A Fault-Tolerant
Shared Memory System Architecture
for a Byzantine Resilient Computer

by

Bryan P. Butler

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements
for the degree of
Master of Science
at the
Massachusetts Institute of Technology
June 5, 1989
© Bryan P. Butler, 1989

The author hereby grants to MIT permission to reproduce and to
distribute copies of this thesis document in whole or in part.

Signature of Author

Bryan Butler
Department of Electrical Engineering and Computer Science; May 12, 1989

Certified By

Thomas F. Knight, Jr.
Thesis Supervisor, Massachusetts Institute of Technology

Approved By

Richard E. Harper
Thesis Supervisor, Charles Stark Draper Laboratory

Accepted By

Arthur C. Smith
Chairman, Committee on Graduate Students
A Fault-Tolerant
Shared Memory System Architecture
for a Byzantine Resilient Computer

by

Bryan P. Butler

June 5, 1989

Abstract

The memory requirements for ultra-reliable computers are expected to increase due to future increases in mission functionality and operating-system requirements. This increase will have a negative effect on the reliability and cost of the system. Increased memory size will also reduce the ability to reintegrate a channel after a transient fault, since the time required to reintegrate a channel in a conventional fault-tolerant processor is dominated by memory realignment time.

In this paper, a Byzantine Resilient Fault-Tolerant Processor with Fault-Tolerant Shared Memory (FTP/FTSM) is presented as a solution to these problems. The FTSM uses an encoded memory system, which reduces the memory requirement by one-half compared to a conventional quad-FTP design. This increases the reliability and decreases the cost of the system. The realignment problem is also addressed by the FTSM. Because any single error is corrected upon a read from the FTSM, a faulty channel's corrupted memory does not need realignment before reintegration of the faulty channel. A background scrubbing operation is proposed to prevent the accumulation of transient errors in the memory. With a hardware-implemented scrubber, the scrubbing cycle time, and therefore the memory fault latency, can be upper-bounded at a small value. This technique increases the reliability of the memory system and facilitates validation of its reliability model.

Thesis Supervisor: Thomas F. Knight, Jr.
Title: Professor of Electrical Engineering
Acknowledgements

I would like to thank Dr. Richard Harper and the staff of the Fault-Tolerant Systems Division at the Charles Stark Draper Laboratory for their support and enthusiasm for this project. I would also like to thank Dr. Thomas Knight for his assistance and contribution.

This work was done at the Charles Stark Draper Laboratory, Inc. with the support of NASA Langley Research Center under contract NAS1-18565.

Publication of this report does not constitute approval by the Charles Stark Draper Laboratory of the findings or conclusions contained herein. It is published for the exchange and stimulation of ideas.

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

Bryan Butler

Charles Stark Draper Laboratory hereby grants permission to the Massachusetts Institute of Technology to reproduce and distribute this thesis in whole or in part.
Table of Contents

1. Introduction 9
   1.1. Problem Statement and Background 9
   1.2. Objective 10
   1.3. Approach 10
   1.4. Organization 10

2. Background Information 11
   2.1. Fundamentals of Byzantine Fault Tolerance 11
   2.2. Non-Byzantine Resilient Architectures 13
       2.2.1. Space Shuttle Computers 14
       2.2.2. Triple Modular Redundancy 15
   2.3. Existing Fault-Tolerant Computers 17
       2.3.1. Fault-Tolerant Multi-Processor 17
       2.3.2. The (4,2)-Concept FTP 19
       2.3.3. The VLSI-FTP 20
       2.3.4. Network-Element Fault-Tolerant Processor 22
   2.4. Shortcomings of existing FTP designs 23

3. FTP/FTSM Overview 27
   3.1. Introduction to architecture 27
   3.2. Comparison to a quad-FTP 28
   3.3. Coding 30
       3.3.1. Encoding Procedure 31
       3.3.2. Decoding Procedure 32
       3.3.3. Use of (4,2) Code as a Fault-Masking Function 33

4. FTP/FTSM Functional Description 39
   4.1. Data Exchanges 39
       4.1.1. Class I 39
           4.1.1.1. Class I Write 39
           4.1.1.2. Class I Read 40
           4.1.1.3. Cache Burst 41
       4.1.2. Class II 42
           4.1.2.1. Class II Write 42
           4.1.2.2. Class II Read 43
   4.1.3. Memory Scrubbing 44
   4.1.4. Command Exchanges 45
   4.2. Synchronization 46
       4.2.1. Steady-State Synchronization 48
       4.2.2. Initial Synchronization 50

5. FTP/FTSM Architecture 53
   5.1. Fault-Tolerant Shared Memory 53
       5.1.1. Memory Controller/Address Register 55
       5.1.2. Encoder/Decoder 57
       5.1.3. RAM Modules 59
       5.1.4. Inter-FCR Communication System 59
       5.1.5. FTSM Controller 62
   5.2. Fault-Tolerant Clock 62
   5.3. Command Exchange Mechanism 65
5.4. Synchronous I/O Bus
5.5. I/O Processor 68
5.6. Processing Elements 69
5.7. Local Memory
   5.7.1. PROM 71
   5.7.2. RAM 71
5.8. Miscellaneous Devices 71
5.8.1. Multi-Function Peripheral 72
5.8.2. Control Registers
   5.8.2.1. Syndrome Registers 73
   5.8.2.2. Command Registers 74
   5.8.2.3. FTSM Control Registers 75

6. Analysis of FTP/FTSM
6.1. Reliability
   6.1.1. Combinatorial Model 77
   6.1.2. Markov Model 80
   6.1.3. Parameter Study Using Markov Model
   6.1.3.1. FTSM Encoded vs. Quadruplicated Memory 83
   6.1.3.2. Reliability vs. Memory Size 84
   6.1.3.3. Reliability vs. Scrub Rate 84
6.2. Performance
   6.2.1. Baseline System Performance Analysis 88
   6.2.2. NEFTP Performance Analysis 90
   6.2.3. VLSI-FTP Performance Analysis 91
   6.2.4. FTP/FTSM Performance Analysis 92
   6.2.5. Performance Analysis Results 95
   6.2.6. Performance Improvement of the FTP/FTSM 97

7. Future Enhancements
   7.1. Multiprocessor/Parallel Processor Architectures 99
   7.2. Input/Output Processing 100
   7.3. External Caches 101
   7.4. Event-Driven I/O 103

8. Conclusion and Recommendations 105

9. Appendices
   9.1. Galois Fields 107
   9.2. Bus Cycles 108
   9.3. Controller Microcode 110
   9.4. Memory Controller/Address Register Logic 125
   9.5. Fault-Tolerant Clock 130
   9.6. Analysis Procedures and Parameters 132
   9.7. Glossary 140
   9.8. References 144
## List of Figures

1. **Introduction**

2. **Background Information**
   - Figure 2-1  Space Shuttle Computer System Architecture
   - Figure 2-2  TMR Architecture with a Simplex Sensor
   - Figure 2-3  FTP/FTSM Architecture
   - Figure 2-4  (4,2)-Concept Fault-Tolerant Computer
   - Figure 2-5  VLSI-FTP Architecture
   - Figure 2-6  NEFTP Architecture

3. **FTP/FTSM Overview**
   - Figure 3-1  Physical Configuration of FTP/FTSM
   - Figure 3-2  Effective Configuration of FTP/FTSM
   - Figure 3-3  Majority Vote Fault-Masking Function
   - Figure 3-4  (4,2) Code Fault-Masking Function
   - Figure 3-5  Quad-FTP Source Congruency; Scenario 1
   - Figure 3-6  Quad-FTP Source Congruency; Scenario 2
   - Figure 3-7  Quad-FTP Source Congruency; Scenario 3

4. **FTP/FTSM Functional Description**
   - Figure 4-1  Function Code Table
   - Figure 4-2  Class I Write
   - Figure 4-3  Class I Read
   - Figure 4-4  Class II Write
   - Figure 4-5  Class II Read
   - Figure 4-6  Fault-Tolerant Clock Cycle
   - Figure 4-7  Fault-Tolerant Clock Adjustments

5. **FTP/FTSM Architecture**
   - Figure 5-1  FTP/FTSM FCR Architecture
   - Figure 5-2  FTP/FTSM Data Paths
   - Figure 5-3  Memory Controller/Address Register
   - Figure 5-4  IFC Messages
   - Figure 5-5  FTC Block Diagram
   - Figure 5-6  Synchronization of THEIRFTC
   - Figure 5-7  Command Exchange Mechanism
   - Figure 5-8  FTP/FTSM Memory Map
   - Figure 5-9  FTP/FTSM Control/Status Registers
   - Figure 5-10 Syndrome Register
   - Figure 5-11 Syndrome Address Register
   - Figure 5-12 Command Register
   - Figure 5-13 Watchdog Timer Control Register
   - Figure 5-14 Decoder Mode Register

6. **Analysis of FTP/FTSM**
   - Figure 6-1  Combinatorial Reliability Graph
   - Figure 6-2  Markov Model
   - Figure 6-3  Reliability of Encoded vs. Quad Memory
   - Figure 6-4  Reliability vs. Memory Size
   - Figure 6-5  Memory Access Distribution
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-6</td>
<td>Reliability vs. % Fast Memory</td>
<td>85</td>
</tr>
<tr>
<td>6-7</td>
<td>Reliability vs. Scrub Rate</td>
<td>86</td>
</tr>
<tr>
<td>6-8</td>
<td>Control Loop Iteration for Flight Control</td>
<td>88</td>
</tr>
<tr>
<td>6-9</td>
<td>Control Loop for NEFTP</td>
<td>90</td>
</tr>
<tr>
<td>6-10</td>
<td>Control Loop for VLSI-FTP</td>
<td>91</td>
</tr>
<tr>
<td>6-11</td>
<td>Control Loop for FTP/FTSM</td>
<td>93</td>
</tr>
<tr>
<td>6-12</td>
<td>Overhead vs. Iteration Rate</td>
<td>95</td>
</tr>
<tr>
<td>6-13</td>
<td>Task Completion Ratios</td>
<td>96</td>
</tr>
</tbody>
</table>

7. Future Enhancements
   Figure 7-1 FTP/FTSM Multiprocessor Architecture | 100  
   Figure 7-2 Using the NEFTP as an IOP           | 101  

8. Conclusion and Recommendations                 | 105  
1. Introduction

1.1. Problem Statement and Background

Highly reliable computer systems have many applications in life-critical and mission-critical situations. These applications include use in aircraft, spacecraft, sea-going vehicles, and chemical and nuclear plants. A number of fault-tolerant processors have been designed with the goal of providing the necessary reliability for these applications.

Fault-tolerance has been an issue for many years. The first generation of fault-tolerant processors (FTP) which were designed according to the rules of Byzantine resilience were the fault-tolerant multiprocessor (FTMP), designed by the Charles Stark Draper Laboratory, and the software-implemented fault-tolerant computer (SIFT), designed by SRI International. These two designs demonstrated the concept of Byzantine resilience. However, because the overhead imposed by the fault-tolerance was high and the programming models were complex, the systems were impractical for actual real-time application.

A second generation of Byzantine resilient computers has been developed which can meet both the reliability and the performance requirements for imbedded real-time applications. These systems include the processor/interstage architecture of the AIPS-FTP and the VLSI-FTP, and the network element architecture of the FTPP and the NEFTP. All of these designs were developed at Charles Stark Draper Laboratory. While these systems are suitable for real applications, they do present other problems. Increasing the memory capacity of these systems to enhance their functionality will result in an increase in the system cost and a decrease in the system reliability. Also, the increase in memory size will reduce the ability to reintegrate a channel after a transient fault, since the time required to reintegrate a channel in a conventional fault-tolerant processor is dominated by memory realignment time.

Fault-tolerance has also been applied to memory systems. Most fault-tolerant memory systems use an error-correcting code for informational redundancy, requiring significantly less additional memory than simple replication schemes. The well-known single-bit correcting Hamming codes have been used extensively in large mainframe and minicomputers. The use of a fault-tolerant shared memory system for a Byzantine resilient computer was investigated in [1]. More recently, the (4,2)-computer, designed by Philips Research Laboratory, was presented which applies an error-correcting code to a system very similar in architecture to a quadruplicated Byzantine resilient fault-tolerant computer.
1.2. Objective

The objective of this thesis is to design a fault-tolerant computer which solves the reliability, cost, and reintegration problems encountered by the second generation of Byzantine resilient fault-tolerant computers. Since the design is to be Byzantine resilient, a complete understanding of the impact of, and requirements for, Byzantine resilience is necessary. The computer will need to provide informational redundancy with more efficiency than simple replication, requiring the development of a suitable coding scheme. The design should not significantly compromise previous gains in reliability or performance, so analysis of the resulting design must be performed to ensure that these two parameters are maintained.

1.3. Approach

This thesis presents the conceptual design for a fault-tolerant computer which attempts to solve the memory-related problems of the second generation of Byzantine resilient fault-tolerant computers without compromising the performance and reliability improvements of these designs over the first generation FTPs. The design presented within uses a fault-tolerant shared memory as the central reliability enhancement device. A combination of techniques from the theories of Byzantine resilience and error-correcting codes is utilized by the design.

