process a SERP, and the time needed by a PE to transfer a packet from its local memory space to the dual-port RAM shared by it and the NE. Figure 4-8 shows the effect of varying the process packet time and varying the transfer packet time. For a transfer packet time of 60 μ sec, an 80% reduction in process packet time (from $pr_pkt = 10 \mu$ sec to $pr_pkt = 2 \mu$ sec) results in a 26.5% performance improvement.

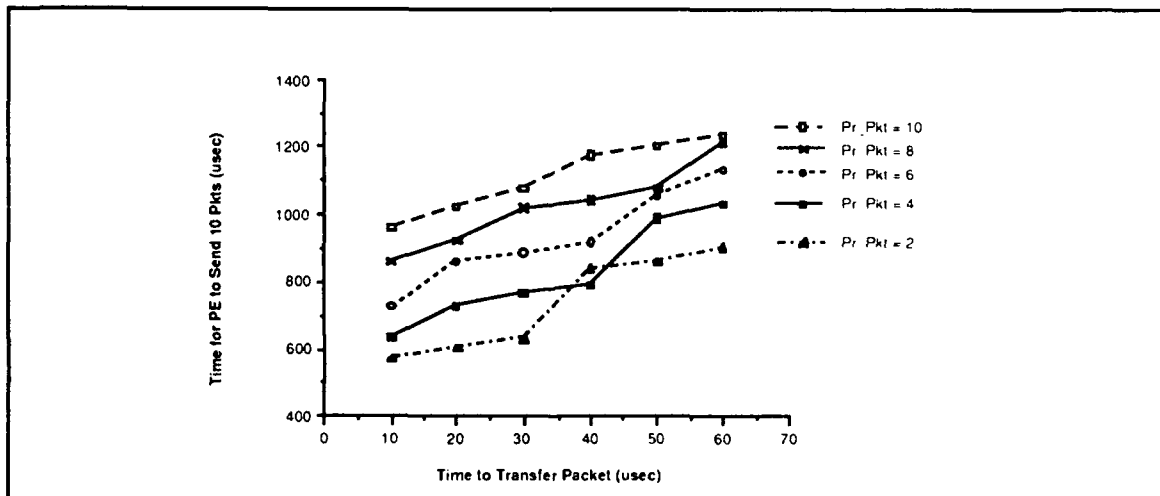


Figure 4-8. Effect of Varying Process Packet Time and Transfer Packet Time on Time to Send Message Packets

The system designer also determines the time needed by the NE to process a SERP. Figure 4-9 presents the simulation data for different values of the process SERP time. As expected, reducing the process SERP time reduces the delay needed by a PE to send its packets. With a transfer packet time of 60 μsec , a reduction in the process SERP time of 80% (from $\text{pr_serp} = 25 \mu\text{sec}$ to $\text{pr_serp} = 5 \mu\text{sec}$) results in a performance improvement of 24.5%. This is approximately the same effect as varying the process packet time from 10 μsec to 2 μsec .

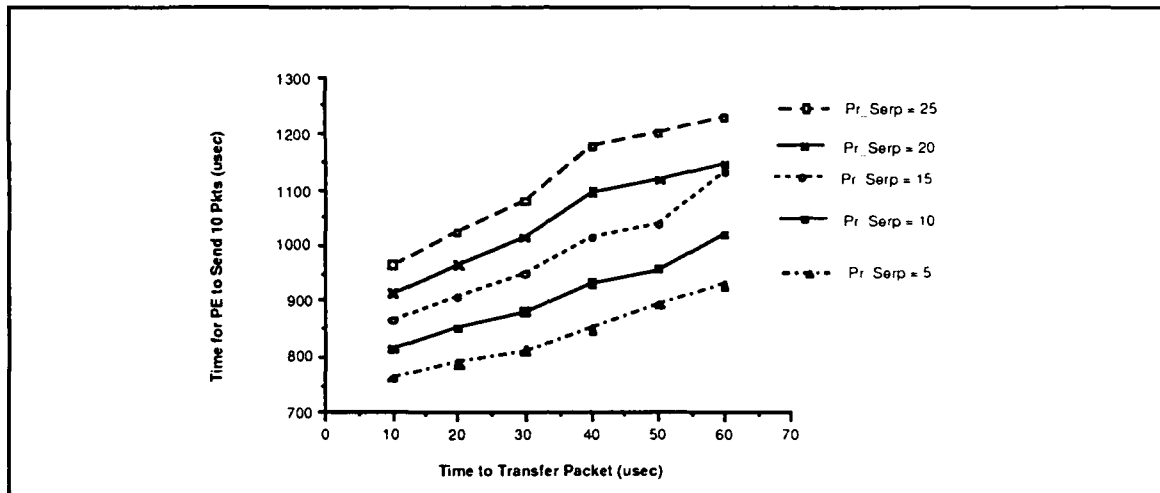


Figure 4-9. Effect of Varying Process SERP Time and Transfer Packet Time on Time to Send Message Packets

The goal of both the applications engineer and the system designer is to reduce the amount of time needed for a PE to send its message packets, even when contending with other PEs for NE service. To find out which parameter has the greatest single impact, each parameter was decreased by 50% of its default value. The results are given in Figure 4-10. This graph shows that the largest improvement in performance for a given number of packets is obtained by reducing the transfer packet time by 50%. This implies that if effort can only be spent reducing one parameter, it should be spent reducing the transfer packet time. The transfer packet time can be reduced by using direct memory access or by using a faster bus.

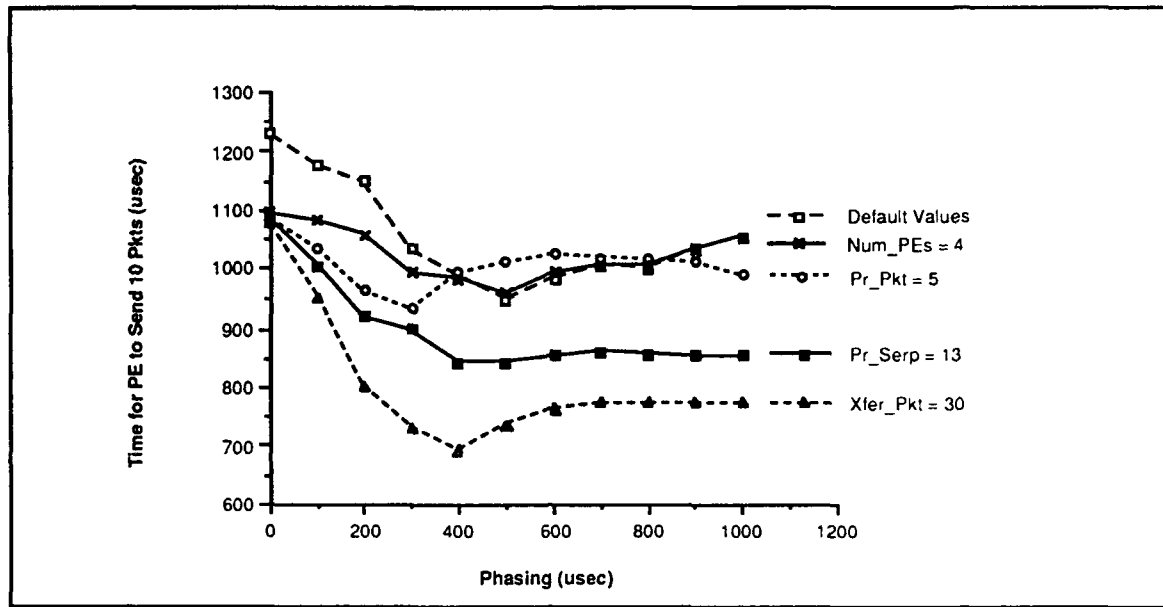


Figure 4-10. Effect of Reducing Each Default Parameter by 50% on Time to Send Message Packets

The simulation results presented in this section show that application engineers should minimize the amount of message passing in the system to minimize the effects of contention on the time needed by a PE to send queued message packets. System designers should reduce the time needed by a PE to transfer a message packet from its local memory space to the dual-port RAM shared by the PE and the NE. This could be accomplished by using direct memory access to accomplish the transfer.

Chapter 5

Performance Measurement Methodology

There are numerous advantages associated with collecting measurements of system performance. First, empirical measurements can be developed into analytical models which can be used to predict system performance under various configurations and workloads. Second, the empirical performance data can be used to measure the system overhead, a parameter critical for real-time applications. Finally, when the performance measurements are collected concurrently with prototype operating system (OS) development, potential performance bottlenecks can be removed at an early and cost-effective stage of development.

Raw performance data is collected through the use of software probes. The probes are a software routine which records relevant system information, including the value of the system clock. These probes are placed around or directly inside the code of the operating system procedures of interest. During execution, the probes are activated along with the OS procedure of interest. The probes record execution times and other parameters of interest in the processor's local memory. The real-time FTPP system is not suited to perform the analysis of this raw data, so the data are transferred to a host VAX computer for reduction and analysis. The FTPP IO System Services are used to move the data, via an Ethernet link, from the PE to the host. Figure 5-1 shows the path the performance data take from initial storage in the debug log to final analysis on the VAX.

This chapter describes the methodology used to collect and analyze performance data for the FTPP. Section 5.1 describes the software probes used to collect performance data. Section 5.2 outlines the transfer of data from the FTPP to the host VAX (where it is analyzed). Section 5.3 summarizes the statistical analysis of the data performed on the VAX.

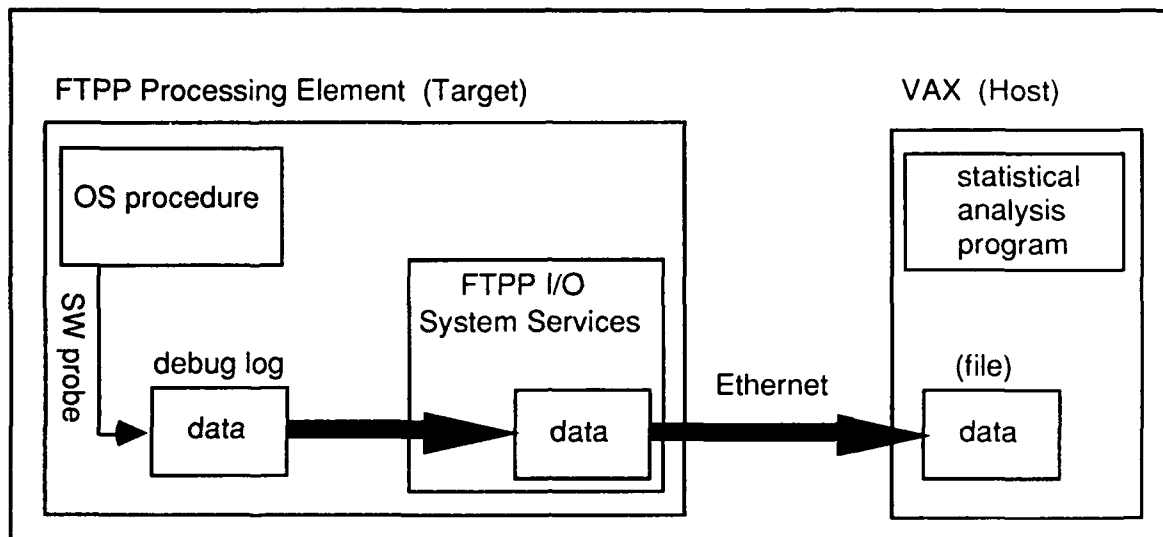


Figure 5-1. Performance Measurement Overview

5.1 Software Probes

Software probes are the basic data collection tool. A similar approach to recording performance information was taken by researchers at Carnegie Mellon University who used software "sensors" in their Parallel Programming and Instrumentation Environment [Leh89]. A description of software probe use with FTPP performance measurement is given by describing the data that is collected and then providing an example of how this data can be used to determine the time it takes the operating system to enqueue a message.

5.1.1 Description of Data Recorded by Software Probes

Software probes are the mechanisms used to collect performance data. A software probe is an Ada procedure which uses an assembly language routine to store information in an area of the processing element's memory known as the *debug log*. Each entry in the debug log contains three fields of information:

- label field
- parameter field
- timestamp field

The *label* field records a tag to the probe in the source code. Since numerous software probes are to be imbedded in FTPP procedures and tasks, it is necessary to identify the saved data with the probe which stored it. The tag in the label field uniquely identifies which probe recorded the data for that debug log entry.

The *parameter* field is used to store a value of pertinent system information. The choice of what data to store in this field depends on what aspect of system performance the probe is measuring. For example, the overhead associated with the delivery of queued message packets by the Network Element (via the `send_queue` procedure) depends on the number of packets queued. Since this is an independent variable, it is useful to record the value in the parameter field of the debug log entry. Likewise, some system overheads are a function of the minor frame number. Probes used to measure those overheads store the current frame number in the parameter field.

The final data field in each debug log entry is the *timestamp* field. The value of the system clock is automatically stored in this field each time the software probe is activated. The system clock value is a 32-bit quantity and has a resolution of 1.28 μ sec per tick. The clock wraps around to 0 after reaching its maximum value (this occurs after approximately 92 minutes).

5.1.2 Example of Software Probe Use

As an example of how the debug log entry fields are used to measure system performance, consider a method to determine the length of time for the operating system to queue a message for delivery. Software probes are inserted in an application task just prior to, and immediately after, calling the queueing procedure. This is illustrated in Figure 5-2.

```
while size < max_size loop
  debug_log(16#1111#, size);
  queue_message();
  debug_log(16#1112#, size);
end loop;
```

Figure 5-2. Placement of Software Probes

The software probe is activated by the call to the Ada procedure `debug_log`. Two parameters are passed during the call to `debug_log`. The first parameter is a number that will be stored in the label field of the debug log entry. For the first software probe shown in Figure 5-2, the hexadecimal number 1111 is used as the label (or tag). The label field for the second software probe (just after the `queue_message` procedure) contains 1112 hex.

The second data passed to the `debug_log` procedure is a variable whose value will be stored in the parameter field of the debug log entry. When collecting performance data on the time to queue a message, it is important to know the size of the message being enqueued. This information can be stored in the debug log entry's parameter field by including the variable "size" in the call to `debug_log`.