1.4. Organization

The presentation of the design study is as follows: Section 2 investigates the fundamentals of fault-tolerance and the requirements for Byzantine resilience. Several existing fault-tolerant computer architectures are also presented. Section 3 introduces the architecture of the fault-tolerant computer with fault-tolerant shared memory, the primary subject of this thesis. This section also presents the coding algorithm used by the fault-tolerant shared memory. The various fault-tolerance functions are presented in section 4. A description of each major subsection of the computer appears in section 5. The content of section 5 emphasizes the impact of each subsection on the maintenance of Byzantine resilience in the system. The computer system is analyzed with respect to reliability and performance in section 6. Finally, section 7 contains some possible enhancements to the architecture to improve the performance of the system.
2. Background Information

The primary objective of a fault-tolerant computers is to survive faults by containing and isolating their effects. Failures of one component should not cause failures in other components. Two approaches have been taken to achieve this goal. The first approach is to enumerate and estimate the likelihood of failure modes for each component. These components are then used in such a way that the more likely failure modes will not precipitate faults in adjacent components. Complexity of computer systems and human biases make this approach extremely difficult and expensive with dubious results.

A more universal approach can be taken if no assumptions are made about possible failure modes. This is the approach known as Byzantine resilience. Fault isolation is obtained by physical and electrical isolation of groups of components into fault-containment regions (FCR). The failure of one component within one FCR may cause the failure of other components within that FCR, but cannot induce faults in another FCR.

This section discusses the theoretical requirements for Byzantine resilience. A few non-Byzantine resilient architectures are presented to demonstrate the importance of Byzantine resilience. Also included are some existing fault-tolerant computer architectures and a discussion of some of their shortcomings.

2.1. Fundamentals of Byzantine Fault Tolerance

A Byzantine resilient fault-tolerant computer must be able to prevent catastrophic system failure by preventing faults from propagating throughout the system. Fault propagation in an FTP is controlled by partitioning the system into fault-containment regions (FCR). A device suffering from a physical failure can affect another device through electrical, magnetic, or mechanical interference. Fault propagation by physical means is prevented by mechanical and electrical isolation of FCRs so that while faults can propagate freely within an FCR, they cannot affect another FCR.

Error propagation occurs when a faulty FCR emits corrupted data to another FCR. If a functional recipient FCR does not react the same as other functioning FCRs, that FCR may appear faulty. Informational redundancy and a fault-masking function are used to prevent corrupt data from debasing a functioning FCR. Redundant information is delivered to an FCR from other FCRs. The recipient FCR applies the fault-masking function to the redundant data, thereby masking a given number of erroneous data items.
The theoretical requirements for Byzantine resilience have been demonstrated in a number of studies, including [2], [3], [4], and [5]. An F-Byzantine resilient system, able to tolerate the loss of F fault containment regions, must meet the following requirements:

- 3 F + 1 fault containment regions are required (cardinality requirement)
- Each FCR must be connected to at least 2 F + 1 other FCRs by independent links (connectivity requirement)
- F + 1 rounds of exchange are required to distribute single-source data (source congruency)
- The functioning FCRs must be synchronized to within a known skew (synchronization requirement).

The requirement for 3 F + 1 FCRs is necessary to provide a sufficient number of FCRs such that, during the source congruency, the source FCRs will not be able to increase the weight of its response with respect to the response of other FCRs. To prevent the source FCR from having undue influence on the system, the source is prohibited from participating in any except the first communication round. The result is that, after F + 1 rounds of communication, only 2 F + 1 values remain for each FCR to resolve using a fault-mask function. Since this is the minimum number of values required for a majority decision in the case of F faults, the 3 F + 1 requirement is demonstrated.

The 2 F + 1 requirement is necessary so that each functioning FCR will receive an unambiguous majority during message exchanges in the presence of F faults. If one considers a functioning FCR connected to fewer than this minimal requirement, that FCR could receive F faulty values from faulty FCRs. Since the connectivity of the receiving FCR is less than or equal to 2 F, a clear majority can not necessarily be determined. Consequently, the FCR may not make the same decision as other functioning FCRs, resulting in system loss.

The source congruency operation, described in detail by [6], is necessary to distribute single source data between processors. Certain values, such as input from I/O devices or information regarding system health, are often only known to one processor. In order to act on these values, the value must be distributed to all members of the redundant group. The primary requirement is that all functioning FCRs receive the same value, even in the presence of errors on behalf of the source processor. If the source processor is faulty, the correct value
may be lost, but at least the functioning processors will make consistent decisions based on the input value.

FCR synchronization is required because asynchronous protocols have the potential for non-termination. A functioning FCR which receives messages using an asynchronous protocol from another FCR is unable to tell in finite time whether that FCR is simply excruciatingly slow in broadcasting a message or whether the FCR is failed. The functioning FCR would have to wait an indefinite amount of time to find out. Therefore, system loss could be caused by the loss of one FCR, since the other FCRs would wait indefinitely for the failed FCR and would not be able to continue. The solution is to bound the tolerable skew between FCRs and any FCR which is not within this bound is failed by definition.

2.2. Non-Byzantine Resilient Architectures

The following paragraphs describe two architectures which do not meet the minimum requirements for single fault Byzantine resilience. Each architecture will be discussed with respect to which requirements are not met, and how each is susceptible to single-point failures. For the discussion, a single-point failure is an isolated failure of any single component in the system. Byzantine resilient systems are not susceptible to any arbitrary single-point failures. Therefore, if a single point failure mode can be imagined which causes system loss in a particular architecture, that architecture is proven to be non-Byzantine resilient. However, it should be noted that the apparent lack of such a failure mode does not imply Byzantine resilience.

Consider the problem of distributing single source data to all processors. This task is often the downfall of non-Byzantine resilient architectures. Single source data originates in a single processor but must be distributed so that all functioning processors have a consistent value before acting on the value. An example of single source data is input from an external sensor or system health monitor.

The problem of distributing single source data in a Byzantine resilient architecture is solved by a source congruency algorithm. The primary goal of the source congruency is to ensure that all functioning FCRs reach agreement, which implies that they all receive a consistent value. The secondary goal is to ensure that if the source FCR is functioning, all functioning FCRs reach validity, i.e. they all receive the correct value. The goal of agreement is necessary to ensure that all functioning FCRs make the same decision based on the input value. The goal of validity ensures that functioning FCRs receive valid data if it is available.
2.2.1. Space Shuttle Computers

The computer system on the Space Shuttle uses four computers in a quadruplicated configuration with a fifth computer available as an active spare simplex computer that can be switched in manually under crew control\(^7\). The computers are connected to I/O systems through a network of redundant bit-serial buses. Some of the I/O units are flight-critical and are represented in quadruplicate. Non-flight-critical components are simplex devices. This discussion will focus on the quadruplicated set and its use as an autonomous flight-critical system.

A simplified architecture is shown in figure 2-1. Each computer is connected to the other computers by a quadruplicated redundant bus. Quadruplicated I/O units are connected to each computer through a similar redundant bus system. One of the computers is designated under software control to operate a member of a redundant bus group in the command mode. The other computers operate the same bus in the listen mode. In the command mode a computer can send a command or data to an I/O device or a processor and can read data sent by an I/O device. In the listen mode, the computer only receives data on that bus line.

![Diagram of Space Shuttle Computer System Architecture](image)

Figure 2-1: Space Shuttle Computer System Architecture

14
If each computer in the Space Shuttle system were considered to be a separate FCR, this system would satisfy the cardinality and connectivity requirements. However, there is some question as to processor isolation, since each processor can seize control of every bus it is connected to. Since the assignment of command or listen mode to each bus is a software function, a faulty processor could conceivably switch every bus to which it is connected into the command mode. If this were done to the inter-computer redundant bus, each computer would be effectively cut off from the others, and there would be no way to perform informational exchange and reach agreement.

Processor synchronization is performed by a software synchronization task using the inter-computer links, so the synchronization requirement is also satisfied, given that the inter-computer bus is functioning.

The system has no provision for performing a source congruency exchange. An example of the procedure for reading input data from an I/O device is as follows: the command processor sends a command to an input device to return an input value. The input device reads the value from the sensor it is connected to and broadcasts the value to the four computers through the shared bus. The critical assumption is that since each processor is connected to the same bus, they will all receive the same data. However, this is not always the case. At least one example was observed on the Space Shuttle system where improper termination of the shared bus resulted in two computers receiving one value and the other two receiving another value\(^8\). Without a source congruency operation, this situation can not be resolved, so the computers are not guaranteed to reach agreement on input values.

The inconsistency problem could be solved with another round of exchange between processors. Inter-computer redundant buses provide the necessary interconnect for this purpose. However, the system is so severely loaded that the additional processor throughput needed to perform the exchange is not available\(^9\).

### 2.2.2. Triple Modular Redundancy

The triple modular redundancy architecture, or TMR, is often used for highly-reliable systems. Figure 2-2 shows a TMR system with an input sensor connected to one FCR. The TMR architecture is fully connected, i.e. each FCR is connected to all other FCRs, but it does not meet the cardinality requirement for Byzantine resilience.
Figure 2-2: TMR Architecture with a Simplex Sensor

The source congruency operation illustrates the deficiencies of the TMR system. The source congruency operation in the TMR proceeds as follows: first the source FCR broadcasts an input value to each of the other two FCRs. In the second round, the recipient FCRs exchange the value they received from the source FCR and also reflect the value back to the source FCR. The source FCR does not participate in the second round. Following the second round, each FCR votes the resulting data set using a voting algorithm. The purpose of the voting algorithm is to mask out single errors and return a value which represents the majority. Since it is assumed that failed processors are in the minority, the voting algorithm should return a valid value if a majority of the input values are valid.

In one possible scenario, FCR A is faulty. In this case, validity is unachievable; there is no guarantee that FCR A will send the correct value to any FCR. In fact, FCR A could be failed in such a way that it sends different data to FCRs B and C. Assume for a moment that A sends the value X to B and the value Y to C. During the second round, B and C exchange their values and also send them back to A. Following the second round, each functioning FCR has the data set (X,Y). Since each FCR uses the same voting algorithm, agreement is guaranteed.

In another case, FCR B could be faulty. Validity should be achievable in this situation since the source FCR is functional. First, FCR A reads an input value M from the sensor. During the first round of exchange, M is broadcast to B and C. During the second round, B and C exchange values. Since B is faulty, it may do anything. Assume for a moment that it sends the value X to A and Y to C. A does not participate in the second round. C transmits the value M to both A and B since it is functional. Following the second round, the FCRs have the following data sets:
FCR A  ( M X )
FCR B  ( ? ? )
FCR C  ( M Y )

Since the two functioning FCRs (A and C) do not have the same data set, the fault-masking function must return the correct value M to achieve validity. However, this is not possible, since three values are necessary to determine a majority. Therefore, the functioning FCRs have no way to guarantee validity or agreement on single-source data in the presence of an arbitrary single point failure.

One important aspect of the TMR approach is that agreement and validity are obtainable for commonly-sourced output. In this situation, each processor in the TMR group calculates an output based on one or more input values (assume for a moment that some mechanism exists for reaching agreement and validity with input data). Since each processor has calculated the output value, the locally generated value can be used in addition to the two copies obtained from the other FCRs. Three copies of data are sufficient to determine a majority in the presence of a single fault.

The implication of this property is that of the four FCRs required for 1-Byzantine resilience, only three of them must contain active processors. This property has been exploited by a number of designs which use a very minimal implementation for the fourth FCR in an attempt to enhance system reliability. The justification is that the reliability can be increased by using less complex hardware. Other systems with four processors take advantage of this property by allowing reconfiguration around a failed processor. If a processor fails without affecting the data exchange mechanisms in the FCR (a situation not guaranteed by Byzantine resilience, but nevertheless a conceivable condition), the processor can be effectively removed from the system without any loss of fault coverage.

2.3. Existing Fault-Tolerant Computers

Several designs for fault-tolerant computers have emerged from a variety of sources. Four of these designs are discussed below.

2.3.1. Fault-Tolerant Multi-Processor

The Fault-Tolerant Multi-Processor (FTMP) is a design developed in 1978 by the Charles Stark Draper Laboratory (CSDL) in Cambridge, Massachusetts. The FTMP is composed of a number of line-replaceable units, or LRUs. Each LRU contains a processing unit and a
The processor and memory units are arbitrarily grouped into tripex redundant groups and are connected by a redundant bus. The memory appears to be a triplicated memory shared by all processor redundant groups. The current FTMP implementation contains ten LRUs, each of which contains a processor and 16K of memory. This configuration allows three processor redundant groups and 48K of fault-tolerant shared memory with one spare LRU that can be used to replace any member of one of the redundant groups, should one of those members fail.

Figure 2-3: FTMP Architecture

Figure 2-3 shows an abridged FTMP architecture with P processing units and M memory units. Each processing unit and memory unit is connected to a shared bus system through a bus guardian unit (BGU). The BGU isolates each unit so that if a unit fails, that failure can
not induce faults in any other units. Note that the actual number of processing and memory units is unlimited except by increased contention for the shared bus system. The current FTMP implementation has ten processing units, ten memory units of 16K each, and a 5-way redundant shared bus. The processing units and memory units are arbitrarily grouped into triplex redundant groups. Three of the five lines are activated to comprise the shared bus. Reconfiguration following the occurrence of faults is possible since an extra processor and memory are included, as well as two extra shared buses.

Each processing and memory unit has a three-way voter which collects data from the three active bus lines. When a processing group writes a value to the shared memory, each processor in the group (after obtaining control of the shared bus) writes data to a different line on the shared bus. Each of the memory units takes the three incoming data values, votes them, and writes the result to memory. A read cycle is similar. The three memory units read the requested data value out of memory and each presents the value on one of the active bus lines. Each processor votes the three incoming values and loads the result into one of the processor's internal registers.