In addition, the value of the system clock at the time `debug_log` is executed is stored in the timestamp field of the debug log entry. Two consecutive debug log entries for enqueueing a 64-byte message would be similar to those given in Table 5-1.

label	parameter (size)	timestamp
1111	64	1645338
1112	64	1645449

Table 5-1. Representative Use of Debug Log Data Fields

The data contained in these debug log entries is used to determine how long it took the system to queue the message. Although the processing of the debug log data is discussed more thoroughly in Section 5.3, a brief overview of the process follows in order to explain the use of the data fields. First, the timestamp for the probe labeled 1111 is subtracted by the timestamp for the probe labeled 1112. This number is then multiplied by the resolution of the clock (1.28 microseconds per clock tick) to give the time needed to queue the message. In addition, the overhead of making the call to `debug_log` is also subtracted out. The result of these operations is the time it took to enqueue the message.

5.2 Transfer of Data from the FTPP to the Host VAX

The software probes store debug log entries in the local memory of the FTPP processing element. However, the programs written to analyze this data run on a VAX computer, so the data must be transferred from the FTPP to the host VAX for processing. The FTPP IO System Services are used to oversee the data's transfer via Ethernet to the host VAX. An IO application task consisting of an Ethernet output IO request was created to perform the transfer of debug log data from the FTPP to Ethernet. On the VAX end of the Ethernet connection is a program which continuously polls the Ethernet port for the arrival of new data. Once the data is read in, it is stored in a VAX file for off-line statistical analysis.

5.3 Data Analysis

At this point in the performance measurement process, raw performance data has been collected and transferred to the host VAX. This raw data must be processed to obtain desired and meaningful results. This processing occurs in two phases. First, the time interval between two debug log entries (taking into account the clock resolution and the overhead of making the calls to the debug log procedure) must be determined. This process is described in Section 5.3.1. Second, the sorting of these time values (for example, by message size) and the performing of statistical functions (such as determining the average time, maximum time, minimum time, standard deviation, and counting the number of samples) is accomplished. Section 5.3.2 describes the statistical analysis process. The source code for the analysis program is given in Appendix C.

5.3.1 Determination of Time Interval

In determining the time interval between two debug log entries, the analysis program uses the label field of the debug log entries to identify the data associated with each software probe. The user supplies the analysis program with the labels for each pair of appropriate software probes. For instance, in the queue message example, the pertinent labels are 1111 hex and 1112 hex. The analysis program searches through all the debug log entries stored in the VAX file, and saves entries that have the given labels. These saved entries are then paired, and their timestamp values are subtracted. This value gives

the number of clock ticks that occurred between the activation of the pair of software probes. To convert this number to a time value, it is multiplied by the clock resolution, which is 1.28 μsec per clock tick. One final bit of processing is needed before determining the length of the time interval. The overhead of activating the software probe (the time it takes to make the `debug_log` procedure call) needs to be subtracted from the time interval value.

To determine the overhead for software probe activation, a number of `debug_log` procedure calls were sequentially executed. As shown in Table 5-2, there was a 22 μsec time delay between the activation of two software probes. This implies that the overhead that should be subtracted between a pair of consecutive probes is 22 μsec . This is the overhead value that would be subtracted for the queue message example because the probes are in consecutive debug log entries. However, sometimes other debug entries are located in between the two entries that are of interest. For example, suppose we want to measure the length of time it takes for a task to execute, and within that task is a queue message call that we also want to measure. The task measurement probes would not be consecutive entries in the debug log because the queue message measurement probes would be located between them. Since there are two nested probes between the task probes, the overhead associated with the queue message probes also needs to be subtracted from the time for the task. Therefore, the number of intervening probes must be counted, so the overhead for all these probes can be taken into account. Hence, 22 μsec should be subtracted as additional overhead for each intervening software probe activation.

# of <code>debug_log</code> calls	Avg Time (μsec)	Stand Dev (μsec)	Max Time (μsec)	Min Time (μsec)	# Samples
2	22	3	25	17	177
3	43	2	51	43	177
4	66	3	69	61	177
5	87	2	94	87	177
6	110	3	112	106	177
7	131	2	138	130	177

Table 5-2. Overheads Associated with `debug_log` Procedure Calls

5.3.2 Statistical Analysis of Time Data

Once the overhead has been accounted for, the time interval between two debug log entries is known. These values are saved in an array, and it is easy to determine the average time, standard deviation, maximum time, minimum time, and number of samples in the array. These results are displayed on the monitor and stored in a file.

The analysis program can also sort the data according to the contents of the parameter field of the debug log entries. For the queue message example, the execution times are sorted according to message size. For each message size, the average time, standard deviation, etc. are given as well as overall statistics. As before, the results are displayed on the monitor and stored in a data file for later analysis.

Chapter 6

Performance Measurement Results

Using the methodology described in Chapter 5, empirical performance data for FTPP operating system overheads were collected. This chapter summarizes the measurements. Performance data for each of the operating system tasks are presented in the order in which they occur during each minor frame (Figure 4-1). Section 6.1 describes the overhead associated with the interrupt handler. Data for Rate Group Dispatcher performance are given in Section 6.2 for Part One (RGD₁) and in Section 6.4 for Part Two (RGD₂). Section 6.3 discusses the IO Dispatcher overhead. Performance data for the Fault Detection, Identification and Recovery (FDIR), IO Source Congruency Manager (IOSC), and the IO Processing (IOP) tasks are summarized in Section 6.5, Section 6.6, and Section 6.7, respectively. Finally, Section 6.8 presents results of other OS overheads, including queueing and retrieving message packets.

System Configuration

Before giving performance measurement results, the FTPP configuration preceding the data collection is described. All performance measurements were taken on a prototype FTPP Ada operating system running on a 20 MHz 68030-based Motorola MVME147S-1 Processing Element. Caches and compiler optimizations were turned on. The system used a prototype, hardware-implemented Network Element

Since many aspects of system performance are dependent upon the distribution of tasks, the task list used for all these measurements, unless stated otherwise, is given below. The user application task simply sent messages of varying length to itself, which it later read itself.

RG4 tasks (six)
fdir (local)
system_fdir
io_source_congruency_mgr
io_processing_task_rg4
io_application_task (user task)
application_task (user task)

RG3 tasks (one)
io_processing_task_rg3

RG2 tasks (one)
io_processing_task_rg2

RG1 tasks (one)
io_processing_task_rg1

6.1 Interrupt Handler Overhead

The interrupt handler (IH) updates the clock time, sets the next interrupt time, and scoops all queued messages. The IH code is in assembly, except for the `scoop` procedure which is in Ada. The time to scoop messages dominates the IH overhead, and no measurements have been taken of the assembly code, whose execution time is negligible. The performance data for the `scoop` procedure is given in the following section.

6.1.1 Scoop Message

The `scoop` procedure transfers message packets from a PE's receive queue in the Network Element's dual-port RAM to the PE's local memory space where they are reassembled into complete messages. This is described in Section 3.2.2.4. The time to scoop messages is dependent on the number of packets to be scooped, as shown in Table 6-1.

num pkts (64 bytes)	avg time (μ sec)	std dev (μ sec)	max time (μ sec)	min time (μ sec)	# samples
2	321	3	328	320	18
3	433	1	434	433	18
4	542	3	547	539	18
5	652	1	653	652	17
6	763	3	766	759	17
7	871	3	877	864	17
8	981	3	984	977	17
9	1091	2	1097	1089	17
10	1200	3	1203	1196	17
11	1310	3	1315	1308	17
12	1420	2	1422	1414	17

Table 6-1. Scoop Message Execution Time as a Function of Number of Packets

6.2 Rate Group Dispatcher (Part One) Overhead

The primary functions of the first part of the Rate Group Dispatcher (RGD₁) are to check for task overruns and to schedule the IO Dispatcher for execution, as described in Section 3.2.1.2. Overall, the execution time for RGD₁ varies as a function of the minor frame number, as shown in Table 6-2. The reason for this variance is that different minor frames have a different number of Rate Groups that have reached their RG boundaries (Section 3.2.1.1). When a Rate Group reaches its boundary, all tasks within that Rate Group should have completed their iterative cycle. RGD₁ ensures that all tasks that should have completed actually did, and the number of tasks to check depends on the number of Rate Groups that have reached RG boundaries.

minor frame	RG boundaries	avg time (μsec)	std dev (μsec)	max time (μsec)	min time (μsec)	# samples
0	4, 3, 2, 1	168	0	168	168	20
1	4	130	2	137	130	20
2	4, 3	143	0	143	143	20
3	4	130	0	130	130	20
4	4, 3, 2	156	0	156	156	20
5	4	130	0	130	130	20
6	4, 3	143	2	150	143	19
7	4	130	0	130	130	19

Table 6-2. Overall Rate Group Dispatcher (Part One) Execution Time as a Function of Minor Frame Number

Notice that RGD₁ executes longest during minor frame 0. This is because all Rate Group tasks have completed their iterative cycle at the completion of minor frame 7. Therefore, RGD₁ has to check for overruns of tasks in every Rate Group. RGD₁ has a minimal execution time during minor frames 1, 3, 5, and 7 because during those frames it only needs to check RG4 tasks for overruns.

The overall execution time for RGD₁ can be broken down into three main segments. Section 6.2.1 presents performance data for the time needed to record the congruent time value and to check for RGD (part two) overrun. Section 6.2.2 summarizes data on the time to check for Rate Group task overruns, and Section 6.2.3 presents data on the time needed to set up the next Rate Group interval and schedule the IO Dispatcher.

6.2.1 Record Congruent Time Value, Check for RGD₂ Overrun

At the beginning of execution, RGD₁ records the congruent time value and then verifies that the second part of the Rate Group Dispatcher (RGD₂) did not overrun during the previous minor frame. The time to accomplish these duties, as seen in Table 6-3, is constant and thus does not vary with the frame number or task distribution.

avg time (μsec)	std dev (μsec)	max time (μsec)	min time (μsec)	# samples
21	0	21	21	158

Table 6-3. RGD₁ Update Congruent Time Value and Check for RGD₂ Overrun
Execution Time

6.2.2 Check for RG Task Overruns

RGD₁ determines whether any of the tasks that were to complete their iterative cycle and suspend themselves during the previous minor frame overran the frame boundary. The time needed to accomplish this is a function of the number of RG tasks that were scheduled to suspend themselves during the previous minor frame. This is shown in Table 6-4. This segment of RGD₁ is the only one that does not have a constant execution time.

num RG tasks	minor frames	avg time (μsec)	std dev (μsec)	max time (μsec)	min time (μsec)	# samples
6	1, 3, 5, 7	58	0	58	58	79
7	2, 6	69	3	71	65	39
8	4	79	3	84	77	20
9	0	90	0	90	90	20

Table 6-4. RGD₁ Check for Rate Group Task Overruns Execution Time as a Function of
Number of Rate Group Tasks

6.2.3 Set Up Next RG Interval, Schedule IO Dispatcher

Before finishing execution, RGD₁ sets up the next Rate Group interval; this entails determining when the next interrupt should occur. RGD₁ then schedules the IO

Dispatcher to execute next. These duties are done every minor frame, and the amount of time needed to do them is constant for all minor frames. The execution times are summarized in Table 6-5.

avg time (μ sec)	std dev (μ sec)	max time (μ sec)	min time (μ sec)	# samples
48	3	53	47	158

Table 6-5. RGD₁ Set Up RG Interval and Schedule IO Dispatcher Execution Time

6.3 IO Dispatcher (IOD) Overhead

IO performance data collection is incomplete because the FTPP IO System Services are not completely implemented, and the sections that are implemented have not been optimized. IO is application-specific, and as a result it is very difficult to make general statements about IO performance. However, to provide an estimate of IO performance, some data were collected using restricted IO. In particular, all IO was outbound-only and used Ethernet to send out the data.

The IO Dispatcher (IOD) consists of three main sections. First, it determines which IO requests should be executed this frame and then starts their execution. Second, it waits for the IO requests to finish execution. Finally, after waiting, IOD reads any incoming IO data. Each of these activities is discussed in the following paragraphs.

IOD determines which IO requests should execute during the current minor frame by checking the IO execution table, and it then starts the execution of each of these requests. For outgoing IO requests using Ethernet, IOD must first transfer the data to an area of memory used for Ethernet transfers before starting the IO request. This transfer is done on a byte-by-byte basis. The time required to transfer the data varies with the number of bytes to be transferred. This transfer time was measured and is approximately 5 μ sec for each byte sent out. This implies that IOD would spend 500 μ sec transferring data for an IO request consisting of sending out 100 bytes of data.

After starting the execution of all IO requests, IOD waits while the execution takes place. The amount of time spent waiting depends on how long it takes to execute the IO request, which is dependent on the hardware device executing the IO. The wait period is a constant and should equal the longest amount of time needed to execute the IO requests for any minor frame. Since no data on IO execution time has been collected, the set IOD wait period is currently an arbitrarily large number.

IOD's last duty is to read all incoming IO data. No performance data was collected for this because all IO was strictly outgoing IO.

6.4 Rate Group Dispatcher (Part Two) Overhead

The primary functions of the second part of the Rate Group Dispatcher (RGD₂) are to send queued message packets and to schedule Rate Group tasks for execution (see Section 3.2.3.2). A summary of the overall RGD₂ execution times, sorted by minor frame number, is given in Table 6-8.

minor frame	RG boundaries	avg time (μsec)	std dev (μsec)	max time (μsec)	min time (μsec)	# samples
0	4, 3, 2, 1	1454	409	2134	710	20
1	4	1190	406	1817	549	20
2	4, 3	1279	402	1905	730	20
3	4	1173	394	1830	629	20
4	4, 3, 2	1367	388	2036	841	20
5	4	1165	374	1817	636	19
6	4, 3	1317	381	1929	736	19
7	4	1197	378	1824	629	19

Table 6-8. Overall Rate Group Dispatcher (Part Two) Execution Time as a Function of Minor Frame Number

As is evident from the large standard deviations in Table 6-8, RGD₂ execution times do not vary directly with the minor frame number. Unlike RGD₁, which only varied as a function of the number of tasks that suspended themselves during the previous minor frame, the dependencies of RGD₂ are more complicated. In particular, RGD₂ performance is related to the number of message packets that were enqueued during the previous minor frame. Since the number of enqueued packets can differ for a given application from one iteration to the next, it is not meaningful to examine RGD₂ execution times as a function of only the minor frame number.

It is more useful to break RGD₂ into several segments and then examine each segment separately. The following five sections describe the five major segments of RGD₂: update congruent time value and check for RGD₁ and IOD overruns (Section

6.4.1); send queued message packets (Section 6.4.2); update message packet queues (Section 6.4.3); schedule Rate Group tasks (Section 6.4.4); and increment minor frame number and set up IO interval for the next frame (Section 6.4.5).

6.4.1 Update Congruent Time Value, Check for RGD₁ and IOD Overrun

At the beginning of each iteration cycle, RGD₂ updates the congruent time value used by each Rate Group and checks to see if either the Rate Group Dispatcher (part one) task or the IO Dispatcher task exceeded its execution time bound. The time to accomplish these duties is the same during each iteration of RGD₂. A summary of the execution time data is given in Table 6-9.

avg time (μsec)	std dev (μsec)	max time (μsec)	min time (μsec)	# samples
40	2	46	40	157

Table 6-9. RGD₂ Update Congruent Time Values and Check for RGD₁ and IOD Overrun

6.4.2 Send Queue

RGD₂ calls the `send_queue` procedure once for each task that suspended itself during the previous minor frame. `send_queue` transfers enqueued message packets from each PE's local memory space to the Network Element where they are processed and delivered. The execution time of each `send_queue` call is a function of the number of packets that were queued by that task, as shown in Table 6-10. Therefore, the total amount of time RGD₂ spends sending queued packets depends on the number of tasks that suspended themselves during the previous minor frame and on the number of packets enqueued by each task.

It is important to note that the data in Table 6-10 was collected using only one Virtual Group. Since only one PE was connected to the NE, no contention for NE service occurred (Section 4.2). Therefore, these numbers represent best case performance; if there were contention, the `send_queue` execution times would increase.

pkts sent per task	avg time (μ sec)	std dev (μ sec)	max time (μ sec)	min time (μ sec)	# samples
0	5	3	10	2	1140
1	78	14	231	77	256
2	182	12	209	171	22
3	301	14	322	283	22
4	417	7	440	409	21
5	535	10	552	528	21
6	657	12	684	647	21
7	770	11	797	758	21
8	890	8	909	877	21
9	1007	9	1027	990	21
10	1125	9	1146	1115	21

Table 6-10. RGD₂ Send Queue (Per Task) Execution Time as a Function of Number of Packets

6.4.3 Update Queue

RGD₂ calls the `update_queue` procedure once for each task that suspended itself during the previous minor frame. This procedure updates pointers used in each PE's receive queue, located in the dual-port RAM. The execution time of each `update_queue` procedure call varies as a function of the number of receive_queue pointers which need to be updated and is equal to the number of packets enqueued during the previous frame. This is shown in Table 6-11. The total amount of time spent by RGD₂ updating queues is a function of the number of tasks that suspended themselves during the previous minor frame and the number of packets enqueued by each task.

pkts sent per task	avg time (μ sec)	std dev (μ sec)	max time (μ sec)	min time (μ sec)	# samples
0	16	3	24	11	879
1	28	3	31	24	195
2	36	1	37	36	17
3	42	1	43	42	16
4	51	3	56	48	16
5	56	2	61	55	16
6	66	3	69	61	16
7	74	0	74	74	15
8	81	3	87	80	16
9	86	1	87	86	16
10	94	3	100	93	16

Table 6-11. RGD₂ Update Queue (Per Task) Execution Time as a Function of Number of Packets

The data for `send_queue` and `update_queue` are linear, as shown by the graphical representation of the performance data, which is given in Figure 6-1.

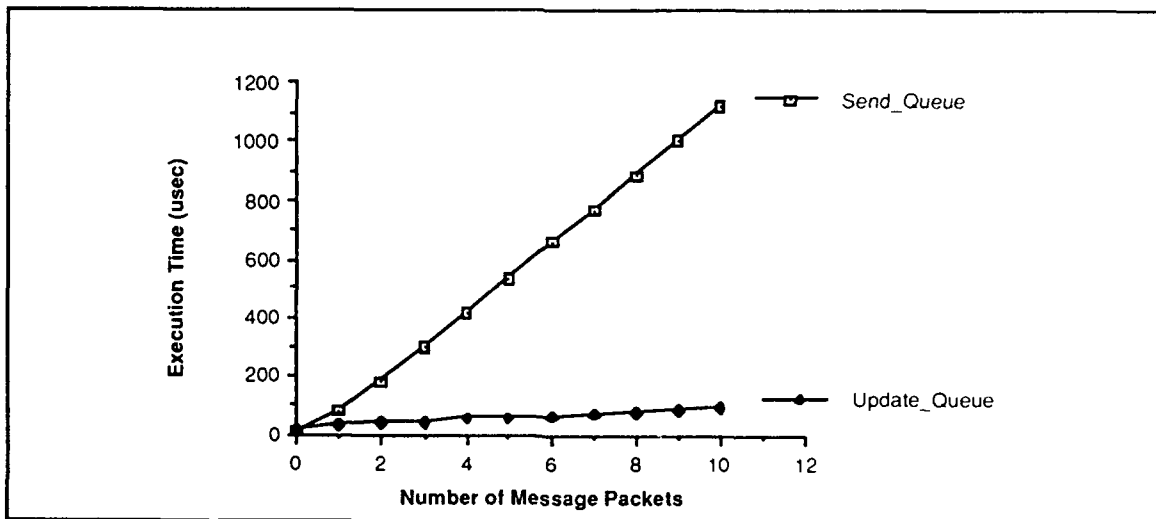


Figure 6-1. Graphical Representation of Send Queue and Update Queue Execution Time as a Function of Number of Packets

6.4.4 Schedule Rate Group Tasks

RGD₂ schedules Rate Group tasks to run in the time remaining in the current frame. It does this by calling a scheduling procedure for each Rate Group that reached its RG boundary during the previous minor frame. Therefore, this scheduling procedure is called a maximum of four times by RGD₂ (in minor frame 0). It is always called at least once during a minor frame. The execution time of the scheduler is a function of the number of tasks that need to be scheduled for a particular Rate Group. Table 6-12 summarizes the scheduler performance data. To collect more data points for the time needed to schedule RG tasks, the system configuration described at the beginning of this chapter was altered by adding more application tasks.

num tasks per Rate Group	avg time (μsec)	std dev (μsec)	max time (μsec)	min time (μsec)	# samples
1	55	3	59	52	74
2	85	2	90	84	72
3	121	2	127	121	70
4	143	3	146	140	65
5	134	2	140	134	129
6	160	3	166	158	147
7	190	2	196	190	140
8	221	1	222	216	131

Table 6-12. RGD₂ Schedule Rate Group Tasks Execution Time as a Function of Number of Tasks Per Rate Group

6.4.5 Increment Frame Number, Set Up IO Interval for Next Frame

At the end of each RGD₂ execution cycle, the minor frame number is incremented and the IO interval is set up for the next minor frame. These activities take place just one time per RGD₂ execution. As seen in Table 6-13, the execution time to perform these duties is constant and is negligible compared to the total RGD₂ execution time.

avg time (μsec)	std dev (μsec)	max time (μsec)	min time (μsec)	# samples
9	3	16	8	157

Table 6-13. RGD₂ Increment Frame Number and Set IO Interval Execution Time

6.4.6 RGD₂ Summary

The overall RGD₂ execution time has five constituent parts. Two of these are constant, and they account for 49 μsec of RGD₂ overhead. Of the other three constituents, two (`send_queue` and `update_queue`) have execution times which are a function of the number of enqueued message packets. The final constituent of RGD₂ overhead is the time needed to schedule Rate Group tasks; this is a function of the number of tasks to schedule.

6.5 Fault Detection, Identification, and Recovery (FDIR) Overhead

The FDIR overhead for all Virtual Groups (VGs) within FTPP is the time to execute the Local FDIR task, except for the System VG, which executes the System FDIR task in addition to Local FDIR. Performance data for the System FDIR task are not presented because the task has not yet been fully implemented. Data for the execution times of the Local FDIR task are given in Table 6-14. Local FDIR simply enqueues a one-packet message which is delivered to the System FDIR task. Its execution time is constant, even with faults present in the system.

avg time (μsec)	std dev (μsec)	max time (μsec)	min time (μsec)	# samples
84	2	90	84	210

Table 6-14. Local FDIR Execution Time

6.6 IO Source Congruency Manager (IOSC) Overhead

The IO Source Congruency Manager ensures that all members of a redundant VG receive a copy of any input read by another member. The system configuration used to collect performance data used a simplex VG for IO, so the IOSC execution time reported in Table 6-15 should be regarded as a "best case" execution time.

avg time (μ sec)	std dev (μ sec)	max time (μ sec)	min time (μ sec)	# samples
52	1	59	52	142

Table 6-15. Minimal IO Source Congruency Manager Execution Time

6.7 IO Processing Task (IOP) Overhead

The IO Processing task is responsible for ensuring that all members of a redundant VG end up with a single input value. This involves some data smoothing or averaging. The performance measurements summarized in Table 6-15 indicate a relatively large standard deviation. This might be because there are four instantiations of this task, one for each Rate Group. The IOP code is not fully implemented, and the implementation will be strongly dependent on the application.

avg time (μ sec)	std dev (μ sec)	max time (μ sec)	min time (μ sec)	# samples
15	12	34	2	357

Table 6-15. Minimal IO Processing Task Execution Time

6.8 Other Overheads

There are several system overheads that are not explicitly shown in the minor frame overview given in Figure 4-1. These include the `queue_message` overhead, the `retrieve_message` overhead, and the time needed to context switch between tasks. Performance data for these three overheads are given in the following sections.

6.8.1 Queue Message

The `queue_message` procedure call is used by a task when sending a message. This procedure decomposes the message into packets and then enqueues these packets in the PE's local memory space for later transfer to the NE. As indicated in Table 6-16, the amount of time needed to enqueue a message is a function of the length of the message.

msg size (bytes)	msg size (packets)	avg time (μ sec)	std dev (μ sec)	max time (μ sec)	min time (μ sec)	# samples
0	1	84	2	90	84	20
100	2	136	3	140	134	19
200	4	221	1	222	221	19
300	5	272	3	278	271	19
400	7	358	1	359	358	18
500	8	410	2	415	409	18
600	10	497	2	502	496	19
700	12	546	1	552	546	19
800	13	634	2	641	633	19
900	15	718	3	722	715	19
1000	16	771	2	778	771	19

Table 6-16. Queue Message Execution Time as a Function of Message Size

6.8.2 Retrieve Message

The `retrieve_message` procedure is used by tasks to reassemble delivered packets into complete messages. As with `queue_message`, the time to retrieve a message is dependent upon the size of the message. This is shown in Table 6-17. Notice that it takes longer to retrieve a message of a given length than to enqueue it. When packets are delivered by the NE, syndrome information indicating whether any redundant copies of the packet differed from the majority vote is attached to each packet. While retrieving a message, some of this syndrome information is processed, and that accounts for the increased execution time.

msg size (bytes)	msg size (packets)	avg time (μ sec)	std dev (μ sec)	max time (μ sec)	min time (μ sec)	# samples
0	1	121	2	127	121	19
100	2	193	3	196	190	19
200	4	312	3	315	308	19
300	5	379	3	385	377	19
400	7	499	3	504	496	19
500	8	567	3	571	565	19
600	10	685	3	690	683	19
700	12	752	2	753	746	19
800	13	871	2	877	871	19
900	15	991	3	996	990	18
1000	16	1058	1	1059	1058	18

Table 6-17. Retrieve Message Execution Time as a Function of Message Size

Figure 6-2 depicts a graphical representation of the data contained in Table 6-16 (Queue Message) and Table 6-17 (Retrieve Message).

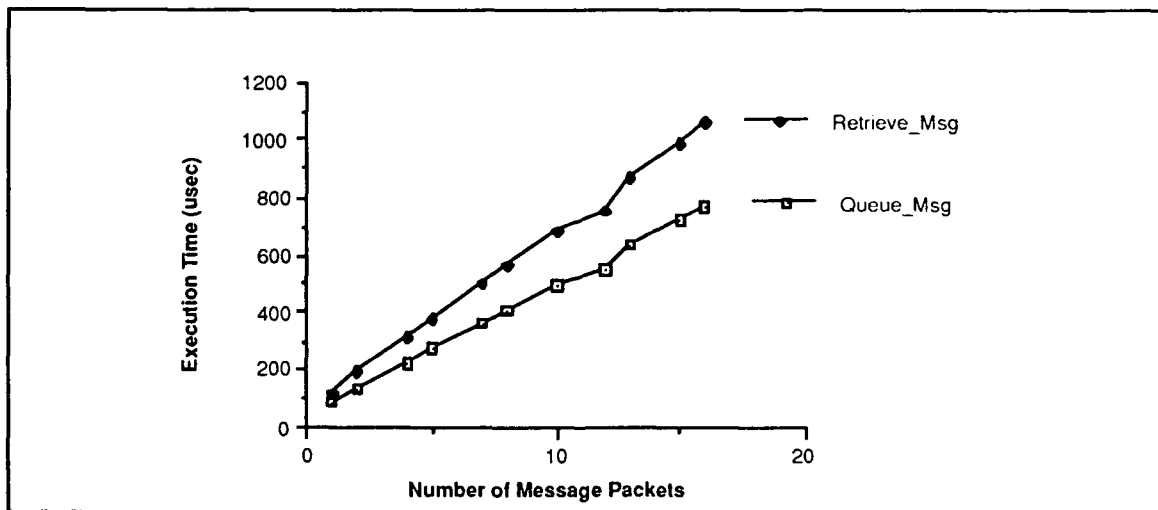


Figure 6-2. Graphical Representation of Queue Message and Retrieve Message Execution Time as a Function of Number of Packets

6.8.3 Context Switch Overhead

The amount of time needed to context switch between two tasks was measured, and the results are summarized in Table 6-18. These measurements were collected by creating a system configuration where two tasks in the same Rate Group were given consecutive priorities. This ensured that one task would execute immediately prior to the second one. Software probes were placed just before the iterative completion point of the first tasks and just after the iterative completion point of the second task. The context switch time was determined by subtracting the two timestamp values,.

avg time (μ sec)	std dev (μ sec)	max time (μ sec)	min time (μ sec)	# samples
19	2	24	18	26

Table 6-18. Context Switch Execution Time

6.9 Performance Data Summary

The overheads in this chapter were presented according to their occurrence during a minor frame. However, the system overheads can be grouped according to their purpose. Using this scheme, four major categories exist: communication overheads, scheduling overheads, IO overheads, and fault detection overheads. The tasks/procedures associated with each of these four groups are listed below.