2.3.2. The (4,2)-Concept FTP

The (4,2) computer\textsuperscript{13} was designed at Philips Research Laboratories in the Netherlands. The general architecture is shown in figure 2-4. The system is intended for applications in telephone switching networks.

The (4,2) system is composed of four slices, which roughly correspond to fault-containment regions. Each slice contains a processor and memory. Interposed between the processor and memory is an encoder/decoder pair. The encoder/decoder pair uses a (4,2) code to generate a four-symbol code for each data byte that is written to memory. Each symbol is four bits wide, so one 8-bit word requires 16-bits of the distributed memory system. The code is designed so that the loss of any one symbol does not result in the loss of data. Each slice stores a different symbol, so that if one of the slices is lost, the other three can still recover the data and continue to function.
When a value is read from the memory, each slice places its symbol on a distributed bus system which broadcasts the symbol to the other slices. Each slice takes the three symbols from the other slices plus its own local symbol and uses the decoder to regenerate the data word while masking out any errors which may have appeared in the code word.

One important aspect of the (4,2) computer architecture is that it is not Byzantine resilient. It appears to satisfy all of the requirements except for the $F + 1$ round source congruency operation. I/O is performed by simply distributing a copy of the input data to each slice. A failure mode could be imagined which would cause the I/O device to send a different value to each of the four slices. This could result in processor divergence, since the processors have no way to reach agreement. Also, there is no capability for distributing the contents of the fault register for use by fault detection and isolation software.

2.3.3. The VLSI-FTP

The VLSI-FTP is the most recent incarnation of the processor/interstage architecture employed by the Charles Stark Draper Laboratory's AIPS system\(^{14}\). The VLSI-FTP architecture is shown in figure 2-5. Unlike earlier AIPS FTP designs which use triplicated hardware, the VLSI-FTP uses quadruplicated processors. Note that the fourth processor/interstage pair is not required to satisfy the requirements for Byzantine resilience. However, its inclusion improves reliability of the VLSI-FTP over the traditional triplex
configuration by giving it fail-operational/fail-operational/fail-safe reconfiguration capability. This capability means that the VLSI-FTP will be operational following two non-simultaneous failures and will be able to shut down safely after the third non-simultaneous failure.

The processors in the VLSI-FTP are micro-synchronized. A separate exchange mechanism is used to distribute a fault-tolerant clock (FTC) among the fault-containment regions. Each FCR compares the median of the incoming FTCs with its own local FTC to maintain synchronization. If an FCR perceives itself to be ahead or behind the majority of the other FCRs, an appropriate adjustment to the local FTC will be made. When an FCR makes an adjustment to its local FTC, the frequency of the processor clock in that FCR is also adjusted momentarily so that each FTC cycle always contains exactly the same number of processor clock cycles. Each processor must receive the exact same number of clock cycles during an FTC cycle to ensure synchronous behavior.

The message exchanges are clocked by the fault-tolerant clock. During the first half of the FTC cycle the processors send an 8-bit message to the interstages. This message is clocked into the interstage on the falling edge of FTC. During the second half, the processors
broadcast another 8-bit message while the interstages send the first message back to the processors. As a result, one 16-bit message can be sent every FTC cycle. Since the frequency of the FTC is nominally 888 kHz, the maximum data exchange rate in the VLSI-FTP is 1.776 Mbytes/sec.

For commonly sourced messages, each processor only broadcasts to one interstage, namely the directly adjacent interstage. The interstages re-broadcast the message to all four of the processors, which vote the four messages they receive. If the message exchange is a source congruency operation, the source processor broadcasts to all four interstages. Note that the vote and source congruency operations require the same amount of time.

2.3.4. Network-Element Fault-Tolerant Processor

One of the recent FTP designs to emerge from the Charles Stark Draper Laboratory is the Network-Element Fault-Tolerant Processor, or NEFTP\textsuperscript{15}. This design, shown in figure 2-6, is derived directly from the requirements for Byzantine resilience. The basic reliability enhancement unit is the network element, which is designed to satisfy the requirements for Byzantine fault-tolerance. The network elements, of which there is one in each of the four FCRs, are tightly synchronized to each other, and are responsible for all message passing between FCRs. The processing elements, which are functionally synchronized\textsuperscript{16}, communicate with each other through the network element.

![Network-Element Fault-Tolerant Processor Diagram](image)

**Figure 2-6:NEFTP Architecture**

The network elements (NE) perform message exchanges at the request of the processing elements (PE). Since the PEs are not tightly synchronized, some method must be used to ensure that message exchanges do not occur until all functioning PEs have requested the
exchange. This mechanism is the SERP exchange, performed at the beginning of every NE fault-tolerant clock cycle. The SERP, or system exchange request pattern, is exchanged through a proven source congruency operation\textsuperscript{17}. Each NE is responsible for providing its own SERP to the other NEs. After the four SERPs are exchanged (one for each NE), a decision is made regarding which messages to exchange. A message will not be exchanged until all non-faulty processors request its transmission. Since all functioning NEs have the same set of SERPs following the SERP exchange, each will make a consistent decision about which messages to transmit.

Message passing is invoked explicitly from software using the SEND function. The parameters for SEND include the message (from 16 to 240 bytes), the message length, the message class (output vote, source congruency), and the source processor (if the message is to be exchanged using the source congruency). Reception of messages is performed by the SCOOP function. Executing SCOOP retrieves all messages transmitted since the previous SCOOP. SCOOP does not return until all messages have been received, thus making SCOOP a synchronizing act. Functioning processors can maintain synchronization by executing SCOOP periodically.

2.4. Shortcomings of existing FTP designs

While the FTP designs described above are highly reliable, they have some shortcomings which may make them unsuitable for some applications.

One of the major shortcomings of most designs is the large amount of memory required. All the systems mentioned, except the (4,2) computer, use informational redundancy in the form of multiple copies. To ensure fault tolerance, at least three, and sometimes four, copies of each piece of data are kept. The result is an enormous increase in the amount of memory required compared to a simplex computer.

Memory systems dominate two aspects of fault tolerant computers: reliability\textsuperscript{18} and cost\textsuperscript{19}. With system functionality expected to increase greatly, much larger memory systems will be required and further deterioration of these two parameters can be expected. Therefore, it would be desirable to find a system which will tend to offset this trend.

Another difficulty with most FTP designs is that following a transient error, the memory in the failed FCR must be realigned before the FCR can be reintegrated into the system. Since Byzantine resilience makes no assumptions about failure modes, the memory contents of an FCR following a transient error are indeterminate. Therefore, to reintegrate the failed FCR,
the entire contents of memory (at least the segments used by the currently active tasks) must be exchanged, voted, and written to memory to guarantee consistency. Since most systems require an explicit vote operation to be invoked by the executing program, the amount of time required to perform this realignment is usually on the order of seconds for memory systems larger than 1 megabyte, making reintegration difficult, if not impossible, in a high-throughput real-time system.

The FTMP system and the (4,2) system do not require immediate memory realignment. In the FTMP system, vote operations are performed implicitly on every access to a triplicated memory group. The (4,2) computer provides implicit fault masking through the use of the decoder during memory reads. The result is that all functioning processors will receive correct data following a memory read, even if one of the processors' local memory is corrupted. Consequently, memory does not have to be realigned before a failed processor is reintegrated. If one of the replicated memory systems is corrupted, it will be masked out by the other functioning units during every memory access. Of course, realignment is necessary to recover the fault masking properties of the redundant group, but this realignment can take place concurrently with application task execution, rather than before resumption of the application.

Another potential problem with conventional FTPs is fault latency and the accumulation of uncorrectable errors. In a large memory system, some locations may be accessed very rarely. Since error correction can only occur if a memory location is accessed, these rarely accessed locations have a high probability of accumulating multiple errors which are uncorrectable. In an FTP requiring an explicit vote operation, errors may go undetected for long periods of time.

This thesis presents the design of an FTP called the Fault-Tolerant Processor with Fault-Tolerant Shared Memory (FTP/FTSM). This FTP uses the following features to solve the reintegration, fault latency, and uncorrectable error problems:

- An encoding scheme to provide Byzantine resilient informational redundancy at a cost of only doubling the amount of memory required.

- A memory scrubber to periodically exchange and correct every location in memory. A hardware implementation allows this process to be interleaved with processor accesses, thus removing the scrubbing burden from the processor.
• An implicit fault-masking operation (voting). Fault-masking is performed every time the processor makes access to the memory system, as in the FTMP. Because extreme delays in performing the voting operation severely restrict performance, the latency of the operation must be kept to a minimum.
3. FTP/FTSM Overview

The design presented below is a fault-tolerant processor which uses a fault-tolerant shared memory system. To the processing elements, the fault-tolerant shared memory (FTSM) appears to be a simplex, highly reliable shared memory to which all processing elements have access. In reality, the FTSM is distributed among the fault containment regions with physical and electrical isolation between the FCRs to eliminate all single-point failure modes. The FTSM is designed to meet all of the requirements for Byzantine resilience.

3.1. Introduction to architecture

The FTP/FTSM is a highly reliable computer system which makes use of a fault-tolerant shared memory (FTSM) system to enhance reliability. The FTSM is partitioned into pieces, called quadrants, such that the loss of any quadrant will not cause the loss of data or the functionality of the computer. This configuration is shown in figure 3-1. Each quadrant of the FTSM is contained in a separate FCR. If one quadrant of the FTSM is lost, the remaining three quadrants contain enough information to reconstruct all data values.

![Figure 3-1: Physical Configuration of FTP/FTSM](image)

The FTSM meets all of the requirements for single fault Byzantine resilience for the following reasons. The FTSM is divided into quadrants, each of which is contained by a separate FCR, to conform to the cardinality requirement. The connectivity requirement is satisfied by an inter-FCR communication system (IFC) which connects every quadrant to all other quadrants. The source congruency operation is a function performed by the FTSM at the processors' request. A fault-tolerant clock (FTC) is distributed by the IFC to perform FCR synchronization at the micro-instruction level.
The FTSM system can be viewed as a Byzantine resilient shared memory system to which four processors are connected. This view, shown in figure 3-2, is the programming model of the FTP/FTSM. Since each processor is connected to the FTSM, all processors can read a value from a location in memory. If the FTSM quadrant connected directly to a given processor is functional but the memory storage device has corrupted data stored at the requested location, that processor will obtain correct data. When the processors attempt to access the location, the FTSM quadrants will exchange their locally stored symbols with all other quadrants. The coding scheme used by the FTSM is designed so that a symbol from one quadrant can be arbitrarily corrupted without corrupting the resulting data. Since this exchange operation is performed for every read access to the FTSM, correct data is always assured in the presence of one corrupted symbol.

![Figure 3-2: Effective Configuration of FTP/FTSM](image)

3.2. Comparison to a quad-FTP

The design of the FTP/FTSM is similar in many respects to the canonical quad-FTP design. Both systems are composed of four fault-containment regions, the minimum number required for 1-Byzantine fault resilience.

A Byzantine resilient FTP uses a fault-masking function to prevent propagation of corrupted data. The simplest and most popular fault-masking function is the bitwise majority vote. At least three incoming data items are required to perform a majority vote in the presence of no more than one error. In the bitwise majority vote, each bit position in the incoming data items is independently compared, and the bit value (either 0 or 1) represented by the majority of copies is selected as the correct bit value.
The majority vote function is shown in figure 3-3. To perform a majority vote, three (or four, in the case of a quad-FTP) copies of data are kept. Each copy is stored in an isolated location so that if one copy is lost, the remaining copies will constitute a majority from which the data can be recovered. This scheme requires total storage on the order of 3n or 4n and inter-FCR communication bandwidth of order n, where n is the width of the data. Also, if a bitwise majority vote is used, the FCRs must be capable of generating bit-for-bit identical results in the presence of no errors.

![Diagram of majority vote fault-masking function](image)

**Figure 3-3: Majority Vote Fault-Masking Function**

Other fault-masking functions are available besides the bitwise majority vote. The fault-masking function used by the (4,2) computer and the FTP/FTSM is a four-symbol minimum-distance separable (MDS) error correcting code. An encoder generates, from a data word, four symbols which provide informational redundancy for the data word. Each symbol is stored by a different FCR. Fault-masking is performed by a decoder. The decoder uses the four symbols to regenerate the data word. If one of the symbols is corrupted, the decoder will still be able to recover the data from the three remaining symbols.

The use of the encoding scheme as a fault-masking function is demonstrated in figure 3-4. A data word of size $n$ can be stored as four symbols, each of size $n/2$, with only three symbols needed to recover the data in the presence of a single random error. The resulting storage requirements are of order $2n$, and the communication bandwidth is of order $n/2$. 

29
Some of the advantages of this architecture compared to a quad-FTP are the following:

- Less memory is required, resulting in increased reliability and decreased cost.
- No memory realignment is necessary during reintegration of a failed FCR.
- Fault latency is reduced.
- Bandwidth requirements for inter-FCR communication links are reduced.

3.3. Coding

The FTSM uses an encoding scheme, known as the (4,2) code, to encode data words before they are stored into the RAM of the FTSM. The (4,2) code is the same code described in [20] for use in the (4,2)-concept FTP. The encoding process takes two 4-bit symbols which represent a data word and generates four 4-bit symbols. The code is designed to tolerate any single symbol loss. Each FCR stores a different symbol so that if one FCR is lost, the data word can be reconstructed from the three remaining symbols.