Communication Overheads

IH (scoop message)
 RGD₂ (send_queue and update_queue)
 Queue Message (called by application task)
 Retrieve Message (called by application task)

Scheduling Overheads

RGD₁
 RGD₂ (excluding send_queue and update_queue)
 Context Switching

Input/Output

IOD
 IOSC
 IOP

Fault Detection

FDIR

For the system configuration used in this chapter, the communication overheads dominate the total overhead. On average, the application task sends five packets per minor frame; therefore, an average of six packets are processed per minor frame (including the one-packet FDIR message). The OS communication overheads per minor frame include `scoop` (763 μ sec), `send_queue` (637 μ sec), and `update_queue` (162 μ sec). The total communication overhead is 1562 μ sec. The total OS overhead, excluding IO, is 2199 μ sec (average $RGD_1 = 141$ μ sec, average $RGD_2 = 1268$ μ sec, local FDIR = 84 μ sec). Therefore, the three communication procedures account for 71.0% of the total overhead. Note that the `queue_message` and `retrieve_message` overheads aren't counted in the communication overhead. This is because they are system procedures which are called by the application tasks. Therefore, the overhead for queueing and retrieving messages is included in the task's execution time.

The overheads associated with scheduling and fault detection are rather low compared with those associated with communication. Scheduling activities take, on average, 553 μ sec per minor frame, which is 25.1% of the total OS overhead. The local FDIR task takes just 84 μ sec per minor frame, or 3.8% of the total. Table 6-19 summarizes the percentage of overhead (excluding IO) due to communication, scheduling and fault detection, for an average minor frame. Note that the data in Table 6-19 represent values averaged over eight minor frames; the overhead can vary widely from minor frame to minor frame.

Overhead Category	Average Overhead (μ sec)	% of Total Overhead (excluding IO)
Communication	1562	71.0 %
Scheduling	553	25.1 %
Fault Detection	84	3.8 %

Table 6-19. OS Overhead Due to Communication, Scheduling and Fault Detection
(Average Values for a Minor Frame)

The significance of the IO overhead is highly dependent on the amount and type of IO performed. The important contribution of this thesis concerning IO performance measurement is the development of a methodology which can be used to continuously evaluate IO performance as development progresses.

The overall FTTP OS overhead (excluding IO) is 2199 μ sec per minor frame, on average. Thus, 22% of the 10 msec minor frame is consumed by operating system overhead. This compares favorably to the Software Implemented Fault Tolerance (SIFT) computer which requires 64.3% OS overhead [Pal85]. The primary source of SIFT overhead is due to voting and data consistency functions. In FTTP, the voting and data consistency functions are considered part of the communication overhead. Therefore, as with SIFT, voting and data consistency functions can also be considered a primary source of overhead for FTTP. However, FTTP uses the hardware-based Network Element to reduce the total overhead.

Chapter 7

Detailed OS Overhead Model

One important use of the performance data presented in the previous chapter is its incorporation into a model which can estimate the operating system (OS) overhead under various configurations and workloads. Using the empirical performance data summarized in Chapter 6, Section 7.1 presents a detailed description of the OS overhead model. Section 7.2 illustrates the use of the model with a given system configuration and workload; predicted overheads are compared with measured overheads.

7.1 OS Overhead Model with Empirical Data

This section gives a detailed description of the FPHP operating system overhead model, based on the empirical performance data presented in Chapter 6. A general description of this model was given in Section 4.1. As in Section 4.1, the overhead model will be described according to the occurrence of each OS task in the minor frame (Figure 4-1).

The amount of overhead per minor frame is the sum of the execution times for each of the following operating system tasks: interrupt handler (IH), Rate Group Dispatcher (RGD), IO Dispatcher (IOD), Fault Detection Identification and Recovery (FDIR), IO Source Congruency Manager (IOSC), and IO processing (IOP). This overhead is represented by the following equation:

$$OH = IH_1 + RGD_1 + IOD + IH_2 + RGD_2 + FDIR + IOSC + IOP$$

A detailed description of each of these eight overheads follows.

7.1.1 Interrupt Handler (IH₁) Overhead

The overhead associated with the first interrupt handler (IH₁) is given by the following general equation:

$$IH_1 = (\text{time to update clock}) + (\text{time to schedule next interrupt}) + \\ (\text{time to scoop messages})$$

Updating the clock and scheduling the next interrupt are executed in assembly language routines and therefore could not be directly measured using the Ada-based software probes described in Section 5.1. However, the IH overhead is overwhelmingly dominated by the time needed to scoop messages, so the time needed to update the clock and schedule the next interrupt is negligible and will be ignored.

The time to scoop message packets is a function of the number of packets that arrived in the processor's receive queue since the last time a scoop was executed. The data in Table 6-1 indicate that the relationship between the scoop time and the number of packets is linear. As a result, the overhead associated with the interrupt handler can be given as below:

$$IH_1 = 110 * \text{number_of_packets} + 103 \text{ (}\mu\text{sec)}$$

7.1.2 Rate Group Dispatcher - Part One (RGD₁) Overhead

The amount of time needed to execute the first part of the Rate Group Dispatcher (RGD₁) can be summarized with the following general equation:

$$RGD_1 = (\text{time to update congruent time}) + (\text{time to check for RGD}_2 \text{ overrun}) + \\ (\text{time to check for task overruns}) + (\text{time to set up next RG interval}) + \\ (\text{time to schedule IOD})$$

With the exception of checking for task overruns, all the components of the Rate Group Dispatcher (part one) are constant. Table 6.3 and Table 6.5 quantify this total constant overhead as 69 μ sec. The time needed to check for task overruns varies with the number of tasks that completed their iterative cycle during the previous minor frame. Table 6.4 shows that this overhead is approximately 10 μ sec per task. Therefore, the total overhead associated with RGD₁ can be described by the following:

$$RGD_1 = 10 * \text{number_of_suspended_tasks} + 69 \text{ (}\mu\text{sec)}$$

7.1.3 IO Dispatcher (IOD) Overhead

The general overhead associated with the IO Dispatcher task is given below:

$$IOD = (\text{time to increment frame counter}) + (\text{time to start IOR execution}) + \\ (\text{time to wait for IO to complete}) + (\text{time to read input data})$$

As explained in Section 6.3, IO performance measures were limited to outgoing IO data. This makes it very difficult to explore the constituent IOD overheads in much detail. The time to increment the frame counter is constant and is negligible (one 'add' statement in Ada). The other constant is the time to wait for IO to complete. This is simply a busy-wait of a duration chosen by the application programmer to ensure that any outward-bound IO is finished before any attempt is made to read incoming IO data. Though the wait is constant for a given system configuration, it can vary widely depending on the application and type of IO performed for the given configuration.

The two remaining constituents of the IOD overhead are variable and depend on the type and amount of IO activity to be performed during a given minor frame. The time to start the execution of IO Requests depends on the number of IO requests scheduled to run this minor frame that have outgoing data, and it also depends on how much data each IO request is sending out. As stated in Section 6.3, the time needed to start IOR execution is approximately 5 μ sec per outgoing byte of IO data. Finally, the time to read input data obviously depends on the number of IO requests that have incoming data and on the amount of data coming in. No performance measurements were taken using incoming IO data.

7.1.4 Interrupt Handler (IH₂) Overhead

The overhead equation associated with the second interrupt handler (IH₂) is the same as that given for IH₁ (Section 7.1.1) and is repeated below:

$$IH_2 = 110 * \text{number_of_packets} + 103 \text{ } (\mu\text{sec})$$

Even though both instances of the interrupt handler are modeled by the same equation, in general the overheads associated with IH₁ and IH₂ will be different. This is because the time to scoop messages will vary with the number of packets present in the receive queue for the processor. Typically, the time interval between the occurrence of

IH₁ and IH₂ is less than the time duration from IH₂ to the next occurrence of IH₁. This implies that more packets have had an opportunity to arrive in the receive queue during the interval from IH₂ to IH₁, and therefore the time to scoop messages should generally be longer for IH₁ than IH₂.

7.1.5 Rate Group Dispatcher - Part Two (RGD₂) Overhead

The execution time for the second part of the Rate Group Dispatcher (RGD₂) can be generally described as follows:

$$\begin{aligned} RGD_2 = & (time\ to\ update\ congruent\ time) + (time\ to\ check\ for\ RGD_1\ overrun) + \\ & (time\ to\ check\ for\ IOD\ overrun) + (time\ to\ send\ queued\ messages) + \\ & (time\ to\ update\ queues) + (time\ to\ schedule\ RG\ tasks) + \\ & (time\ to\ increment\ frame\ count) + (time\ to\ set\ up\ IO\ interval) \end{aligned}$$

All but three of the RGD₂ constituents listed above have constant execution times. The time to update the congruent time value, check for RGD₁ and IOD overrun, increment frame count, and set up IO interval is constant and equals 49 μ sec. The three variable constituents of RGD₂ are the time to send queued messages, the time to update queues, and the time to schedule RG tasks.

The time to send queued messages is a function of the number of tasks that suspended themselves during the previous minor frame and the number of message packets that each task had enqueued since the last time its queue was sent. For each task, the time to send the queued packets (Table 6-10) is given by:

$$Send_Queue\ (per\ task) = 115 * number_of_packets - 31\ (\mu sec)$$

The time to update a task's queue is a function of the number of packets received and the number of packets read since the last time the queue was updated. Table 6-11 yields the following equation:

$$Update_Queue\ (per\ task) = 8 * number_of_packets + 19\ (\mu sec)$$

Since the time to send queued messages and update the message queues both vary with the number of packets enqueued, they can be combined into the following single equation:

$$\text{Send_and_Update_Queue (per task)} = 123 * \text{number_of_packets} - 12 (\mu\text{sec}) \quad (1)$$

The time to schedule the Rate Group (RG) tasks is a function of the number of RG tasks that are to be scheduled this minor frame. The data in Table 6-12 results in the following equation:

$$\text{Schedule_Tasks (per Rate Group)} = 26 * \text{number_of_rg_tasks} + 15 (\mu\text{sec}) \quad (2)$$

The three variable constituents of RGD_2 can be represented by two equations. Including the constant constituent, the general expression of the RGD_2 overhead can now be expressed as:

$$\text{RGD}_2 = (\text{time to send and update queues}) + (\text{time to schedule tasks}) + (\text{a constant})$$

Using equations (1) and (2), the detailed equation for the total RGD_2 overhead is given by:

$$\text{RGD}_2 = \sum_{i=1}^{\text{num_tsk}} [(123 * \text{num_pkt}_i) - 12] + \sum_{j=1}^{\text{num_rg_tsk}} [(26 * \text{num_rg_tsk}_j) + 15] + 49 (\mu\text{sec})$$

where,

- num_tsk is the number of tasks with messages to send that completed their iterative cycle during the previous minor frame.
- num_pkt is the number of packets a task has enqueued since its last send_queue call.
- num_rg is the number of Rate Groups that begin a new frame boundary in the current minor frame.
- num_rg_tsk is the number of tasks in a given Rate Group.

It is interesting to note that the RGD_2 overhead is much more sensitive to the number of packets to send than to the number of tasks to schedule. There is approximately five times as much additional RGD_2 overhead for each additional message packet than that for each additional task.

7.1.6 Fault Detection Identification and Recovery (FDIR) Overhead

The overhead of running the Local FDIR task is the same as that for enqueueing a one-packet message, which is all the Local FDIR task does.

$$FDIR = (\text{time to enqueue message to System FDIR task})$$

The Local FDIR task has a constant execution time, as shown in Table 6-14. Therefore, the overhead for FDIR can be expressed as:

$FDIR = 84 \text{ } (\mu\text{sec})$

7.1.7 IO Source Congruency Manager (IOSC) Overhead

The IO Source Congruency Manager (IOSC) ensures all members of a redundant Virtual Group receive a copy of any input read by another member. The overhead associated with the IOSC task is given below:

$$IOSC = (\text{time to exchange input data among VG members})$$

The data for IOSC were collected using a simplex VG for IO. Therefore, the data represents a best case value since the IO data did not need to be exchanged among members of a redundant VG. The minimal overhead for IOSC is given as:

$IOSC = 52 \text{ } (\mu\text{sec})$

7.1.8 IO Processing Task (IOP) Overhead

The IO Processing (IOP) task is responsible for ensuring that all members of a VG performing redundant IO end up with a single input value. This usually involves some data smoothing or averaging. For instance, the average of three sensor values could be used as the single input value. This processing or smoothing of the input data is specific to the application, and can vary widely as far as execution time is concerned. The general IOP overhead is given below:

$$IOP = (\text{time to process input data})$$

The IOP task is not fully implemented, and the implementation will be strongly dependent on the application. Therefore, the data for IOP execution time given in Table 6-15 represent minimum execution times for IOP. Using these data, the minimal IOP overhead is:

$$IOP = 15 \text{ } (\mu\text{sec})$$

7.1.9 Total OS Overhead

The total OS overhead for a given minor frame, excluding IO, is given by:

$$OH = IH_1 + RGD_1 + IH_2 + RGD_2 + FDIR$$

where,

$$IH_1 = 110 * \text{number_of_packets} + 103$$

$$RGD_1 = 10 * \text{number_of_suspended_tasks} + 69$$

$$IH_2 = 110 * \text{number_of_packets} + 103$$

$$RGD_2 = \sum_{i=1}^{\text{num_tsk}} [(123 * \text{num_pkt}_i) - 12] + \sum_{j=1}^{\text{num_rg_tsk}} [(26 * \text{num_rg_tsk}_j) + 15] + 49$$

$$FDIR = 84$$

By combining both IH overheads into one and merging all constants, the overall OS overhead (excluding IO overhead) for a given minor frame becomes:

$$OH = (110 * \text{num_pkt_scooped}) + (10 * \text{num_tsk}) + \sum_{i=1}^{\text{num_tsk}} [(123 * \text{num_pkt}_i) - 12] + \sum_{j=1}^{\text{num_rg_tsk}} [(26 * \text{num_rg_tsk}_j) + 15] + 305 \text{ } (\mu\text{sec})$$

where,

<i>num_pkt_scooped</i>	is the total number of packets scooped during the minor frame.
<i>num_tsk</i>	is the number of tasks with messages to send that completed their iterative cycle during the previous minor frame.
<i>num_pkt</i>	is the number of packets a task has enqueued since its last <i>send_queue</i> call.
<i>num_rg</i>	is the number of Rate Groups that begin a new frame boundary in the current minor frame.
<i>num_rg_tsk</i>	is the number of tasks in a given Rate Group.