The generation of the (4,2) code begins with the definition of a Galois Field of $2^4$, or 16, elements. This definition includes an addition and a multiplication operation, both of which are closed over GF($2^4$). The addition operation defines an abelian group, and the multiplication operation defines a commutative monoid (every element except zero has a multiplicative inverse). The arithmetic tables for these operations are shown in appendix 9.1.

The code is designed to tolerate a number of error conditions. In one mode of operation, known as the random mode, the code can tolerate the loss of any symbol (from one to four bits in error) or the loss of any two bits (in the same or different symbols). Another mode,
known as erasure mode, allows an additional bit error to be corrected in the presence of a known symbol in error. In this mode, one of the four symbols is suspected of being wrong. Note that it is not necessary to know the value of the error; only the position of the error must be known. A third mode, known as duplex mode, allows the extraction of data when two symbols are suspect. The duplex mode is incapable of masking any errors beyond the two suspect symbols.

It should be noted that the (4,2) code is not the most efficient code with respect to the number of bits required to implement it. The (4,2) code requires order(n) additional bits, whereas a SEC-DED (single error correction-double error detection) requires only order(\log_2 n) additional bits. However, a SEC-DED code can only tolerate the loss of a single bit. To use a SEC-DED code for the type of application proposed in this report, a separate fault-containment region would be required to store each bit of the code, otherwise bit failures would not necessarily be independent. Considering the cost of supporting an FCR, including physical isolation, electrical isolation, and separate power supplies, it is clear that SEC-DED codes are not appropriate for this type of application.

3.3.1. Encoding Procedure

The code word is generated by first dividing a byte into two 4-bit symbols. These two symbols form the data vector, B. The data vector can be represented by the following:

\[
B = \begin{bmatrix} b_1 & b_0 \end{bmatrix}
\]

The code word is generated using the generator matrix, G:

\[
G = \begin{bmatrix} 1 & 0 & \alpha^{11} & \alpha^7 \\ 0 & 1 & \alpha^7 & \alpha^{11} \end{bmatrix}
\]

Each element of the generator matrix is a member of GF(2^4). To generate the code vector, C, the data vector is multiplied by the generator matrix:

\[
C = B \cdot G = \begin{bmatrix} b_1 & b_0 \end{bmatrix} \begin{bmatrix} b_1 \alpha^{11} + b_0 \alpha^7 \\ b_1 \alpha^7 + b_0 \alpha^{11} \end{bmatrix}
\]

The matrix-vector multiplication is performed using exactly the same procedure as traditional matrix arithmetic, except that the additive and multiplicative operations defined over GF(2^4) are substituted for traditional numerical addition and multiplication. Each processor selects a
different symbol from C to store in its local part of the FTSM. The selection is made based on the processor's unique FCR identification number (FCR-ID).

Note that the code word contains the two data symbols and two parity check symbols. This property is characteristic of a separable code and implies that if the code word, C, is known, the data vector can be recovered by simply discarding the two parity check symbols. No final decryption operation is required.

3.3.2. Decoding Procedure

During a read cycle, the four symbols are exchanged between FCRs. Each FCR should receive three symbols from the other FCRs in addition to the locally stored symbol. These four symbols make up the received vector, R. R can be modeled as the transmitted vector, C, plus an error vector, E. If no errors have occurred, C=R and E=0. If an error has occurred, and the error condition is correctable as described above, the error vector, and therefore the original transmitted vector, can be determined.

The first step in determining the error vector is to multiply the received vector by the parity check matrix, H. The form of the parity check matrix is:

\[
H = \begin{bmatrix}
11 & 7 \\
\alpha & \alpha \\
7 & 11 \\
\alpha & \alpha \\
1 & 0 \\
0 & 1 
\end{bmatrix}
\]

The result of this operation is a 2-symbol vector known as the syndrome. If the syndrome is zero (i.e. each symbol in the syndrome vector is zero), no correctable errors are present. If the syndrome is non-zero, the syndrome can be used to determine the error vector, as shown by the following derivation:

\[
C \cdot H = [0 \ 0] \ \forall \text{ valid } C
\]

\[
S = R \cdot H = (C + E) \cdot H = E \cdot H
\]

Each correctable error pattern for a given code operation mode (random, erasure, or duplex) maps to a unique syndrome. Therefore, if the syndrome is known, the corresponding error vector can be determined. Note that if the actual error vector does not meet the conditions
required by the particular mode of operation, the error vector determined by the decoder can
not be the actual error vector, and the original code word can not be recovered.

Since each element of \( \text{GF}(2^4) \) is its own additive inverse, the following is true:

\[
R = E + C; \quad C' = E + R
\]

\( C' = C \) if the error condition meets the necessary criteria (i.e. single-symbol or double-bit
error, single bit error in the presence of a suspect symbol, or no errors in the presence of two
suspect symbols). Since the code is separable, the data symbols can be extracted directly
from \( C' \). Note that if the error conditions do not meet the criteria, the resulting code word,
\( C' \), will not necessarily be valid.

3.3.3. Use of (4,2) Code as a Fault-Masking Function

The (4,2) code presented here can be used as a fault-masking function in place of the
traditional bitwise majority vote function found in most fault-tolerant computers. While the
effect of the code is similar to the majority vote function, a few subtle differences exist which
must be thoroughly investigated to ensure that the FTP/FTSM design is Byzantine resilient.

A candidate fault-masking function must guarantee agreement and validity in the presence of
arbitrary errors to be useful in a Byzantine resilient computer. Validity is assured by a fault-
masking function which will return a particular value if a sufficient number of inputs to the
function indicate that that value is the correct one. Agreement is reached by a fault-masking
function if different instances of the function which are presented with identical data return
the same result.

The majority vote function requires at least three redundant copies as inputs to determine an
unambiguous output in the presence of a single error. More than three copies can be used, as
long as only one is assumed to be in error. The (4,2) code requires exactly four symbols as
inputs, of which one symbol can be in error. Another difference between the two functions is
that each copy in a voted system contains all information necessary to reconstruct the data. If
any given copy is determined to be correct, the data value is readily available. For the
encoded system, two symbols are required to recover the value of the data object.

Two message exchange classes are of concern for determining the usefulness of a candidate
fault-masking function. The first is the class I exchange, which is performed by the FCRs to
reach agreement on a commonly sourced output value. This situation occurs when the FCRs
have completed some calculation on identical input data. In a quadruplicated FTP with majority voting, each of the four FCRs exchanges a copy of the output value with all other FCRs. Each FCR has four copies of the value to reduce through a majority vote function. Assuming that at most one FCR is faulty, at least three of these values will be in agreement. Since the vote function requires three valid copies for consistent results with four inputs in the presence of at most one error, agreement by all functioning FCRs is guaranteed.

A possible implementation of the (4,2) code for a class I exchange is for each FCR to exchange one of the four symbols generated by encoding the output data value using the generator matrix described in section 3.3.1. Each FCR chooses a different symbol to exchange so that, collectively, all four symbols are present in the system. Each FCR will receive the four different symbols, one from himself and three from other FCRs. If at most one FCR is faulty, at most one of the symbols will be corrupted. Since the other three symbols were generated by functioning FCRs from valid data, the valid data can be recovered by the functioning FCRs. Since all functioning FCRs can reach validity in the presence of a single arbitrary fault, this implementation of the class I exchange is acceptable for a Byzantine resilient processor.

A second message exchange situation occurs for single-sourced data. This exchange, known as a class II, or source congruency, is necessary to guarantee that functioning FCRs can reach agreement even in the presence of arbitrary faults on behalf of the source FCR. In addition, Byzantine resilience requires that if the source FCR is non-faulty, then all non-faulty FCRs reach the validity condition. A complete discussion of the conditions of agreement and validity appears in section 2.2.

To validate a class II exchange implementation in a quadruplicated FTP design, two scenarios must be investigated. The first scenario is a faulty source FCR, and the second is a faulty recipient FCR. Consider first a quad-FTP which uses a majority vote for fault masking. If the source FCR is faulty, it can forward any data value to the recipient FCRs. During the first round of the class II, the corrupted source sends data to the recipients. For the second exchange round, the recipients exchange the values each received from the source during the first round.

Note that the source does not participate in the second round. If the source is allowed to participate in the second round, the source's contribution to the data set to be resolved by the majority vote function carries more weight than does the contribution by the recipient FCRs, allowing the source to corrupt the system. Figure 3-5 demonstrates the source-failed
condition in a conventional quad-FTP in which the source participates in the second round. During the first round, the source could send a different result to each of the recipients. During the second round, the recipients faithfully re-broadcast the value they received from the source. Since the source participates, it can send another set of faulty values to each recipient. Following the second round, none of the data sets held by the recipients contains a clear majority, so validity can not be achieved, although this is not required since the source FCR is failed. However, since each FCR has a different data set which could be resolved differently by the separate instances of the majority vote function, agreement can not be guaranteed.

The theoretically correct source congruency algorithm for a quadruplicated FTP is shown in figure 3-6. The first round is identical to that shown above. The second round is different because the source is prohibited from participating in the second round. In this case, each of the functioning recipient FCRs receives identical data sets, resulting in agreement among the recipients following the majority vote.
The case of a faulty recipient FCR is demonstrated in figure 3-7. If a recipient FCR in a quad-FTP is faulty, that FCR will not necessarily forward the value it received from the source FCR to the other recipient FCRs. However, each recipient FCR will receive two valid copies, one from the source, which is assumed to be functioning, and the other from the third recipient FCR. Since only three values are used by the majority vote function during a class II exchange, two values are sufficient to generate a majority. Since the majority of input values agrees with the copy from the source, the value resulting from the majority vote will also agree with the source, thus ensuring that all functioning FCRs reach the validity condition.

Figure 3-7: Quad-FTP Source Congruency; Scenario 3
One possible implementation of the (4,2) code for the class II exchange is to use the data symbols as direct replacements for data copies in the voted system. The source would generate the four code symbols from the single-source data value, and would send a different symbol to each of the three recipient FCRs. The recipient FCRs would re-broadcast this symbol during the second round. The fourth symbol would be discarded. The problem with this implementation is that it does not allow for faults on behalf of recipient FCRs. Essentially, the symbol position corresponding to the source FCR is guaranteed to be faulty, so the erasure mode is used to recover the symbol. However, the code is not capable of recovering a second symbol error caused by a faulty recipient FCR. This implementation of the class II is not acceptable because validity can not be achieved in the presence of a faulty recipient processor.

Another possible solution is to allow the source FCR to broadcast the fourth symbol during the second round. The problem with this approach is essentially the same problem encountered by allowing the source to participate in the second round in the majority vote implementation. In situations where the source is faulty, the source can generate any data symbols. These data symbols do not necessarily correspond to a valid code word. Since the recipient FCRs are functional, they will faithfully re-broadcast the symbols they received from the source. Therefore, all functioning FCRs will agree on all of the code symbols except the one from the source. If the three code symbols stored by the recipients was guaranteed to map to a valid code word, there would be no problem. However, this can not be guaranteed, since the number of possible three symbol combinations which the source could broadcast is 4096 \(16^3\) and there are only 256 \(16^2\) valid code words. If, during the second round, the faulty source FCR chooses to broadcast a symbol to one FCR which makes its received code word look like a valid code word, and broadcasts to another FCR a symbol which does not complete a code word, the two FCRs will diverge, even though they are both functioning correctly.
A solution to this problem is to provide some mechanism for each FCR to reach agreement on the value of the source FCR's data symbol. Agreement and validity can be assured by the (4,2) code if either of the following conditions are met:

- All functioning FCRs agree on the value of three of the four symbols, and these three symbols map to a valid code word (this condition can be met by the second proposed implementation in cases where a recipient FCR is faulty)

- All functioning FCRs agree on the value of all four symbols (recipient FCRs will agree on the value of the three recipient symbols if the source FCR is faulty in both implementations)

Reaching agreement on the source's symbol is necessary to satisfy the second condition.

The solution chosen for implementation in the FTP/FTSM is to treat the source symbol like a piece of single-source data and exchange it using the traditional class II exchange with a majority vote function to mask faults. The drawbacks to this approach are that two additional rounds of exchange are required (actually, only one additional exchange cycle is required; see section 4.1.2.2) and additional hardware in the form of a voter and holding registers is required. The justification for this decision is based on the fact that the class II can be implemented on the existing data paths with minimal impact on increased hardware and control complexity. Other solutions to the class II problem considered for the FTP/FTSM required more exchange cycles and more additional hardware than the chosen solution.
4. FTP/FTSM Functional Description

This section describes the functions of the FTP/FTSM.

4.1. Data Exchanges

The primary function of the data exchange mechanism is to ensure that each of the functioning FCRs obtains consistent data before acting on the data. The 68030 processor in the FTP/FTSM initiates a data exchange by requesting a data transfer bus cycle with the FTSM. The read/write signal and the function codes\textsuperscript{21} determine the type of exchange to perform as defined in figure 4-1.

<table>
<thead>
<tr>
<th>FC</th>
<th>R/W</th>
<th>Memory Space</th>
<th>Exchange Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>R</td>
<td>User Data</td>
<td>Class I Read</td>
</tr>
<tr>
<td>1</td>
<td>W</td>
<td>User Data</td>
<td>Class I Write</td>
</tr>
<tr>
<td>2</td>
<td>R</td>
<td>User Program</td>
<td>Class I Read</td>
</tr>
<tr>
<td>2</td>
<td>W</td>
<td>User Program</td>
<td>Class I Write</td>
</tr>
<tr>
<td>3</td>
<td>R</td>
<td>Reserved: User Defined</td>
<td>Class II Read</td>
</tr>
<tr>
<td>3</td>
<td>W</td>
<td>Reserved: User Defined</td>
<td>Class II Write</td>
</tr>
<tr>
<td>5</td>
<td>R</td>
<td>Supervisor Data</td>
<td>Class I Read</td>
</tr>
<tr>
<td>5</td>
<td>W</td>
<td>Supervisor Data</td>
<td>Class I Write</td>
</tr>
<tr>
<td>6</td>
<td>R</td>
<td>Supervisor Program</td>
<td>Class I Read</td>
</tr>
<tr>
<td>6</td>
<td>W</td>
<td>Supervisor Program</td>
<td>Class I Write</td>
</tr>
</tbody>
</table>

Figure 4-1: Function Code Table

4.1.1. Class I

Class I exchanges are performed to reach agreement on commonly-sourced output. This situation occurs when the processors perform a calculation on existing data. In conventional fault-tolerant computers, the class I is a voting operation which takes place as a single indivisible cycle. In the FTP/FTSM, the class I is divided into two segments, the write and the read.

4.1.1.1. Class I Write

The first segment of the class I exchange is performed when the output of a calculation is written to the FTSM. Writing data during a class I is very simple. Each processor is expected to have the correct data, so all four processors write the data is written into the FTSM, where the encoder generates the four symbols to be stored in the FTSM RAM. Note that all four
FCRs generate all four symbols. However, each FCR selects only one symbol from the set of four to store in the local memory. This selection is based on the FCR's ID number. Since each FCR has a unique ID, a different symbol is selected by each FCR so that, collectively, all four symbols are stored somewhere in the FTSM.

A schematic of the class I write is shown in figure 4-2. Note that a class I write cycle does not require the use of the data exchange mechanism. Each FCR is expected to have consistent data before the class I is initiated, so nothing can be gained by exchanging the data. This means that the class I write can be performed quickly. Fault masking is performed during the read cycle, discussed below.

<table>
<thead>
<tr>
<th>Channel</th>
<th>Processor</th>
<th>Encoder</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N</td>
<td></td>
<td>N₀</td>
</tr>
<tr>
<td>B</td>
<td>N</td>
<td></td>
<td>N₁</td>
</tr>
<tr>
<td>C</td>
<td>N</td>
<td></td>
<td>N₂</td>
</tr>
<tr>
<td>D</td>
<td>N</td>
<td></td>
<td>N₃</td>
</tr>
</tbody>
</table>

Figure 4-2: Class I Write

4.1.1.2. Class I Read

After a value has been stored into the FTSM using the class I write cycle described above, that data can be read by the processor using the class I read cycle. At this point, each of the four FCRs is expected to have one of the four symbols needed to reconstruct the code word. To reconstruct the code word, each FCR distributes the code symbol stored during the write cycle to the other three FCRs. When the FCRs have received the symbols from the other functioning FCRs, the four symbols are processed by the decoder, where single symbol errors are masked.
A class I read cycle requires one exchange through the data exchange mechanism, as demonstrated in figure 4-3. This exchange imposes a penalty on the response time of the FTSM compared to a zero wait-state system. The data exchange network requires two DATACLK cycles to complete an exchange. Maximum processor skew requires that another cycle be reserved to allow maximally fast FCRs to receive results from maximally slow FCRs. A fourth wait-state is required by the one-stage pipelined decoder. Including the minimum of three cycles required by the 68030, seven processor cycles are required to do a memory read using the class I read cycle. Considering the frequency of memory reads in a computer system, this penalty is the primary source of performance degradation in the FTP/FTSM.

4.1.1.3. Cache Burst

The 68030 supports a special type of read cycle called the cache burst cycle. The internal caches on the 68030 are organized into lines, each of which contains four 32-bit words. Each entry in the cache line can be replaced independently, or the entire line can be filled through the cache burst cycle. The cache burst is intended to increase the cache hit ratio by prefetching objects the processor is expected to use in the future. The cache burst can take advantage of memory systems, especially dynamic RAMs, in which many values from nearby locations can be read very quickly after one value is read.

The FTSM is ideal for supporting the cache burst cycle. Since the exchange and decode stages of the FTSM are pipelined, a steady-state read rate of one 32-bit word every clock cycle can be sustained.
The first longword transferred during the cache burst is located at the address presented on the address lines by the 68030. All of the addresses used during a cache burst are longword aligned, i.e. the lower two bits of the address are zero. The address for subsequent cycles are calculated by incrementing the next two bits (bits 3 and 4) modulo 4. A complete description of the cache burst cycle is described in [22].

Cache burst cycles from the FTSM can be completed in ten cycles. A single longword read cycle takes 7 cycles. Since the cache burst transfers four longwords, the cache burst can theoretically provide a 280% improvement in memory read access time. However, the other three values read during the cache burst must be usable to the processor for this improvement to apply. If the values are not used, the performance will be degraded, since the extra cycles used to complete the cache burst will be wasted.

4.1.2. Class II

A less common situation than the class I occurs when only one FCR has a data object. This situation is usually the result of reading from an input device. Since I/O is unique to each FCR, only one FCR will actually be able to read from a device input register.

Single-source data must be distributed to all FCRs in a fault-tolerant manner. The procedure for distributing single-source data is called a source congruency operation, or class II exchange. The principles of the source congruency have been described in [23]. To tolerate single malicious faults, two rounds are required for a class II exchange.

A class II cycle is initiated by the 68030 when the user-defined function code (FC=3) is asserted. Although a class II could be performed on any memory location in the FTSM, a special region is reserved specifically for class II exchanges. The memory scrubber is programmed to skip this special region, so that locations containing the intermediate results of a class II will not be scrubbed. The reason for skipping these locations will be discussed below.

4.1.2.1. Class II Write

The class II write cycle is demonstrated in figure 4-4. When single source data is written to the FTSM, the FCR distributing the data generates all four symbols from the data word. One of these symbols is stored in the local memory, while the other three symbols are distributed to the other three FCRs. The recipient FCRs take the symbol from the appropriate input and write it into their local memory.
4.1.2.2. Class II Read

One of the constraints imposed on the class II exchange is that two rounds are necessary to guarantee consistency. The FTP/FTSM design imposes another constraint which requires an additional round. This constraint is required by the decoder, which requires four symbols to reconstruct the data word, given that one of the symbols may be erroneous. In a traditional voting scheme, only three copies are required to determine an unambiguous majority in the presence of a single error. Since the decoder requires four symbols, the source's symbol must be included in the class II exchange. A simple solution is to allow the source FCR to broadcast its symbol to the other FCRs during the second round. However, this procedure would cause data from the source processor to carry extra weight during the fault masking operation compared to the other processors. If the source FCR is maliciously failed, it could broadcast symbols to the other (functioning) FCRs which would cause them to reach different conclusions regarding the validity of the data. This indecision can cause divergence, and therefore catastrophic loss of the system.

The solution is to redistribute the source symbol among recipient processors and use a vote function to ensure data consistency, as shown in figure 4-5. The first round shown distributes the symbol $N_0$ from the source FCR to all recipient FCRs. During the second round, the recipient FCRs exchange and vote the value of $N_0$, thus ensuring that all functioning FCRs reach agreement on the value of $N_0$. Essentially, these two exchanges are a traditional source congruency operation interleaved with the FTSM class II exchange.
During the third round shown in figure 4-5, the symbols stored by the recipient FCRs are exchanged. These symbols originated from the source FCR, so validity is not assured. Agreement is assured because if the source FCR is faulty, all of the recipient FCRs are assumed to be non-faulty and will therefore forward consistent copies of their respective symbols. At the end of the third round, each functioning FCR has a consistent value for each of the four code symbols, so they are all able to reach agreement.

Note that the first and third exchange rounds use mutually exclusive data paths. Since the third round is not dependent upon the results of the first round, these two rounds can actually be performed simultaneously.

4.1.3. Memory Scrubbing

The memory scrubber reduces the probability of transient bit errors from accumulating in memory locations, causing uncorrectable situations. The scrubber cycles through every memory location in the FTSM, except for a special region defined for class II accesses. Since the scrubber uses class I read and write cycles, locations written with a class II write cycle must not be scrubbed. The scrubber reads data from a memory location, exchanges it through the data exchange network, and re-writes it to memory. During the process, the encoder/decoder pair corrects any correctable errors which were present when the value was read out of memory.
The scrub cycle operates on four locations in an indivisible operation. The scrub cycle as defined in appendix 9.2 takes eleven cycles to complete. Because the FTSM exchange and decode mechanism is pipelined, the first six cycles are appended to the end of a processor read cycle (either class I read or cache burst). These cycles are already required to complete the read cycle, so they are essentially invisible to the processor. The remaining five cycles extend past the end of the previous read cycle and represent cycles the processor might want to use for another bus access. To prevent the processor from colliding with the end of a scrub cycle, the scrubbing controller asserts the WAIT signal on the S-BUS. The processor must idle until WAIT is deasserted before starting a new S-BUS cycle.

The frequency of memory scrub operations is regulated by the scrub regulator. The scrub regulator keeps the scrubber from performing a scrub until a specific number of FTC cycles have elapsed since the last scrub cycle. In this way, the scrubber is prevented from interfering significantly with processor accesses to the FTSM.

The scrubber address generator is selected by the address source multiplexer in the memory controller and the scrub address is loaded into the address register. Simultaneously, the scrub address generator is incremented to the next address to be scrubbed. Next, the code symbols are read out of memory and broadcast to each FCR. This process is performed four times, with the address register incremented to the next long word after each broadcast. Shortly after the fourth location to be scrubbed is read out of memory, the first decoded value appears at the end of the read pipeline.

For the next four clock cycles, the scrub values are written back to memory. The address register is incremented to loop back to point to the first longword in the scrub cycle.

4.1.4. Command Exchanges

The purpose of the command exchange is to provide a mechanism for initiating certain hardware exceptions from software. These hardware exceptions are intended for use during initial synchronization and reintegration.

The first processor clock cycle after the falling edge of FTC is reserved for command exchanges. The source of commands is the command register. If the processor writes to the command register, the contents will be exchanged during the next command exchange period. If the contents of the command register are unchanged since the last exchange, no exchange is performed and instead, a null command message is sent.
After the message exchange, the commands are voted by a special command voter. The voter generates two signals, MAJ and UNAN, which indicate majority and unanimity, respectively. The definition of majority and unanimity is dependent on the number of FCRs enabled by the vote mask field in the command register. These definitions are as follows:

<table>
<thead>
<tr>
<th># FCRs enabled</th>
<th>Majority</th>
<th>Unanimity</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The command executor uses these two signals and a watchdog timer to determine if a command is to be executed. The watchdog timer is enabled by a bit in the command register. If the watchdog timer is enabled, the command executor will start the timer as soon as a majority is observed. The command will not be executed until either unanimity is observed or the watchdog timer expires. Note that this is not a Byzantine resilient function since it requires all FCRs to receive commands on the same FTC cycle in order to start the watchdog timer simultaneously. However, a Byzantine resilient initial synchronization routine is difficult since, before initial synchronization, the processors violate one of the requirements for Byzantine resilience.

If the watchdog timer is disabled by the bit in the command register, the command executor will validate the command as soon as a majority is observed.

4.2. Synchronization

One of the requirements for Byzantine resilience is that the FCRs must be synchronized to within a known skew. There are two primary synchronization functions to be performed: initial synchronization and steady-state synchronization. The former is provided by the command exchange mechanism (CEM) and the fault-tolerant clock (FTC) while the latter is performed strictly by the FTC.

The type of synchronization to use in a fault-tolerant computer is a widely debated topic. There are two fundamental types of synchronization: micro-synchronization and functional synchronization.

Micro-synchronization implies that all processors belonging to a redundant group are executing identical code at the exact same time, within tolerances on the order of fractions of a processor clock cycle. The implication of this type of synchronization is that each of the
processors must execute the same code in lock-step. To achieve this requirement, the
processing elements must be clock deterministic. Every task the processors initiate must
complete within a known time, and the processors must wait that maximum time before
responding to the results. Usually, the engineer must design a unique processing element,
since commercially available processing boards are not guaranteed to be clock deterministic.
Also, the design of unique I/O devices which respond in a clock deterministic fashion is
required since asynchronous buses, such as VMEbus, are inappropriate for clock
deterministic processors.

The alternative to micro-synchronization is functional synchronization. Used on the FTPP\textsuperscript{24}
and the NEFTP\textsuperscript{25}, functional synchronization does not require absolute clock deterministic
processors. Instead, synchronizing acts are performed explicitly by the processor.

Functional synchronization is an ideal way to build a fault-tolerant computer\textsuperscript{26}. In the SIFT
computer functional synchronization was performed by a software task, resulting in a loosely
synchronized design with a considerable performance penalty\textsuperscript{27}. This problem was solved in
the network element designs (the FTPP and NEFTP) by performing tight synchronization
through the network elements. Since each processor is connected to a network element, the
processors have a tightly synchronized temporal reference readily available. By using the
network element approach to synchronization, the advantages of functional synchronization,
including the ability to use non-deterministic processors and to employ hardware and
software diversity can be realized with minimal penalty.