7.2 Example of Overhead Model Use

To illustrate the use of the detailed OS overhead model presented in Section 7.1, an example system configuration is created (Section 7.2.1), the system parameters are used as input to the overhead model in order to predict the OS overheads (Section 7.2.2), and the predicted overheads are compared with empirically measured overheads (Section 7.2.3). Section 7.2.4 illustrates several other ways to use the OS overhead model.

7.2.1 Description of Example System Configuration

For our example, the FTPP is configured with three user application tasks. The first one is an RG4 task that sends and retrieves a 3-packet message during each iteration. The second application task is an RG3 task that sends and retrieves a 6-packet message during each iteration. The third application task is an RG1 task that sends and retrieves a 2-packet message during each iteration. A listing of all schedulable tasks, sorted by Rate Group, is given below:

RG4 tasks (six)

fdir (local)
system_fdir
io_source_congruency_mgr
io_processing_task_rg4
io_application_task (user task)
application_task_1 (user task)

RG3 tasks (two)

io_processing_task_rg3
application_task_2 (user task)

RG2 tasks (one)

io_processing_task_rg2

RG1 tasks (two)

io_processing_task_rg1
application_task_3 (user task)

7.2.2 Predicted Overheads

The OS overheads vary as function of several parameters, as described in Section 7.1. These parameters include the total number of message packets scooped, the number of tasks that completed their iterative cycle during the previous minor frame, the number of packets sent by each task during the previous frame, the number of Rate Groups that reached a frame boundary during the previous minor frame, and the number of schedulable tasks for each of the Rate Groups which are at a frame boundary. Based on the system configuration given above, the values for each of these parameters during each minor frame are given in Table 7-1.

minor frame number	num pkts scooped	num task compl	num pkts per task	num RG at frame boundary	num tasks per RG
0	12	11	3 (appl_1), 6 (appl_2), 2 (appl_3), 1 (fdir)	RG4, RG3, RG2, RG1	6 (RG4), 2 (RG3), 1 (RG2), 2 (RG1)
1	4	6	3 (appl_1), 1 (fdir)	RG4	6 (RG4)
2	10	8	3 (appl_1), 6 (appl_2), 1 (fdir)	RG4, RG3	6 (RG4), 2 (RG3)
3	4	6	3 (appl_1), 1 (fdir)	RG4	6 (RG4)
4	10	9	3 (appl_1), 6 (appl_2), 1 (fdir)	RG4, RG3, RG2	6 (RG4), 2 (RG3), 1 (RG2)
5	4	6	3 (appl_1), 1 (fdir)	RG4	6 (RG4)
6	10	8	3 (appl_1), 6 (appl_2), 1 (fdir)	RG4, RG3	6 (RG4), 2 (RG3)
7	4	6	3 (appl_1), 1 (fdir)	RG4	6 (RG4)

Table 7-1. System Parameters for Each Minor Frame

Using the parameter values given in Table 7-1 and the OS overhead equations given in Section 7.1, the OS overhead for each minor frame can be predicted. These predictions are presented below:

Frame 0

$$IH_1 = (110 * 12) + 103 = 1423 \mu\text{sec}$$

$$RGD_1 = (10 * 11) + 69 = 179 \mu\text{sec}$$

$$\begin{aligned} RGD_2 &= [(123 * 3) - 12] + [(123 * 6) - 12] + [(123 * 2) - 12] + [(123 * 1) - 12] + \\ &\quad [(26 * 6) + 15] + [(26 * 2) + 15] + [(26 * 1) + 15] + [(26 * 2) + 15] + 49 \\ &= 1823 \mu\text{sec} \end{aligned}$$

$$FDIR = 84 \mu\text{sec}$$

$$TOTAL = 3509 \mu\text{sec} \text{ (35.1\% of minor frame)}$$

Frames 1, 3, 5, and 7

$$IH_1 = (110 * 4) + 103 = 543 \mu\text{sec}$$

$$RGD_1 = (10 * 6) + 69 = 129 \mu\text{sec}$$

$$RGD_2 = [(123 * 3) - 12] + [(123 * 1) - 12] + [(26 * 6) + 15] + 49 = 688 \mu\text{sec}$$

$$FDIR = 84 \mu\text{sec}$$

$$TOTAL = 1444 \mu\text{sec} \text{ (14.4\% of each minor frame)}$$

Frames 2 and 6

$$IH_1 = (110 * 10) + 103 = 1203 \mu\text{sec}$$

$$RGD_1 = (10 * 8) + 69 = 149 \mu\text{sec}$$

$$\begin{aligned} RGD_2 &= [(123 * 3) - 12] + [(123 * 6) - 12] + [(123 * 1) - 12] + \\ &\quad [(26 * 6) + 15] + [(26 * 2) + 15] + 49 = 1481 \mu\text{sec} \end{aligned}$$

$$FDIR = 84 \mu\text{sec}$$

$$TOTAL = 2917 \mu\text{sec} \text{ (29.2\% of each minor frame)}$$

Frame 4

$$IH_1 = (110 * 10) + 103 = 1203 \mu\text{sec}$$

$$RGD_1 = (10 * 9) + 69 = 159 \mu\text{sec}$$

$$\begin{aligned} RGD_2 &= [(123 * 3) - 12] + [(123 * 6) - 12] + [(123 * 1) - 12] + \\ &\quad [(26 * 6) + 15] + [(26 * 2) + 15] + [(26 * 1) + 15] + 49 = 1522 \mu\text{sec} \end{aligned}$$

$$FDIR = 84 \mu\text{sec}$$

$$TOTAL = 2995 \mu\text{sec} \text{ (30.0\% of minor frame)}$$

Note that IO overheads are not considered in this example. Also, all the overhead for scooping messages is assumed to occur in IH_1 . This is because the system was configured using only a single VG. Since all messages are sent from and received by the same VG, all message packets will be scooped at the beginning of each minor frame, during IH_1 .

7.2.3 Comparison of Predicted and Actual Overheads

To determine the accuracy of the OS overhead model, empirical performance data were collected using the system configuration described in Section 7.2.1. A comparison of the overheads predicted by the model and the observed overheads is given in Table 7-2. Note that the overheads are the average values for a minor frame; individual overheads varied from minor frame to minor frame.

over-head	predicted time (μ sec)	measured time (μ sec)	difference
IH_1	901	870	+ 3.5 %
RGD_1	144	144	0.0 %
RGD_2	1132	1364	- 17.0 %
FDIR	84	84	0.0 %
TOTAL	2264	2462	- 8.0 %

Table 7-2. Comparison of Predicted and Measured Overheads
(Average Values for a Minor Frame)

Table 7-2 shows that the overhead model is accurate for the IH_1 , RGD_1 , and FDIR overheads. However, the predicted RGD_2 overhead is 17.0% less than the observed overhead. The RGD_2 error caused the total predicted overhead to be 8% less than the total measured overhead (excluding IO).

There are several causes for the inaccuracy of the RGD_2 model. The primary cause is the model does not account for the time consumed by `send_queue` and `update_queue` when a task has no message packets to send. The overhead of making the `send_queue` call for tasks with no message packets is 5 μ sec per task (Table 6-10). The corresponding overhead for `update_queue` is 16 μ sec per task (Table 6-11). Therefore, 21 μ sec is spent for each task that doesn't have any message packets to send.

If a 0 is inserted into Equation 1 for the number of packets, the equation results in a -12 μ sec overhead to send_and_update for each task, instead of the correct 21 μ sec value. Equation 1 is a least squares line approximation to the data contained in Table 6-10 and Table 6-11. The approximation is very accurate except for the case when the number of packets equals zero. The model currently only considers send_and_update overheads for tasks that have message packets to send. To be more accurate, it should account for the overhead for tasks that have no packets.

To see the effect of this on the RGD₂ overhead, consider minor frame 1. The model predicts an RGD₂ overhead of 640 μ sec, versus the observed overhead of 804 μ sec (25.6% error). If the send_and_update overhead for the four tasks in that minor frame which had no messages is included, the predicted RGD₂ overhead becomes 724 μ sec, and the RGD₂ error is reduced to 10.0%. This reduction in the RGD₂ error can be achieved by using the following modified send_and_update queue equation:

Send_and_Update_Queue (per task)

*= 123 * number_of_packets - 12 (μ sec), if task has message packets in queue*

= 21 (μ sec), if task has no enqueued message packets

Another cause for the RGD₂ overhead error is the inaccuracy of the least square line used to predict the time to schedule RG tasks (Equation 2). The time predicted by this equation can be as much as 22% in error. For better accuracy, the time to schedule RG tasks should be determined by a second- or third-order polynomial, instead of a linear approximation. Figure 7-1 is a graphical comparison of the measured overhead associated with scheduling tasks with the least square line approximation of that data.

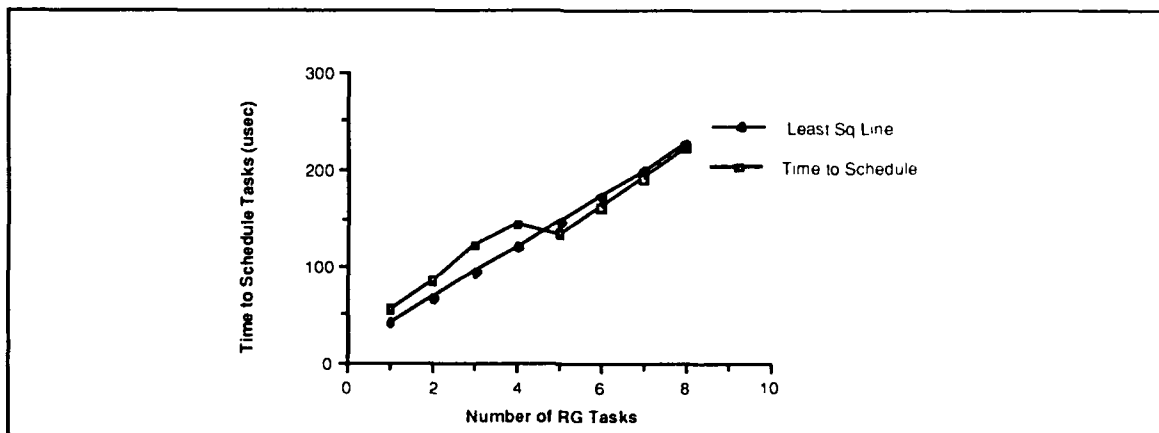


Figure 7-1. Comparison of Time to Schedule Tasks (Measured) with Least Square Line Approximation

The RGD₂ overhead is much more susceptible to inaccuracies in the model than the other overheads because the RGD₂ code contains several nested loops that can cause small errors to quickly multiply into significant ones. For example, `send_queue` and `update_queue` are called once each for every task that completed its iterative cycle during the previous minor frame. For the configuration given in this chapter, `send_queue` and `update_queue` are called 11 times each during minor frame 0. Any error in the predicted overheads for `send_queue` and `update_queue` will be multiplied by 11; thus, a small error may quickly become a significant one.

7.2.4 Other Uses of OS Overhead Model

In addition to its use in predicting overhead for a given system configuration, the OS overhead model can also be used to predict bounds on OS performance. For example, the model can be used to determine the minimum amount of OS overhead. A minimal configuration would consist of the following system tasks: Local FDIR (RG4), IOSG (RG4), IOP (RG4), IOP (RG3), IOP (RG2), IOP (RG1). One RG1 application task which did not send any messages would also be present. The OS overhead model predicts the average total OS overhead per minor frame (excluding IO) for this minimum configuration to be 698 μ sec (7% of minor frame).

Another example of using the OS overhead model is to determine the amount of message traffic which saturates the system, resulting in an OS overhead of 100%. Using the system configuration of Section 7.2.1, the overhead model predicts that the total OS overhead will exceed 100% for minor frame 0 when each of the three application tasks sends 19 message packets apiece during each RG frame.

Similarly, the model can be used to predict the number of tasks which will saturate the system. Consider the system configuration in Section 7.2.1 with each application task sending one message packet per RG frame. According to the overhead model, an additional 59 RG4 tasks (each task sends one message packet per frame) can be added to the system before the total OS overhead exceeds 100% of a minor frame.

Chapter 8

Conclusions

Fault-tolerant real-time systems demand high reliability and execute tasks within bounded time constraints. To satisfy these dual requirements, such systems must manage redundant processing resources and monitor the timing characteristics of tasks. The overhead associated with these activities uses computing time that could otherwise be used by application tasks.

The measurement of this overhead is important for several reasons. Empirical overhead data can be used to develop models which predict the operating system (OS) overhead under various configurations and workloads. Once the overhead is known, the amount of time available for executing application tasks is known. Another use of performance measurement data is to identify potential performance bottlenecks in the operating system. If measurements are done concurrently with OS development, potential bottlenecks can be eliminated in a cost effective manner rather than by waiting until later in development. Another benefit of measuring and modeling the system overhead is its use in determining where to focus future design resources.

This chapter concludes the Fault-Tolerant Parallel Processor (FTPP) performance study. Section 8.1 reviews the major contributions of this thesis, and Section 8.2 suggests areas for further study.

8.1 Major Contributions

The major findings of this thesis can be grouped in three primary categories.

1. Empirical Performance Data

- (a) A methodology for collecting performance data was developed (Chapter 5).
- (b) For the system configuration described in Chapter 6, the prototype operating system (OS) overhead was 22% of the 10 msec minor frame.
- (c) The overhead associated with task communication dominates the total operating system overhead (Chapter 6). For the system configuration

described in Chapter 6, the `scoop`, `send_queue`, and `update_queue` communication procedures accounted for 66.4% of the OS overhead, excluding IO. Scheduling contributed 29.8% of the total overhead, and the local FDIR task constituted 3.8% of the overhead.