Unfortunately, functional synchronization is not appropriate for the FTP/FTSM. The
message passing operations of the FTP/FTSM have been implemented at a level of finer
temporal granularity than on most other FTP designs, including the NEFTP. Maximum FCR
skew directly affects the FTSM access time. Each FCR must wait twice the maximum skew
to allow a maximally fast FCR to receive information from a maximally slow FCR.
Synchronization skew must be minimized to prevent severe performance degradation in the
FTP/FTSM. The network elements in the NEFTP have a post-synchronization skew of 375
ns\textsuperscript{28}. Since each processor must wait for a period of twice the maximum skew, memory
accesses would be delayed by at least 750 ns with a functionally synchronized design similar
to the NEFTP, corresponding to almost 20 wait states for a 25 MHz 68030. This is clearly an
unacceptable amount of performance degradation. Consequently, micro-instruction
synchronization was chosen for the FTP/FTSM.
4.2.1. Steady-State Synchronization

The fault-tolerant clock, or FTC is responsible for generating a consistent timing references for all functioning FCRs.

The FTC cycle is shown in figure 4-6. The majority of the FTC cycle is the data phase, during which data can be exchanged by the FTSM elements in each FCR. During the low part of the FTC, called the command phase, no data can be exchanged. Since the inability to exchange data inhibits the performance of the computer, the command phase is minimized.

![Figure 4-6:Fault-Tolerant Clock Cycle](image)

The command phase is always two processor clock cycles long and consists of two parts: the command exchange period and the adjustment period. The command exchange, which occurs on the first processor clock (PROCCLK) cycle after FTC goes low, is the exchange and vote of the command register described in section 5.8.2.2. If a command is validated, that command will be executed by the command executor.

The second part of the command phase, the adjustment period, is from two to four GENCLK cycles long (GENCLK is an unadjusted 50 MHz reference). The FTC circuitry compares the relative positioning of the local FTC to the median of the remote FTCs to determine the amount of adjustment needed. Normally, an adjustment is made by stretching the FTC for one GENCLK cycle. However, if the local FTC is observed to be ahead or behind the majority FTC, the local FTC is adjusted by stretching it 2 or 0 GENCLK cycles, respectively. Since the adjustment period must contain exactly two PROCCLK cycles, the second PROCCLK cycle in the adjustment period is also subject to adjustment by the FTC mechanism. An example of the three possible FTC adjustment cycles is shown in figure 4-7. A more detailed description of each FTC adjustment cycle appears in appendix 9.5.
A message is broadcast by the IFC on every PROCCLK cycle, regardless of whether the processor requests one or not. If the processor requests a message, a valid message is sent. If no message exchange is requested, a null message is sent. The type of message sent depends on the state of the local FTC. During the data phase, data messages are sent. Commands are broadcast during the two-cycle command phase. The type of message received can therefore be used to determine the state of the FTC on a remote FCR.

An edge-voter circuit is used to determine a majority FTC signal, called THEIRFTC, from the three incoming FTC signals. The median (or second) incoming FTC edge is used to indicate the majority. The majority FTC is then compared with the local FTC and PHASE signals to determine if the local FCR is synchronized with the majority of the external FCRs.

A phase detector determines what type of adjustment, if any, should be made to the local fault-tolerant clock. The phase detector uses a delayed version of MYFTC, called MYFTC-dly, for comparison with THEIRFTC. MYFTC is delayed to match the expected delays of the inter-FCR communications network. The adjustment decision is made based on the alignment of MYFTC-dly with THEIRFTC and on the value of PHASE-dly. The signal
PHASE (or PHASE-dly) is used so that all FCRs will converge toward synchronization (i.e. they will never all see themselves behind). If all FTCs are at most one GENCLK cycle out of sync (as they should be during steady-state operation), convergence not difficult without the use of the PHASE signal. If, however, the FTCs are far out of sync (which is possible after system power-up), guaranteed convergence is not possible without using PHASE.

4.2.2. Initial Synchronization

Initial synchronization is the synchronization of each FCR to a common fault-tolerant reference after system reset or power-up. The synchronization is performed in two steps: FTC synchronization and micro synchronization.

The FTC is a free-running phase-locked circuit so FTC synchronization is performed without stimulus. FTC synchronization is completed in a relatively short amount of time. If two FCRs are 180° out of sync, and each FCR makes a one-GENCLK adjustment toward synchronization on every FTC cycle, the two FCRs should come to synchronization within approximately:

\[
\frac{1}{4} \left( \frac{\tau_{fcl}}{\tau_{fc}} \right)^2 \frac{1}{\tau_{adj}}
\]

where \( \tau_{fcl} \) is the nominal period of the FTC and \( \tau_{adj} \) is the magnitude of the maximum adjustment made to the FTC during each adjustment period. For the design presented in this paper, \( \tau_{fcl} \) is approximately 5 \( \mu \)s and \( \tau_{adj} \) is 20 ns, so the FTC synchronization time is about 310 \( \mu \)s.

Micro synchronization is the synchronization of the processors' internal micro-instruction controllers. Because micro synchronization involves the processors themselves, the actual implementation of micro synchronization is processor dependent.

The standard procedure for implementing micro synchronization is the use of a conditional loop which is one micro-instruction long. With this type of loop, we know that if all functioning FCRs have entered the loop by some point, then we know that they are all executing the same micro-instruction. By setting the loop condition at some known synchronized reference point(such as the FTC, which is synchronized before the processors), all processors will exit the loop at the same time and will then be micro-instruction synchronized.
Micro synchronization on the FTP/FTSM is performed by the 68030 using the STOP instruction\textsuperscript{29}. First, the FTCs are allowed to synchronize during initial startup procedures. Following initial startup, a processor writes a command for voted interrupt to the command register, and immediately executes the STOP instruction. The request for voted interrupt is exchanged and voted during the command phase of the FTC. When either all four FCRs have received the voted interrupt command, or a majority (3) have receive the command and a known timeout has elapsed, an interrupt is performed at a specific point during the FTC cycle.
5. FTP/FTSM Architecture

This section describes the architecture of the FTP/FTSM. Figure 5-1 shows an overview of the architecture of a single FCR. Each of the sub-systems will be discussed below.

![Diagram of FTP/FTSM FCR Architecture]

**Figure 5-1: FTP/FTSM FCR Architecture**

5.1. Fault-Tolerant Shared Memory

The fault-tolerant shared memory, or FTSM, is the heart of the FTP/FTSM system. This system is designed to act as a single memory unit to which all processors have access. The FTSM also provides fault-tolerance. In the presence of all single faults and some double faults, the FTSM will still deliver the correct value to all functioning processors.

The FTSM is composed of four main units, the encoder/decoder, RAM memory modules, an address decoder/register, and an inter-FCR communication system. Access to the FTSM from the synchronous bus is controlled by a suite of finite-state machine controllers. The data paths of the FTSM are shown in figure 5-2.
Figure 5-2: FTSM Data Paths
The address register decodes the addresses from the processor and controls the RAM memory modules. The address register also contains the scrubbing address generator.

The encoder generates code symbols during processor write cycles and during the writeback phase of scrubbing exchanges. The decoder regenerates data values and corrects errors during read accesses and scrubbing exchanges. The exchange mechanism and decoding logic is pipelined to allow the full memory access bandwidth to be realized in some situations.

The RAM modules store the data symbols.

The inter-FCR communication system exchanges code symbols for regenerating data words, special commands requested by the processor, and temporal information for fault-tolerant synchronization.

The data bus width in and out of the FTSM is 32 bits wide, allowing a maximum transfer of one 32-bit word in a single cycle. However, the data paths and coding scheme are designed to operate on the individual bytes of a longword independently. The result is that single byte, 16-bit word, and three byte oriented operations can be performed. These modes are important for supporting all of the potential bus cycles requested by the 68030.

5.1.1. Memory Controller/Address Register

The device which controls the memory modules in the FTSM is the memory controller/address register (hereafter referred to as the address register). This device contains an address register, address decoding logic, the scrubbing address source, and the address pipeline. A block diagram of the address register is shown in figure 5-3.
Figure 5-3: Memory Controller/Address Register

The address register latches the address presented by the processor or the scrubber so that the memory modules are guaranteed a stable address during the memory access cycle. The upper bits of the address (A19-A4) are direct latches. Bits (A3-A2) are latched by a special device which can increment these two bits modulo 4. This incrementable register is used during cache burst accesses and memory scrub cycles, when four adjacent 32-bit words are read from the memory on subsequent processor clock cycles. The lowest bits, (A1-A0), are used in conjunction with the SIZE and R/W signals to determine which byte(s) of the 32-bit word are being referenced. The logic for decoding the byte references is shown in appendix 9.4.

The scrubbing address generator is a counter which is programmed to scan through every address in the FTSM. The counter is also designed to skip the region(s) defined for class II exchanges, since these locations should not be scrubbed. The scrubbing source is used as the address source when the following conditions are met:

- The processor is completing a class I read (either single location or cache burst)
- The scrubbing regulator has approved the scrubbing operation

The scrubbing regulator prevents the scrubber from interfering significantly with the throughput of the processor. The regulator counts FTC cycles following a scrub cycle, approving a scrub cycle only after a specific number of FTC cycles have occurred. In this
way, the frequency of the scrub operation can be regulated, with the actual frequency adjustable in a trade-off between transient system reliability and throughput penalty.

The address pipeline contains the addresses of the data objects which are passing through the class I read pipeline. The pipelines are of equal length, and are designed so that the address at the end of the address pipeline corresponds to the data object at the end of the read pipeline. If the syndrome of the data object is non-zero, meaning that the decoder detected errors in the received code word, the syndrome is recorded by the syndrome register and the address at the end of the address pipeline is recorded by the syndrome address register. The contents of these registers can be used by fault-detection and isolation software (FDI) to determine if the system contains any faults. Since different FCRs may see different syndromes, the contents of the syndrome registers must be exchanged using the source congruency operation.

5.1.2. Encoder/Decoder

The encoder generates the code symbols for data words to be stored in the FTSM. The decoder is responsible for correcting any errors which occur in the code words read from the FTSM.

The encoder multiplies a 2-symbol vector (the data word) by a 2x4 array to generate a 4-symbol vector (the code word). Each symbol in the array and the vectors is an element of the Galois Field of $2^4$ elements. The encoder is byte-sliced, i.e. each byte in a 32-bit longword is operated on by a separate unit, thus providing the capability for performing write operations of any size requested by the 68030.

Fault masking is performed by the decoder. The decoder is composed of several parts. The first part, the parity check matrix multiplier, is used to generate the syndrome from the received code vector, or R vector. The received vector consists of the three symbols received from the remote FCRs plus the local symbol. The generation of the syndrome is performed by multiplying the R vector by a parity check matrix. The mechanics of this operation are essentially the same as that performed by the encoder, except that the input vector is a 4-symbol vector and the result is a 2-symbol vector.

The 2-symbol vector known as the syndrome is then used by the error vector generator to determine what error pattern, if any, is present in the R vector. Each error pattern which meets the constraints listed in section 3.3 will result in a unique syndrome. The error vector generator performs the equivalent of a table look-up to map a syndrome to its corresponding
error vector. If a syndrome is presented to the error vector generator which does not map to a valid error vector, an error vector of zero will be returned.

After determination of the error vector, fault masking is performed by adding the error vector to the R vector. The result will be the C vector, provided that any errors which occurred meet the constraints for correctable error patterns. Note that only the two symbols in the R vector which correspond to data symbols need to be corrected. If the error pattern is limited to the check symbols, no correction is necessary since these symbols will be discarded anyway.

The decoder is fed by the data receiver unit. This unit obtains data symbols received by the IFC. When the IFC receives a data symbol, the symbol is stored in the FIFO registers. The FIFOs allow some skew between FCRs, since the input are clocked by the IFC mechanism, not by the local processor clock.

Most of the extra hardware in the data receiver unit is used during the source congruency, or class II, exchange. The source congruency requires one round of exchange during the write cycle and two rounds of exchange during the read cycle. The round during the write cycle distributes symbols generated by a data item known only to the source processor. The second round during the read cycle is necessary to make sure that the symbol originating from the source processor is consistent among functioning FCRs.

During the class II write, the source processor generates all four symbols to be stored in the FTSM. One symbol is selected to be stored locally. The other three are transmitted to the other FCRs. Each FCR receives a different symbol. The source FCR uses the 2-input multiplexers feeding the IFC transmitters to select the symbols from the encoder during the class II write. The recipient FCRs receive the symbol they are to store in the receiving FIFO associated with the source FCR. Note that all receiving FIFOs will be loaded with something, but only one will actually have a valid symbol. A multiplexer selects one of the three incoming symbols from the FIFOs to store in the SRAM bank.

During the first round of the class II read, symbols are broadcast by all FCRs (including the source). These symbols are stored in the class II delay register (C2DR) and removed from the FIFOs. For the second round of the class II read, (the third round of the class II exchange), the recipient FCRs select the source symbol using a multiplexer (before deletion from the FIFOs) and broadcast to all other FCRs, including the source. All FCRs receive three symbols which represent the source symbol as broadcast by each recipient FCR. These three symbols are voted, and the result of the majority vote is used to replace the source's
symbol from the C2DR. Finally, with a consistent view of the code word, the FCRs use the decoder to regenerate the data word.

5.1.3. RAM Modules

The storage unit for the FTSM is composed of static memory devices. The smallest data object handled by the FTSM is a byte, which is encoded into four 4-bit symbols. Each FCR must therefore be capable of independent addressing in 4-bit increments. The natural device for this purpose might at first seem to be 4-bit wide static memories currently available from a number of manufacturers. However, to take advantage of the double-bit failure protection afforded by the chosen coding scheme, it is necessary to create a pseudo 4-bit wide device by bit-slicing four 1-bit wide parts. This way, the failure of one memory device will only cause the loss of one bit, instead of the loss of a complete symbol.