- (d) The significance of the IO overhead is highly dependent on the amount and type of overhead (Chapter 6). The time needed by the IO Dispatcher (IOD) to transfer outgoing data to the area of memory used by the Ethernet controller is excessive. This is because the data is transferred on a byte-by-byte basis, as opposed to using a more efficient transfer. Though IO performance measurement is incomplete, the important contribution of this thesis is the implementation of the methodology for continuously evaluating performance as IO development continues.

2. OS Overhead Model

- (a) A model, based on empirical data, was developed to predict the system overhead under various configurations and workloads (Chapter 7).
- (b) For a given system configuration, the predicted overhead (excluding IO) of the model were compared with the observed overhead (Chapter 7). The predicted system overhead was 8% less than the observed overhead. The overhead prediction for part two of the Rate Group Dispatcher (RGD₂) was the primary source of the error, and suggestions on how to improve its accuracy are given in Section 7.2.3.

3. Contention Model

- (a) A model was developed to determine how contention by Processing Elements for Network Element service affects the time needed for a PE to send queued message packets (Chapter 4).
- (b) Of all the parameters associated with the contention model, the time spent by a PE transferring a packet from its local memory space to the dual-port RAM has the greatest impact on reducing the time to send queued messages (Chapter 4). This suggests that design efforts should be focused on reducing the transfer packet time to most effectively reduce the time to send queued messages. This could be accomplished by using direct memory access (DMA) to transfer data.

8.2 Suggested Further Research

The following items are suggested for further research.

1. IO performance needs to be more fully explored. The data for IO performance is incomplete because the FTPP IO System Services are not completely implemented, and the sections that have been implemented are not optimized. By using triplex or quad IO Virtual Groups, more complete data for the IO Source Congruency Manager (IOSC) and the IO Processing (IOP) tasks could be gathered.
2. More detailed data for communication procedures should be collected to pinpoint areas for improvement. Once the improvements are implemented, new data can be collected to quantify the amount of improvement in performance.
3. Once more IO and communication performance data are collected, the OS overhead model can be improved by developing equations for IOD, IOSC, and IOP. Also, the equation used to predict the RGD₂ overhead can be modified for more accuracy.
4. The results of contention model could be empirically verified. This could be done by using varying the number of PEs connected to a NE and measuring the `send_queue` execution times.
5. The contention model could be extended to account for bus contention which may occur on the receive side of the communication process. The contention model only examines the contention that can occur during the sending of messages.
6. Currently, only one prototype NE board exists. When other NEs are available, they could be used to determine how long a NE has to wait for redundant copies of message packets to arrive for voting. One of the assumptions in Section 4.2.1.3 is that the time needed for the NE to process a packet is constant. This assumes that the individual copies of a redundant message packet arrive at the NE at the same time. In reality, the processors of a given VG are slightly skewed from each other, and the NE may have to wait for the arrival of some packets. With multiple NEs, this wait could be measured and incorporated into the contention model.
7. Empirical performance data could be used to develop other analytical models. For example, the execution time of application tasks that use system calls could be estimated via static source code analysis. Work has been started in this area [Tre93].

8. The statistical analysis program could be modified to determine distributions of performance data in addition to average, maximum, minimum, and standard deviation values. This would allow the prediction of the probability of the occurrence of a frame overrun.

Appendix A

Acronyms

CID	<i>Communication Identification Number</i> A designation assigned to each task; it is used for intertask communication.
FCR	<i>Fault Containment Region</i> A collection of Processing Elements, Network Element, IO controllers, and power conditioners which are electrically and physically isolated from the rest of the system.
FDIR	<i>Fault Detection, Identification and Recovery</i> Allows FTTP to sustain multiple successive faults by identifying a faulty component and reconfiguring the system to compensate for the fault.
FMG	<i>Fault-Masking Virtual Group</i> A logical grouping of three or four processors to enhance the reliability of critical tasks.
FTTP	<i>Fault-Tolerant Parallel Processor</i> A computer designed for both high reliability and high throughput in a real-time environment.
IH	<i>Interrupt Handler</i> A software routine that executes whenever a hardware interrupt occurs.
IO	<i>Input/Output</i> The reading in or sending out of data.
IOC	<i>IO Controller</i> A device that connects FTTP to the outside world.
IOD	<i>IO Dispatcher</i> A software task which manages the execution of IO requests.
IOP	<i>IO Processing Task</i> A software task which filters multiple copies of input IO data to arrive at one valid value for use by all members of a virtual group.
IOR	<i>IO Request</i> A definition of the requested IO activity.
IOSC	<i>IO Source Congruency Manager</i> A software task which ensures that each member of a fault-masking virtual group receives a copy of an input IO data read by any other member of that virtual group.
NE	<i>Network Element</i> The hardware device which provides the connectivity among virtual groups. Its primary function is to exchange and vote message packets provided by the processing elements.

OS	<i>Operating System</i> A collection of software tasks which manage a computer's resources, schedule the execution of other tasks, and coordinate events in the computer.
PC	<i>Power Conditioner</i> A power source which provides a steady voltage, without any power surges, etc.
PE	<i>Processing Element</i> A hardware device which provides a general or special purpose processing site.
RG	<i>Rate Group</i> A set of tasks whose iteration rate is well-defined and whose execution times do not exceed the iteration frame.
RGD	<i>Rate Group Dispatcher</i> A task which is responsible for controlling the execution of the rate group tasks and providing reliable communication among the tasks throughout the system. It executes in two parts.
RGD ₁	<i>Rate Group Dispatcher (Part One)</i> Its primary functions are to check for task overruns and to schedule the IO Dispatcher for execution.
RGD ₂	<i>Rate Group Dispatcher (Part Two)</i> Its primary functions are to send queued message packets and to schedule rate group tasks for execution.
SERP	<i>System Exchange Request Pattern</i> A string of bytes describing the current state of the input and output buffers for each processor in the system. It is used to determine if packets can be sent from one virtual group to another.
VG	<i>Virtual Group</i> A grouping of one or more processors to form a virtual (possibly redundant) single processing site. All processors in a virtual group execute the same instruction stream, and each processor is located in a separate FCR.
VGID	<i>Virtual Group Identification Number</i> A numerical designation assigned to uniquely identify each virtual group in the system.

Appendix B

Contention Model Source Code

The source code used to simulate the contention model is contained in one file, SIM.C. The file contains the following seven functions:

```
main()
get_parameters()
initialize()
send_queue()
do_stats()
print_header()
print_results()
```

The source code listing for the seven functions in SIM.C follows.

```
/* **** */
/*      SIM.C      */
/*      */
/*  by Bob Clasen  */
/*    16 Mar 1993   */
/* **** */

/* updated 18 Mar 93 to account for phasing */
/* updated 23 Mar 93 to allow user to change system parameters */
/* updated  1 Apr 93 to allow round-robin and priority servicing */
/* updated  5 Apr 93 to calculate average delay */

#include <stdio.h>
#include <math.h>

/* define constants */
#define ARRAY_SIZE  200
#define MAX_NUM_PEs 8
#define ROUND_ROBIN 0
#define PRIORITY    1
#define DONE        0
#define NOT_DONE    1
#define PR_SUMMARY  0
#define PR_FULL     1
```

```

/* global variables */
int current_time;
int phasing;
int ne_status;
int pe_start;
float avg_delay;
float avg_thruput;
FILE *datafile;

/* global system parameters */
int frame_time = 10000; /* length of frame */
int serp_time = 25; /* time for NE to process SERP */
int process_pkt = 10; /* time for NE to vote and deliver packet */
int xfer_pkt = 60; /* time for PE to transfer packet to DPRAM */
int service = ROUND_ROBIN; /* type of servicing algorithm */
int num_pkts = 10; /* number of packets sent by each PE */
int num_pe = MAX_NUM_PEs; /* number of PEs connected to this NE */
int start_phase = 0;
int end_phase = 1000;
int incr_phase = 100;
int print_mode = PR_SUMMARY;

struct {
    int pkt_rdy_time;
    int wait[ARRAY_SIZE];
    int pkts_sent;
    int finish_time;
    int status;
    int delay;
    float throughput;
} pe[MAX_NUM_PEs];

/*-----*/
/*  main  */
/*-----*/

main()
{
    get_parameters();
    phasing = start_phase;

    datafile = fopen("results", "w");
    print_header();

    while ( phasing <= end_phase) {
        initialize();

        while (ne_status == NOT_DONE)
            send_queue();

        do_stats();
        print_results();

        phasing = phasing + incr_phase;
    }
    fclose(datafile);
}

```

```

/*-----*/
/* get_parameters */
/*-----*/
/* prompt user for system parameters */

get_parameters()
{
    int change;

    /* display current system parameters */
    printf("\nCURRENT SYSTEM PARAMETERS\n");
    printf("-----\n\n");
    printf("1. Stop simulation at      %d usec\n", frame_time);
    printf("2. Time to process serp    %d usec\n", serp_time);
    printf("3. Time to process pkt      %d usec\n", process_pkt);
    printf("4. Time to transfer pkt     %d usec\n", xfer_pkt);
    printf("5. Servicing algorithm      ");
    if (service == ROUND_ROBIN)
        printf("round-robin\n");
    else
        printf("priority\n");
    printf("6. Number of pkts per PE %d\n", num_pkts);
    printf("7. Print Mode              ");
    if (print_mode == PR_SUMMARY)
        printf("summary only\n");
    else
        printf("full print of wait times\n");
    printf("8. Number of PEs          %d\n", num_pe);
    printf("9. Phasing starts at      %d usec\n", start_phase);
    printf("10. Phasing ends at       %d usec\n", end_phase);
    printf("11. Phasing increment     %d usec\n", incr_phase);

    /* prompt user for any changes to system parameters */
    printf("\nEnter the number of the parameter to change ('0' when done) ");
    scanf("%d", &change);
    while (change != 0) {
        switch (change) {
            case 1:
                printf("Enter the time to stop simulation ");
                scanf("%d", &frame_time);
                break;
            case 2:
                printf("Enter the time to process serp ");
                scanf("%d", &serp_time);
                break;
            case 3:
                printf("Enter the time to process pkt ");
                scanf("%d", &process_pkt);
                break;
            case 4:
                printf("Enter the time to transfer pkt ");
                scanf("%d", &xfer_pkt);
                break;
            case 5:
                if (service == ROUND_ROBIN) {
                    service = PRIORITY;
                    printf("Servicing algorithm is now PRIORITY\n");
                }
                else {
                    service = ROUND_ROBIN;
                    printf("Servicing algorithm is now ROUND-ROBIN\n");
                }
                break;
        }
    }
}

```

```

    case 6:
        printf("Enter the number of packets sent by each PE  ");
        scanf("%d", &num_pkts);
        break;
    case 7:
        if (print_mode == PR_SUMMARY) {
            print_mode = PR_FULL;
            printf("Print mode is now FULL PRINTOUT\n");
        }
        else {
            print_mode = PR_SUMMARY;
            printf("Print mode is now SUMMARY ONLY\n");
        }
        break;
    case 8:
        printf("Enter the number of PEs  ");
        scanf("%d", &num_pe);
        break;
    case 9:
        printf("Enter the phasing to start with  ");
        scanf("%d", &start_phase);
        break;
    case 10:
        printf("Enter the phasing to end with  ");
        scanf("%d", &end_phase);
        break;
    case 11:
        printf("Enter the amount of phasing increment  ");
        scanf("%d", &incr_phase);
        break;
    default:
        printf("Value entered was invalid\n\n");
    }
    printf("Enter the number of the parameter to change ('0' when done)  ");
    scanf("%d", &change);
}

return;
}

/*-----*/
/* initialize */
/*-----*/
/* initialize pe_structure */

initialize() {

    int i;

    for (i = 0; i < num_pe; i++) {
        pe[i].pkt_rdy_time = (i * phasing) + xfer_pkt;
        pe[i].pkts_sent = 1;
        pe[i].status = NOT_DONE;
    }

    current_time = xfer_pkt;
    pe_start = -1;
    ne_status = NOT_DONE;

    return;
}

```

```

/*-----*/
/*  send_queue  */
/*-----*/
/* simulation routine */

send_queue()
{
    int i;
    int cur_serp_time;

    cur_serp_time = current_time;
    current_time = current_time + serp_time;

    /* find index of pe to start SERP with */
    if (service == ROUND_ROBIN) {
        pe_start = pe_start + 1; /* round-robin */
        if (pe_start == num_pe)
            pe_start = 0;
    }
    else
        pe_start = 0; /* priority */

    /* process the ready packets */
    i = pe_start;
    do {
        if ((pe[i].pkt_rdy_time <= cur_serp_time) && (pe[i].status == NOT_DONE)) {
            current_time = current_time + process_pkt;

            if (pe[i].pkts_sent == num_pkts) {
                pe[i].finish_time = pe[i].pkt_rdy_time;
                pe[i].status = DONE;
            }
            else {
                pe[i].wait[pe[i].pkts_sent] = current_time - pe[i].pkt_rdy_time;
                pe[i].pkt_rdy_time = current_time + xfer_pkt;
                pe[i].pkts_sent = pe[i].pkts_sent + 1;
            }
        }
        i = i + 1;
        if (i == num_pe)
            i = 0;
    } while (i != pe_start);

    /* check to see if all packets have been sent */
    ne_status = DONE;
    for (i = 0; i < num_pe; i++)
        if (pe[i].status == NOT_DONE)
            ne_status = NOT_DONE;

    return;
}