Several manufacturers, including Cypress and Integrated Device Technology, make memory modules composed of a number of 1-bit wide parts. The IDT7M4016 from IDT is a 4 megabit hybrid module constructed from 16 256K by 1-bit devices that can emulate a 1M x 4 device with an anticipated access times of 25 ns.

5.1.4. Inter-FCR Communication System

The Inter-FCR communication system, or IFC, connects each FCR to every other FCR, satisfying the connectivity requirement for Byzantine resilience. The IFC also determines a temporal reference for each external FCR. This reference is used by the FTC circuitry to maintain inter-FCR synchronization.

An IFC message is broadcast to external FCRs on every PROCCLK cycle. IFC messages originate from several sources, including the FTSM, the command register, and from within the IFC itself. The local FTC determines what type of message can be sent. When the FTC is high, data messages are sent. Command messages are sent on the low part of the FTC. If the FTSM requests an exchange, the IFC will broadcast a valid data message, which contains the four symbols for the four bytes containing the data requested from the FTSM. If no data exchange is requested, the IFC broadcasts the null data message. Likewise, when the FTC goes low, the IFC will broadcast a command message if the command register contains a new command request from the processor. The null command will be sent otherwise. A null command is always broadcast on the second processor clock after FTC goes low.

59
IFC messages are 20 bits in length. The 4 MSBs indicate the message type. The 16 LSBs are defined by the message type. The defined message types are shown in figure 5-4.

![Figure 5-4: IFC Messages](image)

All other message types are undefined and reserved for future use.

Fault-tolerant clock signals for the external FCRs are derived from the message type received. A data message indicates that the FTC for the FCR from which the message came is high. A command message indicates that the FTC is low. The IFC generates a remote FTC signal for each FCR and sends it to the FTC hardware.

The IFC systems are interconnected by a high bandwidth, low latency, highly isolated communication link. Bandwidth is important to to support the high data-exchange rate between FCRs. To support a longword access on every cycle of a 25 MHz clock, one 20-bit word must be exchanged by the IFC every 40 ns. The resulting bandwidth requirement is 500 Mbits/sec. The latency of the IFC link must be minimized to prevent severe performance degradation. The design presented in this thesis presumes an IFC link with a latency of less than 80 ns, or two processor clock cycles. Isolation is necessary to prevent the electrical, magnetic, or mechanical propagation of faults from one FCR to another.

Three possibilities for the implementation of the IFC link are available. The first is the use of parallel wires. The second is a high-speed fiber-optic or coaxial serial link. The third
alternative is a hybrid system, utilizing multiple serial links in parallel in a 16 to 2, 16 to 4, or 16 to 8 configuration.

Parallel interconnect is the simplest solution from the standpoint of controller complexity. A parallel interface would need only a strobe signal to indicate the transmission of data. Parallel interconnect has been used on several FTPs, including the VLSI-FTP and the FTPP. The problems with parallel interconnects include large connector size, the need for extensive isolation circuitry (one for each incoming wire), and bulky cables.

Serial interconnect has several advantages over parallel. Connector size is greatly reduced. The wire bulk is reduced since only one wire is needed for each incoming port. Also, only one isolation circuit is needed for each incoming port, saving valuable board real-estate. However, the bandwidth of the individual wires must be greater, since a single cable must carry the same amount of information as 20 wires in the parallel solution. Also, the protocol for serial transmission is more complicated, since the data transfer control information is interleaved with the data.

A serial link can be implemented with either coaxial cable or fiber-optics. Coaxial cable has a useful limit of about 1 GHz over short distances, suitable for 50 MHz parallel data input from the FTSM. Fiber-optics can be used at bandwidths at least as high as coaxial cable, provide excellent isolation, and are not susceptible to EMI noise.

The third possibility, the hybrid serial-parallel system, was not considered for the FTP/FTSM. The primary benefit from the hybrid system is increased link bandwidth at the cost of increased wires, connector size, and board space consumption. Serial links using currently available technology have enough bandwidth for the desired application, so there is no clear advantage to using this type of system.

Using the criteria discussed above, the ideal link for the IFC system would be a fiber-optic link with a maximum bandwidth of 500 Mbps and a latency of less than 80 ns. Currently, Advanced Micro Devices offers a pair of devices called the TAXI chip set. The TAXI set performs serialization, clock recovery, and data recovery for data transmission rates of 125 Mbps over either coaxial cable or fiber optics. The TAXI interface is an 8-bit wide parallel port. The latency of the TAXI set is approximately 360 ns. Clearly, the TAXI chip set falls far short of the ideal system and is not suitable for use in the FTP/FTSM. However, with the advent of gallium-arsenide, serial communication chip sets with a bandwidth of 1 GHz can be expected in the near future.
5.1.5. FTSM Controller

The fault-tolerant shared memory is controlled by two finite-state machine (FSM) controllers. A complete description of the controller appears in appendix 9.3. These controllers regulate the flow of data between the synchronous bus and the FTSM. FSM A is responsible for controlling the flow of information through the encoders, in and out of the RAM devices, and into the transmitter side of the IFC. FSM A also controls the address register. FSM B controls data flow from the outputs of the IFC, through the decoders, and onto the synchronous bus. Also included are the scrubbing regulator, which controls the frequency of scrub operations, and the IFC transmission controller, which determines what type of message to send to other FCRs through the IFC.

5.2. Fault-Tolerant Clock

The fault-tolerant clock circuitry synchronizes the fault-containment regions. The FTC circuitry generates all timing references for the FTP/FTSM.

The FTP/FTSM is driven by three clock signals, GENCLK, PROCCLK, and FTC. GENCLK is a 50 MHZ clock derived from a crystal oscillator. PROCCLK is nominally a 25 MHz signal generated by the fault-tolerant clock circuit which is used by most of the system, including the processor, for a clock reference. FTC is a long period signal, nominally 4.98 μsec, or 249 GENCLK cycles, used for locking the four FCRs into synchronization and for regulating the data exchange mechanism.

PROCCLK is used as the clock source for all components of the FTP/FTSM except the FTC, and the multi-function peripheral (MFP). PROCCLK cycles are adjusted during the adjustment period of the FTC to keep the separate FCRs in synchronization. An adjustment is made by stretching FTC and PROCCLK for one GENCLK cycle. An adjustment of one GENCLK cycle is normally made once every FTC cycle. However, if the local FTC is observed to be ahead or behind the majority FTC, the local FTC and PROCCLK are adjusted by stretching them 2 or 0 GENCLK cycles, respectively, to bring the local FTC back into synchronization with the other FTCs. The FTC circuitry must ensure that an equal number of PROCCLK cycles (124) appear in one FTC cycle to guarantee clock-deterministic processor behavior.

Data exchanges, either valid or null, occur every PROCCLK cycle except during the command phase of the FTC. Command exchanges occur during the command phase. Valid
command exchanges occur only on the first PROCCLK cycle of the command phase. A null command is always exchanged on the second PROCCLK cycle during the command phase.

The local FTC is used as a timing source for the execution of processor requested interrupts (currently the only implemented command). The FTC is the only clock signal which, during initial synchronization, provides a consistent timing reference to all four FCRs. The FTC is also used by the MFP to generate periodic interrupts and by the scrub regulator to moderate the occurrence of scrub cycles.

A block diagram of the FTC is shown in figure 5-5. The design is similar to that of the NEFTP FTC, with a fixed block size.

![FTC Block Diagram](image)

**Figure 5-5: FTC Block Diagram**

The FTC for a remote FCR can be derived from the IFC corresponding to that FCR. A data message means FTC was high when the message was sent, and a command message means the FTC was low. The approximate delay between message transmission and message reception is known, so an accurate view of the FTC for each FCR can be determined. The remote FTCs are voted to determine the majority fault-tolerant clock and compared to the local fault-tolerant clock. This comparison determines whether any adjustment to the local FTC is necessary.

The edge voter detects a majority of rising or falling edges on the input signals. Note that only one type of edge is detected at any given time. That is, if the detector is in the low state, only rising edges will be detected and voted. Any falling edges will be ignored until a majority of rising edges are observed.
The results of the edge voter is the median external FTC, which is referred to as THEIRFTC. THEIRFTC is sampled on falling edges of GENCLK. This sampled value is synchronized with the rest of the FTC system by another register stage clocked on the rising edge of GENCLK. Since the local FTC is synchronized to rising edges of GENCLK, the half-cycle samples of THEIRFTC will indicate the exact time of the median FTC transition, within one-half of a GENCLK cycle. The sampling and synchronization of THEIRFTC is demonstrated in figure 5-6.

![Diagram](image)

**Figure 5-6: Synchronization of THEIRFTC**

The FTC generator is a finite-state machine which checks the signals BEHIND and AHEAD from the phase detector during the FTC adjustment period. If neither of these signals is asserted, the local FTC is assumed to be synchronized to the majority FTC, and no adjustment is necessary. If either signal is asserted, an appropriate adjustment is made to the local FTC. A complete description of the FTC generator is located in appendix 9.5.

The programmable delay between the FTC generator and the phase detector is necessary to delay the local FTC to match the expected delay imposed on the remote FTC signals by the IFC mechanism.

The phase detector compares the delayed local FTC to the majority FTC generated by the edge voter. If the rising and falling edges of the local FTC and the majority of the remote FTCs match exactly (±1/2 GENCLK cycle), the signals BEHIND and AHEAD are not asserted. If a mismatch is observed, the PHASE signal is used to determine if the next FTC cycle should be shortened or lengthened.
5.3. Command Exchange Mechanism

The command exchange mechanism (CEM), shown in figure 5-7, is used to synchronize normally asynchronous hardware exceptions. All of the currently defined exceptions are invoked from software. An implementation for synchronizing hardware invoked exceptions could be developed, but it would require a source congruency operation. So, to simplify the design, only software invoked exceptions were included.

![Command Exchange Mechanism Diagram](image)

**Figure 5-7: Command Exchange Mechanism**

The command message to be exchanged is written to the command register by the processor. When the exchange controller detects a new command in the command register, it signals the exchange mechanism to transmit the command during the next command phase of the FTC. The format of the command register is defined in section 5.8.2.2.

When a command is received by an FCR, the command is loaded into the associated receive command registers. There is one register for each external FCR. These three registers, plus the local value from the command register are fed into a 4-way voter. The voter, using the vote masks as defined in section 5.8.2.2, will determine when either majority or unanimity is obtained.

The command executor actually executes the command. The input from the command executor comes from the voter, providing a fault-tolerant command to execute. The command
executor uses the MAJOR and UNAN signals, which indicate majority and unanimity, respectively, to determine whether to execute the command. The command executor is also connected to a watchdog timer. The watchdog timer is clocked by the FTC, since FTC is a synchronous reference for all FCRs. The command executor starts the watchdog timer by asserting the START signal. The timer signals the command executor through the DONE signal that the timeout period has elapsed.

5.4. Synchronous I/O Bus

One aspect of the FTP/FTSM which is very different from a simplex computer is the requirement for clock-deterministic behavior by the processor. Each FCR in the FTP/FTSM is micro-instruction synchronized, which means that each processor must be executing the same micro-instruction at the same time, within a small tolerance. This synchronization scheme, which is necessary in the FTP/FTSM to prevent severe performance degradation, puts a special constraint on I/O devices. Since each processor must do exactly the same thing, any device connected directly to the FTP/FTSM must respond using a synchronous, clock-deterministic protocol. This precludes the use of asynchronous buses, such as VMEbus or MultiBus, under the direct control of the processor.

A synchronous bus, or S-BUS, is defined for the FTP/FTSM which allows expansion using clock deterministic components. Also, it allows additional FTSM systems to be connected to one processor, thus expanding the memory of the system. The bus definition is very simple, with a provision for a single bus master and multiple bus slaves. Bus arbitration to support multiple bus masters is not currently defined.

The bus is defined to reside on a 64 line backplane. Backplanes of this type are available from several manufacturers for the Motorola VSB (VME Subsystem Bus) which are based on a 96-pin, 3 column Eurocard type connector. Only the outer two columns of the connector are used by the VSB backplanes. The inner column is reserved for the 32 bit extension to the VMEbus. Note that the S-BUS definition is not the same as that for the VSB bus.

The address (24 bits) and data (32 bits) are multiplexed on the same lines. All 32 lines are defined as multiplexed address and data lines, even though the address in the FTP/FTSM is only 24 bits wide. This allows easy extension of the address space if necessary. In the current design, the bus master is not required to drive A31-A24, and bus slaves are not required to decode them.
The bus has several control signals for data transfer. A description of each signal is given below. Examples of bus cycles using these control signals are presented in appendix 9.2. Also included in appendix 9.2 are tentative AC electrical specifications for each signal.

The /ALERT signal, asserted low, is driven by the bus master to signal the bus slaves that a bus cycle might be started. The address is presented by the bus master on the address/data lines during the assertion of /ALERT. The address should be latched by the bus slave on the first rising edge after /ALERT is asserted. /ALERT is only asserted for one clock cycle. /ALERT can be derived from the /ECS signal on the 68030. /ECS, or external cycle start, indicates that a memory reference is requested by the CPU. However, if the reference is a read cycle and the requested item resides in the processor cache, the external cycle is not needed. The /ALERT signal allows bus slaves to prepare for a possible bus cycle, thus decreasing the memory access time.