```

```

/*-----*/
/*   do_stats   */
/*-----*/
/* determines statistics: delay, throughput, and utilization */

do_stats()
{
    int i;
    float sum;

    for (i = 0; i < num_pe; i++) {
        pe[i].delay = pe[i].finish_time - (i * phasing);
        pe[i].throughput = ((float) num_pkts) / pe[i].delay * 1000000;
    }

    /* calculate average delay */
    sum = 0.0;
    for (i = 0; i < num_pe; i++)
        sum = sum + pe[i].delay;
    avg_delay = sum / num_pe;

    /* calculate average throughput */
    sum = 0.0;
    for (i = 0; i < num_pe; i++)
        sum = sum + pe[i].throughput;
    avg_thruput = sum / num_pe;

    return;
}

/*-----*/
/* print_header */
/*-----*/
/* prints header information - current system parameters */

print_header() {

    /* print header info to file */
    fprintf(datafile, "\n-----\n");
    fprintf(datafile, "SYSTEM PARAMETERS\n");
    fprintf(datafile, "-----\n\n");
    fprintf(datafile, "stop simulation at      %d usec\n", frame_time);
    fprintf(datafile, "time to process serp    %d usec\n", serp_time);
    fprintf(datafile, "time to process pkt     %d usec\n", process_pkt);
    fprintf(datafile, "time to transfer pkt    %d usec\n", xfer_pkt);
    if (service == ROUND_ROBIN)
        fprintf(datafile, "servicing algorithm    round_robin\n");
    else
        fprintf(datafile, "servicing algorithm    priority\n");
    fprintf(datafile, "number of pkts sent by each PE %d\n", num_pkts);
    fprintf(datafile, "\nnumber of PEs in system %d\n", num_pe);
    fprintf(datafile, "phasing starts at      %d usec\n", start_phase);
    fprintf(datafile, "phasing ends at        %d usec\n", end_phase);
    fprintf(datafile, "phasing increment      %d usec\n", incr_phase);

    fprintf(datafile, "\n\n\n\n\n\n-----\n");
    fprintf(datafile, "SIMULATION RESULTS\n");
    fprintf(datafile, "-----\n");

    return;
}

```

```

/*-----*/
/* print_results */
/*-----*/
/* prints simulation results to file and to screen */

print_results()
{
    int i, j;
    int mx, mn, nm;
    float av, sd;

    /* print statistics to file */
    fprintf(datafile, "\n\nSTATISTICS for Phasing = %d usec\n", phasing);
    fprintf(datafile, "-----\n\n");
    fprintf(datafile, "\t Delay \t PE Throughput\n");
    fprintf(datafile, "\t (usec)\t (pkt/sec) \n");
    fprintf(datafile, "\t ----- \t ----- \n");
    for (i = 0; i < num_pe; i++)
        fprintf(datafile, "PE %d\t %d \t %.1f\n", i, pe[i].delay, pe[i].throughput);

    /* print average values to file */
    fprintf(datafile, "\nAvg PE Delay %.1f usec ", avg_delay);
    fprintf(datafile, "(time to send %d pkts)\n", num_pkts);
    fprintf(datafile, "Avg PE Throughput %.1f pkts/sec\n\n", avg_thruput);

    /* print individual wait times to file */
    if (print_mode == PR_FULL) {
        fprintf(datafile, "\n\nWAIT TIMES for Phasing = %d\n", phasing);
        fprintf(datafile, "-----\n\n");
        for (i = 0; i < num_pe; i++) {
            fprintf(datafile, "\nPE %d wait times\n", i);
            for (j = 1; j < pe[i].pkts_sent; j++)
                fprintf(datafile, " %d\n", pe[i].wait[j]);
        }
    }

    /* print individual wait times to screen */
    if (print_mode == PR_FULL) {
        printf("\n\nWAIT TIMES for Phasing = %d\n", phasing);
        printf("-----\n\n");
        for (i = 0; i < num_pe; i++) {
            printf("\nPE %d wait times\n", i);
            for (j = 1; j < pe[i].pkts_sent; j++)
                printf(" %d\n", pe[i].wait[j]);
        }
    }

    /* print statistics to screen */
    printf("\n\nSTATISTICS for Phasing = %d usec\n", phasing);
    printf("-----\n\n");
    printf("\t Delay \t PE Throughput\n");
    printf("\t (usec)\t (pkt/sec) \n");
    printf("\t ----- \t ----- \n");
    for (i = 0; i < num_pe; i++)
        printf("PE %d\t %d \t %.1f\n", i, pe[i].delay, pe[i].throughput);

    /* print average values to screen */
    printf("\nAvg PE Delay %.1f usec ", avg_delay);
    printf("(time to send %d pkts)\n", num_pkts);
    printf("Avg PE Throughput %.1f pkts/sec\n\n", avg_thruput);

    return;
}

```

Appendix C

Statistical Analysis Source Code

The Statistical Analysis source code consists of four files. The names of the files and the functions contained in each file are listed below.

STATS.C

- main()

MEAS.H

- system structures and definitions

MEAS.C

- sort_data()
- crunch()
 - stand_dev()
 - avg()
 - max()
 - min()
 - print_results()

OH_INFO.C

- initialize()
- get_oh_info()

The source code listings for each of these files is given on the following pages.

```

/*****
/*      STATS.C      */
/*      */
/*      by Bob Clasen      */
/*      */
/*      25 Mar 1993      */
*****/

#include <stdio.h>

/* link with meas and oh_info */

/* global variables */
int  st_label, end_label;
int  mode;
int  info_min, info_max, info_incr;
char info_str[12];
char fname[12];
char oh_name[15];

FILE *datafile;

main()
{
    int  num_entries;

    /* set up default statistical parameters */
    initialize();

    /* prompt user for which overhead to analyze */
    get_oh_info();

    /* read and sort data from datafile */
    num_entries = sort_data();

    /* crunch numbers and display results */
    crunch( num_entries );
}

```

```

/*****
/*  MEAS.H  */
*****/

/* Header file for performance measurement programs */

/* constants */
#define TIME_CONST 1.28 /* 1.28 usec per clock tick */
#define DEBUG_OVHD 22 /* 22 usec overhead per call to debug log */
#define ARRAY_SIZE 3500
#define INT_MAX 2147483647
#define TABLE_SIZE 11

/* used to indicate measurement mode */
#define NORMAL 0 /* normal mode - no intermediate values to subtract */
#define SUBTRACT 1 /* subtract mode - subtract out given intermediate values */
#define RETRIEVE 2 /* retrieve mode - used when measuring retrieve times */
#define SIMPLE 3 /* simple mode - don't sort by info field */

/* structure definitions */

struct log_entry {
    unsigned short int label;
    unsigned short int info;
    unsigned time;
};

struct results_entry {
    float avg; /* average value */
    int max; /* maximum value */
    int min; /* minimum value */
    int num; /* number of samples */
    float std; /* standard deviation */
    int inf; /* data stored in 'info' field */
};

extern int st_label;
extern int end_label;
extern int mode;
extern int info_min;
extern int info_max;
extern int info_incr;
extern char info_str[];
extern char fname[];
extern char oh_name[];

```

```

/*****
/*      MEAS.C      */
/*      */
/*  by Bob Clasen  */
/*      */
/*   10 Aug 1992   */
/*****/
/* updated 16 Sep 92 to include overall average and overall stand dev */
/* updated 15 Oct 92 to allow program to prompt user for data file name */
/* updated 16 Oct 92 to allow simple crunch and better summary at end */
/* updated 28 Jan 93 to allow retrieve data to have proper msg size */
/* 29 Jan 1993 use info field of second entry, not first */
/*  4 Feb 1993 remove entries with negative times (impl for scoop) */
/* 25 Mar 1993 remove SUBTRACT mode; use global variables */
/* 26 Mar 1993 cleaned up "print results" into separate function */

/* This file contains C functions used by performance measurement programs */

#include <stdio.h>
#include <math.h>
#include <string.h>
#include "meas.h"

float stand_dev();
float avg();
int  max();
int  min();

/* global data structures shared between 'sort_data' and 'crunch' */
int time[ARRAY_SIZE];
int info[ARRAY_SIZE];

struct results_entry results[25];
int  num_info_entries;
int  overall_max, overall_min;
float overall_avg, overall_sd;

/*****/
/*      sort_data      */
/*****/

/* This function opens the file containing debug data and reads the */
/* requested data into an array.  It returns the number of entries */
/* in the array. */

int sort_data()
{
    struct log_entry temp;
    int i, j, num_entries;
    unsigned start_time;
    int flag, overhead[ARRAY_SIZE];
    char data_fname[20];

    FILE *datafile;

    /* open file and check that it's valid */
    printf("\n\nEnter name of file containing debug data ('d' for debug.dat)\n");
    printf("      -->  ");

```

```

scanf("%s", data_fname);
printf("\n\n");
if ((data_fname[0] == 'd') && (data_fname[1] == '\0'))
    strcpy( data_fname, "debug.dat");
datafile = fopen( data_fname, "r");
if (datafile == NULL) {
    printf("ERROR - can't open file %s\n\n", data_fname);
    exit(1);
}

/* initialize arrays */
for (i = 0; i < ARRAY_SIZE; i++) {
    time[i] = 0;
    info[i] = 0;
}

/* read in data from datafile until EOF */
i = 0;
flag = 0;
while (fscanf(datafile, "%x %x %x\n", &(temp.label), &(temp.info),
    &(temp.time)) == 3) {
    if (temp.label == st_label) {
        start_time = temp.time;
        flag = 1;
    }
    if ((flag == 1) && (temp.label != end_label)) {
        /* calculate overhead in making nested debug log calls */
        overhead[i] = overhead[i] + DEBUG_OVHD;
    }

    if (temp.label == end_label) {
        info[i] = temp.info;

        /* calculate delta time in microseconds */
        time[i] = (int)((temp.time - start_time)*TIME_CONST - overhead[i]);

        /* throw out any negative times */
        if (time[i] > 0)
            i++;

        overhead[i] = 0;
        flag = 0;
    }
}

/* clean up file */
fclose(datafile);

/* return number of entries */
return(i);
}

/*****
/*          crunch
*****/
/* This function crunches the data that was extracted via sort_data().
/* It calculates average, max value, min value, and standard deviation.
/* It also displays results on the screen and to a file.

crunch(num_entries)
int num_entries;
{

```

```

int loop_count;
int num, total, mintime, maxtime;
int i, j, k;
float avg_time, st_dev;
int sd_array[ARRAY_SIZE];

/* crunch numbers */
if (mode == NORMAL) {
    loop_count = info_min;
    k = 0;
    while (loop_count <= info_max) {
        num = 0;
        total = 0;
        mintime = INT_MAX;
        maxtime = 0;

        for (j = 0; j < num_entries; j++) {
            if (info[j] == (unsigned short int) loop_count) {
                num = num + 1;

                /* calculate total accumulated time */
                total = total + time[j];

                /* store time in array to be used to calculate standard deviation */
                sd_array[num-1] = time[j];

                /* determine max and min time */
                if (time[j] > maxtime)
                    maxtime = time[j];
                if (time[j] < mintime)
                    mintime = time[j];
            }
        }

        if (num != 0) {
            /* determine averages and standard deviation */
            avg_time = total / num;
            st_dev = stand_dev( sd_array, num, avg_time );

            /* store results */
            results[k].avg = avg_time;
            results[k].max = maxtime;
            results[k].min = mintime;
            results[k].num = num;
            results[k].std = st_dev;
            results[k].inf = loop_count;
        }

        /* increment loop_count by the appropriate amount */
        loop_count = loop_count + info_incr;
        k++;
    }
    num_info_entries = k;
}

/* Calculate overall average and standard deviation */
overall_avg = avg( time, num_entries);
overall_sd = stand_dev( time, num_entries, overall_avg);
overall_max = max( time, num_entries);
overall_min = min( time, num_entries);

print_results(num_entries);
return;
}

```

```

/*-----*/
/*-----*/
/*  functions used by crunch  */
/*-----*/

/*-----*/
/*          stand_dev          */
/*-----*/

/* This function calculates the standard deviation of a data array */

float stand_dev( d_array, size, mean )
    int d_array[];
    int size;
    float mean;
{
    int i;
    float diff, sum;
    float st_dev;

    if (size == 0)
        return(0.0);

    sum = 0.0;
    for (i = 0; i < size; i++) {
        diff = d_array[i] - mean;
        sum = sum + (diff * diff);
    }

    st_dev = sqrt( sum / size );

    return (st_dev);
}

/*-----*/
/*          avg          */
/*-----*/
/* This function returns the average value of an array of integers */

float avg( d_array, size )
    int d_array[];
    int size;
{
    int i, sum;
    float average;

    sum = 0;
    for (i = 0; i < size; i++)
        sum = sum + d_array[i];
    average = sum / size;

    return (average);
}

```

```

/*-----*/
/*          max          */
/*-----*/
/* This function returns the maximum value of an array of integers */

int max( d_array, size )
    int d_array[];
    int size;
{
    int i;
    int maxtime;

    maxtime = 0;
    for (i = 0; i < size; i++)
        if (d_array[i] > maxtime)
            maxtime = d_array[i];

    return (maxtime);
}

/*-----*/
/*          min          */
/*-----*/
/* This function returns the maximum value of an array of integers */

int min( d_array, size )
    int d_array[];
    int size;
{
    int i;
    int mintime;

    mintime = INT_MAX;
    for (i = 0; i < size; i++)
        if (d_array[i] < mintime)
            mintime = d_array[i];

    return (mintime);
}

/*-----*/
/*    print_results      */
/*-----*/

print_results(num_entries)
int num_entries;
{
    int i, j, k;
    int num;
    int loop_count;
    char full_report;
    FILE *datafile;

    /* open file for results and print some header information */
    datafile = fopen( fname, "w" );
    fprintf(datafile, "-----\n");
    fprintf(datafile, "          %s\n", oh_name);
    fprintf(datafile, "START:  %x          END:  %x\n", st_label, end_label);
    fprintf(datafile, "-----\n");
    fprintf(datafile, "          (all times in usec)\n\n");
    printf("-----\n");

```

```

printf("          %s\n", oh_name);
printf("START:  %x          END:  %x\n", st_label, end_label);
printf("-----\n");
printf("-----RESULTS-----\n");
printf("          (all times in usec)\n\n");
if (mode == SIMPLE)
    printf("\n");
else {
    /* print summary of results (sorted for NORMAL mode) */
    printf(" %7s %10s %10s %10s %10s %10s\n", info_str, "avg time", "st dev",
        "max time", "min time", "samples");
    fprintf(datafile, "-----SUMMARY OF RESULTS-----\n\n");
    fprintf(datafile, " %7s %10s %10s %10s %10s %10s\n", info_str, "avg time",
        "std dev", "max time", "min time", "samples");
    for (i = 0; i < num_info_entries; i++) {
        fprintf(datafile, " %7d %10.0f %10.0f %10d %10d %10d \n", results[i].inf,
            results[i].avg, results[i].std, results[i].max,
            results[i].min, results[i].num);
        printf(" %7d %10.0f %10.0f %10d %10d %10d \n", results[i].inf,
            results[i].avg, results[i].std, results[i].max,
            results[i].min, results[i].num);
    }
}

/* print overall results (for both SIMPLE and NORMAL modes */
printf(" \n\nOVERALL DATA for %s\n\n", oh_name);
fprintf(datafile, " \n\nOVERALL DATA for %s\n\n", oh_name);
printf(" %7s %7s %7s %7s %9s\n", "avg", "st_dev", "max", "min", "samples");
printf(" %7.0f %7.0f %7d %7d %9d\n\n",
    overall_avg, overall_sd, overall_max, overall_min, num_entries);
fprintf(datafile, " %7s %7s %7s %7s %9s\n", "avg", "st_dev", "max",
    "min", "samples");
fprintf(datafile, " %7.0f %7.0f %7d %7d %9d\n\n",
    overall_avg, overall_sd, overall_max, overall_min, num_entries);

/* prompt user to see if full report wanted */
printf("Do you want each data sample to be written to the file ");
printf("( 'y' or 'n' )? --> ");
scanf("%s", &full_report);

if (full_report == 'y') {
    fprintf(datafile, " \n\nSORTED LISTING OF EACH DATA SAMPLE\n\n");

    /* print results to file for SIMPLE mode */
    if (mode == SIMPLE) {
        fprintf(datafile, " %7s %10s\n", "num", "time");
        for (j = 0; j < num_entries; j++)
            fprintf(datafile, " %7d %10d\n", j+1, time[j]);
        fprintf(datafile, " \n\n");
    }
}

```

```

else {
    /* print results to file for NORMAL mode */
    k = 0;
    loop_count = info_min;
    while (loop_count <= info_max) {
        fprintf(datafile,"%s %d\n", info_str, loop_count);
        fprintf( datafile, "%7s %10s\n", "num", "time");

        num = 0;
        for (j = 0; j < num_entries; j++) {
            if (info[j] == (unsigned short int) loop_count) {
                fprintf(datafile,"%7d %10d \n", num, time[j]);
                num++;
            }
        }

        /* print results to datafile */
        fprintf(datafile,"----- p");
        fprintf(datafile,"AVG TIME: %.0f usec MAX %d MIN %d \n",
            results[k].avg, results[k].max, results[k].min);
        loop_count = loop_count + info_incr;
        k++;
        fprintf(datafile,"\n\n");
    }
}
fclose( datafile );
return;
}

```

```

/*****
/*  OH_INFO.C  */
*****/

/* This file contains the following functions: */
/* -- initialize  set up default parameters for overhead analysis */
/* -- get_oh_info  prompts user to specify which overhead to analyze */

#include <string.h>
#include "meas.h"

struct {
    char oh_name[15];      /* name of this overhead */
    int  st_label;
    int  end_label;
    int  mode;
    int  info_min;
    int  info_max;
    int  info_incr;
    char info_str[12];
    char fname[12];
} oh_table[TABLE_SIZE];

/*-----*/
/*  initialize  */
/*-----*/

initialize()
{
    /* scoop parameters */
    strcpy(oh_table[0].oh_name, "SCOOP PKTS");
    oh_table[0].st_label = 0x7070;
    oh_table[0].end_label = 0x7f7f;
    oh_table[0].mode = NORMAL;
    oh_table[0].info_min = 1;
    oh_table[0].info_max = 13;
    oh_table[0].info_incr = 1;
    strcpy(oh_table[0].info_str, "num pkts");
    strcpy(oh_table[0].fname, "scoop.dat");

    /* rgd1 parameters */
    strcpy(oh_table[1].oh_name, "RG DISP (Pt1)");
    oh_table[1].st_label = 0xd0d0;
    oh_table[1].end_label = 0xd1d1;
    oh_table[1].mode = NORMAL;
    oh_table[1].info_min = 0;
    oh_table[1].info_max = 7;
    oh_table[1].info_incr = 1;
    strcpy(oh_table[1].info_str, "frame");
    strcpy(oh_table[1].fname, "rgd1.dat");

    /* rgd2 parameters */
    strcpy(oh_table[2].oh_name, "RG DISP (Pt2)");
    oh_table[2].st_label = 0xd2d2;
    oh_table[2].end_label = 0xd7d7;
    oh_table[2].mode = NORMAL;
    oh_table[2].info_min = 0;
    oh_table[2].info_max = 7;
    oh_table[2].info_incr = 1;
    strcpy(oh_table[2].info_str, "frame");
    strcpy(oh_table[2].fname, "rgd2.dat");
}

```

```

/* send_queue parameters */
strcpy(oh_table[3].oh_name, "SEND_QUEUE");
oh_table[3].st_label = 0x9090;
oh_table[3].end_label = 0x9999;
oh_table[3].mode = NORMAL;
oh_table[3].info_min = 0;
oh_table[3].info_max = 11;
oh_table[3].info_incr = 1;
strcpy(oh_table[3].info_str, "pkts sent");
strcpy(oh_table[3].fname, "send.dat");

/* update_queue parameters */
strcpy(oh_table[4].oh_name, "UPDATE_QUEUE");
oh_table[4].st_label = 0x8080;
oh_table[4].end_label = 0x8989;
oh_table[4].mode = NORMAL;
oh_table[4].info_min = 0;
oh_table[4].info_max = 11;
oh_table[4].info_incr = 1;
strcpy(oh_table[4].info_str, "pkts sent");
strcpy(oh_table[4].fname, "update.dat");

/* fdir(local) parameters */
strcpy(oh_table[5].oh_name, "FDIR (local)");
oh_table[5].st_label = 0xfd00;
oh_table[5].end_label = 0xfd0f;
oh_table[5].mode = SIMPLE;
oh_table[5].info_min = 0;
oh_table[5].info_max = 0;
oh_table[5].info_incr = 0;
strcpy(oh_table[5].info_str, " ");
strcpy(oh_table[5].fname, "fdir.dat");

/* iod parameters */
strcpy(oh_table[6].oh_name, "IO DISPATCHER");
oh_table[6].st_label = 0xc0c0;
oh_table[6].end_label = 0xcfcf;
oh_table[6].mode = SIMPLE;
oh_table[6].info_min = 0;
oh_table[6].info_max = 0;
oh_table[6].info_incr = 0;
strcpy(oh_table[6].info_str, " ");
strcpy(oh_table[6].fname, "iod.dat");

/* iosd parameters */
strcpy(oh_table[7].oh_name, "IO SRCE CONG");
oh_table[7].st_label = 0x6060;
oh_table[7].end_label = 0x6969;
oh_table[7].mode = SIMPLE;
oh_table[7].info_min = 0;
oh_table[7].info_max = 0;
oh_table[7].info_incr = 0;
strcpy(oh_table[7].info_str, " ");
strcpy(oh_table[7].fname, "iosd.dat");

/* iop parameters */
strcpy(oh_table[8].oh_name, "IO PROC TASK");
oh_table[8].st_label = 0x6a6a;
oh_table[8].end_label = 0x6f6f;
oh_table[8].mode = SIMPLE;
oh_table[8].info_min = 0;
oh_table[8].info_max = 0;
oh_table[8].info_incr = 0;

```

```

strcpy(oh_table[8].info_str, " ");
strcpy(oh_table[8].fname, "iop.dat");

/* queue parameters */
strcpy(oh_table[9].oh_name, "QUEUE_MSG");
oh_table[9].st_label = 0xf1f1;
oh_table[9].end_label = 0xf2f2;
oh_table[9].mode = NORMAL;
oh_table[9].info_min = 0;
oh_table[9].info_max = 1000;
oh_table[9].info_incr = 100;
strcpy(oh_table[9].info_str, "size");
strcpy(oh_table[9].fname, "q.dat");

/* retrieve parameters */
strcpy(oh_table[10].oh_name, "RETRIEVE_MSG");
oh_table[10].st_label = 0xf5f5;
oh_table[10].end_label = 0xf6f6;
oh_table[10].mode = NORMAL;
oh_table[10].info_min = 0;
oh_table[10].info_max = 1000;
oh_table[10].info_incr = 100;
strcpy(oh_table[10].info_str, "size");
strcpy(oh_table[10].fname, "retr.dat");

return;
}

/*-----*/
/*  get_oh_info  */
/*-----*/

get_oh_info()
{
    int i;
    int selection;
    int parameter;

    /* prompt user to select which overhead to analyze */
    printf("\n\nDEFAULT OVERHEAD ANALYSIS PARAMETERS\n");
    printf("-----\n\n");
    printf("%4s %9s %9s %5s %5s %6s %6s\n", "Num", "Name", "Start", "End",
        "Min", "Max", "Incr");

    for (i = 0; i < TABLE_SIZE; i++) {
        printf("%4d %13s %5x %5x ", i, oh_table[i].oh_name, oh_table[i].st_label,
            oh_table[i].end_label);
        if (oh_table[i].mode == NORMAL)
            printf("%4d %6d %4d\n", oh_table[i].info_min, oh_table[i].info_max,
                oh_table[i].info_incr);
        else
            printf("\n");
    }
    printf("%4d  Modify default parameters or use new parameters\n", i);

    printf("\nEnter the selection number --> ");
    scanf("%d", &selection);
    printf("\n");
}

```

```

/* copy selected parameters into global variables */
if (selection != TABLE_SIZE) { /* use default values */
    strcpy(oh_name, oh_table[selection].oh_name);
    st_label = oh_table[selection].st_label;
    end_label = oh_table[selection].end_label;
    mode = oh_table[selection].mode;
    info_min = oh_table[selection].info_min;
    info_max = oh_table[selection].info_max;
    info_incr = oh_table[selection].info_incr;
    strcpy(info_str, oh_table[selection].info_str);
    strcpy(fname, oh_table[selection].fname);
}

else { /* modify default parameters or prompt for new ones */
    printf("\nEnter the number of OH whose parameters should change --> ");
    scanf("%d", &selection);

    strcpy(oh_name, oh_table[selection].oh_name);
    st_label = oh_table[selection].st_label;
    end_label = oh_table[selection].end_label;
    mode = oh_table[selection].mode;
    info_min = oh_table[selection].info_min;
    info_max = oh_table[selection].info_max;
    info_incr = oh_table[selection].info_incr;
    strcpy(info_str, oh_table[selection].info_str);
    strcpy(fname, oh_table[selection].fname);

    /* prompt for which parameters to change */
    parameter = 10;
    while (parameter != 0) {
        printf("\n\nCurrent Parameters for %s\n", oh_table[selection].oh_name);
        printf("1. oh_name      %s\n", oh_name);
        printf("2. start label %x\n", st_label);
        printf("3. end label   %x\n", end_label);
        printf("4. info_min    %d\n", info_min);
        printf("5. info_max    %d\n", info_max);
        printf("6. info_incr   %d\n", info_incr);
        printf("7. info_str    %s\n", info_str);
        printf("8. fname       %s\n", fname);
        printf("Enter the parameter number ('0' when done) --> ");
        scanf("%d", &parameter);

        switch (parameter) {
            case 0:
                break;
            case 1:
                printf("New oh_name --> ");
                scanf("%s", oh_name);
                break;
            case 2:
                printf("New start label --> ");
                scanf("%x", &st_label);
                break;
            case 3:
                printf("New end label --> ");
                scanf("%x", &end_label);
                break;
            case 4:
                printf("New info_min --> ");
                scanf("%d", &info_min);
                break;

```

```
        case 5:
printf("New info_max --> ");
scanf("%d", &info_max);
break;
        case 6:
printf("New info_incr --> ");
scanf("%d", &info_incr);
break;
        case 7:
printf("New info_str --> ");
scanf("%s", info_str);
break;
        case 8:
printf("New fname --> ");
scanf("%s", fname);
break;
        default:
printf("Selection was invalid \n");
    }
}

return;
}
```

Appendix D

References

- [Abl88] T. Abler, *A Network Element Based Fault Tolerant Processor*, MS Thesis, Massachusetts Institute of Technology, Cambridge, MA, May 1988.
- [Bab90] C. Babikyan, "The Fault Tolerant Parallel Processor Operating System Concepts and Performance Measurement Overview", *Proceedings of the 9th Digital Avionics Systems Conference*, October 1990, pp. 366-371.
- [Dol82] D. Dolev, "The Byzantine Generals Strike Again", *Journal of Algorithms*, 1982, pp. 14-30.
- [Dol84] D. Dolev, C. Dwork, and L. Stockmeyer, "On the Minimal Synchronism Needed for Distributed Consensus", IBM Research Report RJ 4294 (46990), 8 May 1984.
- [Fis82] M. Fischer and N. Lynch, "A Lower Bound for the Time to Assure Interactive Consistency", *Information Processing Letters*, 13 June 1982, pp. 183-186.
- [Har87] R. Harper, *Critical Issues in Ultra-Reliable Parallel Processing*, PhD Thesis, Massachusetts Institute of Technology, Cambridge, MA, June 1987.
- [Har88a] R. Harper, J. Lala, and J. Deyst, "Fault Tolerant Parallel Processor Overview", *18th International Symposium on Fault Tolerant Computing*, June 1988, pp. 252-257.
- [Har88b] R. Harper, "Reliability Analysis of Parallel Processing Systems", *Proceedings of the 8th Digital Avionics Systems Conference*, October 1988, pp. 213-219.
- [Har91] R. Harper and J. Lala, "Fault Tolerant Parallel Processor", *Journal of Guidance, Control, and Dynamics*, May-June 1991, pp. 554-563.
- [Joh89] B. Johnson, *Design and Analysis of Fault Tolerant Digital Systems*, Addison-Wesley, 1989.
- [Lam82] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem", *ACM Transactions on Programming Languages and Systems*, July 1982, pp. 382-401.
- [Leh89] T. Lehr, et. al., "Visualizing Performance Debugging", *Computer*, October 1989, pp. 38-51.

- [Pal85] D. Palumbo and R. Butler, "Measurement of SIFT Overhead", NASA Technical Memorandum 86722, Langley Research Center, Hampton, VA, April 1985.
- [Pea80] M. Pease, R. Shostak, and L. Lamport, "Reaching Agreement in the Presence of Faults", *Journal of the ACM*, April 1980, pp. 228-234.
- [Tre93] S. Treadwell, *Estimating Task Execution Delay in a Real-Time System via Static Source Code Analysis*, MS Thesis, Massachusetts Institute of Technology, Cambridge, MA, May 1993.