The external bus cycle is qualified by the /BCQUAL signal. If /BCQUAL is asserted on the clock cycle immediately following the assertion of /ALERT, the external cycle is necessary and the addressed bus slave should continue the fetch cycle. If /BCQUAL is not observed after an /ALERT, the bus cycle is cancelled. If the bus cycle is qualified and the cycle is a memory write (/WRITE is asserted) the data value will be presented on the address/data lines by the bus master when /BCQUAL is asserted. If the cycle is a memory read, the bus master will release the address/data lines to a high-impedance state so that the bus slave can drive the lines with valid data.

The /WRITE signal, asserted low, signals the bus slave that the data value is to be written to memory. If the /WRITE signal is high, a memory fetch is to be performed.

The /STERM signal is the synchronous termination signal used by the bus slave to signal the bus master that the bus cycle is complete. If the cycle is a write access, the bus master will remove the data from the data lines and will deassert /BCQUAL on the rising edge following the observation of /STERM. If the cycle is a read access, the bus slave should drive the data onto the data lines during the clock cycle following the assertion of /STERM. The bus master will latch the data on the rising edge following the observation of /STERM.

The SIZE signals indicate the amount of data being transferred between bus master and bus slave. The SIZE signals are defined as follows:
00 - longword (32 bit word)  
01 - byte  
10 - word (16 bit word)  
11 - 3 bytes

The SIZE signals can be ignored on read cycles. All devices connected to the S-BUS must be 32-bit wide data ports. A slave should drive all of the data lines, even if the bus master is only expecting data on some of them. However, during write cycles, the SIZE bits and the lower two bits of the address must be used to determine which bytes should be modified in memory. The S-BUS supports all of the 68030 transaction sizes including byte, 16-bit word, 3-byte, and 32-bit longword transfers.²⁰

Two bits on the bus are reserved for defining the address space being referenced. The defined address spaces are as follows:

00 - Standard (Class I) space  
01 - Source congruency (Class II) space  
10 - System space (interrupt vectors, etc.)  
11 - Unused, reserved

The standard space is the region used for most memory accesses. Any of the standard function codes generated by the 68030 (supervisor or user, program or data) are translated to the standard space. CPU space is reserved for future additions, such as interrupt vector fetch cycles. The class II space is used to request a class II cycle instead of a class I. Class II space is requested by the processor's assertion of the user reserved function code (FC=3).

5.5. I/O Processor

It is only practical to connect a particular I/O device to one FCR of the FTP/FTSM. The memory map of the FTP/FTSM defines five separate I/O regions. Each FCR has its own I/O region, with a fifth region defined and reserved for future use. When the processors access a memory location in the I/O space defined for FCR n, only processor n actually reads or writes that location. All processors receive the acknowledge signal from the bus slave at exactly the same time, even though processor n is the only processor actually performing a data transfer cycle. Including all processors in the bus cycle ensures clock-deterministic behavior during I/O transfers.

To simplify the task of connecting I/O devices to the FTP/FTSM, an I/O processor, or IOP, is used to control the I/O devices. The IOP communicates directly with the I/O devices under the command of the FTP/FTSM. The FTP/FTSM processors, also known as computational
processors (CP), communicate with their respective IOPs through a dual-port memory mapped to the I/O regions. The dual-port memory provides a synchronizing interface between the two processors. The IOPs are not required to be clock synchronized. In fact, each IOP could be running a unique program. The IOPs can be connected to a standard asynchronous bus, such as the VMEbus, for direct communication with standard I/O devices.

The CP requests information from the IOP by writing a command to the dual-port memory. The IOP either polls the dual-port memory for a command, or is interrupted when the CP writes to the command location. When the IOP observes a new command, it performs the operation requested. If the command is an output command, the IOP takes data from a specified place in dual-port memory and writes it to the desired output device. If the command is an input, the IOP fetches the value from the requested input device and copies it to dual-port memory. After completing the requested operation, the IOP writes an acknowledge message to the dual-port memory. The CP polls the acknowledge location until it recognizes that the IOP completed the requested task. Polling is required for input requests but is optional for output requests. After receiving the acknowledge for an input request, the CP can read the input data from the dual-port memory. Note that the data and poll results must be exchanged using the class II operation before being used in a computation.

5.6. Processing Elements

The computational processing element of the FTP/FTSM, also called the CP, is composed of two processing devices: a 68030 CPU and a 68882 floating-point co-processor. A specific processor choice was made for the FTP/FTSM so that the implementation of some of the necessary functions could be illustrated. The function implementations may be processor-specific, but the functions themselves are not. The use of other processing elements is not precluded by the FTSM architecture. However, if another processing element is to be used, different implementations of some of the functions may be necessary.

One of the requirements for the processor is that the minimum operating frequency must be at most one-half the maximum operating frequency. This requirement is imposed by the fault-tolerant clock, which makes occasional adjustments to the processor clock frequency to ensure that the processors in each FCR are synchronized (See section 4.2.1).

The 68882 coprocessor is included to enhance the performance of floating-point operations. The coprocessor is connected to the 68030 using the standard coprocessor interface defined
by Motorola\textsuperscript{31}. A special synchronizing device is required for the AS, DS, and DSACK signals to ensure clock deterministic behavior\textsuperscript{32}.

5.7. Local Memory

The memory map for the FTP/FTSM is shown in figure 5-8. The FTP/FTSM only uses the lower 24 bits of the address supplied by the 68030. This is done to simplify the address decode logic. Part of the memory address space is reserved for local memory, both in the form of RAM and ROM. The local RAM is intended for use during periods when the FTSM is not available. The ROM region is for permanent storage of programs and data.

![Memory Map Diagram]

**Figure 5-8:** FTP/FTSM Memory Map

Both types of memory are controlled by the local memory controller (LMC). The LMC decodes the addresses generated by the 68030 and enables the appropriate device. The LMC also generates the proper number of wait states (where applicable) before asserting DSACK to ensure that the selected memory device has enough access time.
5.7.1. PROM

The PROM provided in the FTP/FTSM is useful for permanent storage of the operating system, application programs, and small amounts of data. The PROM exists at the same location for all FCRs. The values read from the PROM are considered to be fault-tolerant because the PROMs provide quadruplicated information to the FTP. If at most one PROM fails, three of the four processors will still function correctly.

If desired, PROM could be made a part of the FTSM system. The only operation that could be performed on a read-only region of the FTSM memory map would be the class I read. However, many of the improvements that the FTSM provides over traditional replication schemes do not apply appreciably to PROM devices. Transient failures of PROM are not expected to be as common as for RAM. Also, PROMs do not usually have as many independent bit failure modes as RAM devices. Since the primary benefits gained by the use of the FTSM do not apply to PROM memory, the inclusion of PROM in the FTSM is not compelling.

5.7.2. RAM

Local RAM is provided by small number of 32-bit wide static-RAM devices. The local SRAM is used for several situations, including initial synchronization and reintegreation. In these situations, the processors are not synchronized, so the fault-tolerant shared memory can not be accessed. The local RAM is used until synchronization is achieved.

Part of the memory map is reserved for local RAM memory. This memory is quadruplicated in the same way as a conventional quad-FTP. However, the local RAM is not intended to be the primary memory for the system. Instead, it provides a scratchpad region for the processor in certain situations where the FTSM system is not available. These situations include the pre-initial synchronization period following power-up, reintegreation of a failed FCR, and severe fault degradation of the system. If desired, the local RAM can be used during normal system operation. However, it suffers from the same drawbacks as other FTPs with triplicated or quadruplicated memory systems.

5.8. Miscellaneous Devices

The description of other devices connected to the synchronous bus are presented below.
5.8.1. Multi-Function Peripheral

One I/O device which deserves special note is the 68901 multi-function peripheral, or MFP. This device provides several useful functions, including discrete I/O, serial I/O, and timing. The MFP is located in the simplex I/O space.

The discrete I/O is useful for a number of applications, including performance analysis, control of a small number of external binary devices, and debugging. The serial I/O port can be used for low-speed communication, program download, and for operating the FTP/FTSM through an interactive debugger. The internal timing devices are used to generate periodic interrupts to the CP. These periodic interrupts are useful for task scheduling and time-keeping functions.

The MFP uses the fault-tolerant clock to trigger the counter modules. The FTC is used because it is phase-locked between functioning FCRs. The serial communication port on the MFP uses a programmable baud-rate generator as a timing reference.

5.8.2. Control Registers

The FTP/FTSM has a number of control/status registers located in the C/S region of the memory map. Each register should be addressed as a 32-bit port, regardless of the actual data length to be read or written. The memory locations of each register are shown in figure 5-9.

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00800000</td>
<td>SRA</td>
<td>Syndrome Register-Channel A</td>
</tr>
<tr>
<td>00800004</td>
<td>SRB</td>
<td>Syndrome Register-Channel B</td>
</tr>
<tr>
<td>00800008</td>
<td>SRC</td>
<td>Syndrome Register-Channel C</td>
</tr>
<tr>
<td>0080000C</td>
<td>SRD</td>
<td>Syndrome Register-Channel D</td>
</tr>
<tr>
<td>00800010</td>
<td>SAR(A)</td>
<td>Syn. Addr. Reg.-Channel A</td>
</tr>
<tr>
<td>00800014</td>
<td>SAR(B)</td>
<td>Syn. Addr. Reg.-Channel B</td>
</tr>
<tr>
<td>00800018</td>
<td>SAR(C)</td>
<td>Syn. Addr. Reg.-Channel C</td>
</tr>
<tr>
<td>0080001C</td>
<td>SAR(D)</td>
<td>Syn. Addr. Reg.-Channel D</td>
</tr>
<tr>
<td>00800020</td>
<td>SRR(A)</td>
<td>Syn. Reg. Reset-Channel A</td>
</tr>
<tr>
<td>00800024</td>
<td>SRR(B)</td>
<td>Syn. Reg. Reset-Channel B</td>
</tr>
<tr>
<td>00800028</td>
<td>SRR(C)</td>
<td>Syn. Reg. Reset-Channel C</td>
</tr>
<tr>
<td>0080002C</td>
<td>SRR(D)</td>
<td>Syn. Reg. Reset-Channel D</td>
</tr>
<tr>
<td>00800040</td>
<td>CR</td>
<td>Command Register</td>
</tr>
<tr>
<td>00800044</td>
<td>WT</td>
<td>Watchdog Timer Register</td>
</tr>
<tr>
<td>00800050</td>
<td>DMRA</td>
<td>Decoder Mode Reg.-Channel A</td>
</tr>
<tr>
<td>00800054</td>
<td>DMRB</td>
<td>Decoder Mode Reg.-Channel B</td>
</tr>
<tr>
<td>00800058</td>
<td>DMR(C)</td>
<td>Decoder Mode Reg.-Channel C</td>
</tr>
<tr>
<td>0080005C</td>
<td>DMRR(D)</td>
<td>Decoder Mode Reg.-Channel D</td>
</tr>
</tbody>
</table>

Figure 5-9: FTSM Control/Status Registers

A description of each set of registers is given below.
5.8.2.1. Syndrome Registers

The syndrome register (SR) and syndrome address register (SAR) collect non-zero syndromes from the decoder and corresponding addresses from the address pipeline. There is a separate syndrome and syndrome address register for each FCR. Each FCR may see different syndromes so the syndrome and address must be treated as single-source data. The syndrome from the SR and the address from the SAR can be used by the fault detection, isolation, and recovery software (FDIR) to attempt to determine the location and nature of faults in the FTP/FTSM system.

The syndrome register is implemented by a 32-bit wide FIFO buffer. Every time the processor reads a syndrome out of the SR, the syndrome at the end of the FIFO is discarded and the next syndrome, if there is one, is ready to be read out. Each value in the SR is actually four syndromes, with each byte representing the syndrome for the corresponding byte in the data word. Since the FTSM always exchanges and decodes a 32-bit word, even if the processor only requests a single byte, the four syndromes are always calculated. The format for the syndrome register is shown in figure 5-10.

\[
\begin{array}{c|c|c|c}
31 & 24 & 23 & 16 \\
\hline
\text{MSB syndrome} & \text{UMB syndrome} & & \\
15 & 8 & 7 & 0 \\
\hline
\text{LMB syndrome} & \text{LSB syndrome} & & \\
\end{array}
\]

Figure 5-10: Syndrome Register

The syndrome address register is a 24-bit wide FIFO, the format of which is described in figure 5-11. The SAR is a 32-bit read only port. Since the FTP/FTSM is a 24-bit address machine, the upper 8-bits of the SAR are undefined. As long as the SR and SAR are read subsequently, the contents of the SR and SAR should correspond to the same memory location.

\[
\begin{array}{c|c|c|c|c}
31 & 24 & 23 & 16 & \\
\hline
& & & & \\
15 & 8 & 7 & 0 & \\
\hline
\text{Address of MSB of corresponding syndrome} & & & & \\
\end{array}
\]

Figure 5-11: Syndrome Address Register
The SR and SAR can be reset by accessing the syndrome register reset (SRR) location. An access, either read or write, to the SRR will cause the SR and SAR FIFOs to be reset to the empty condition. Four reset locations are defined, one for each FCR.

5.8.2.2. Command Registers

The command register, shown in figure 5-12, is used to request a synchronized hardware exception, such as an interrupt or reset, from software. The command register is used primarily for initial synchronization, fault recovery, and reconfiguration.

![Command Register Diagram](image)

**Figure 5-12: Command Register**

The Q-bit selects whether or not to use the watchdog timer. If the watchdog timer is selected, the command will not be executed until either unanimity is achieved or a majority is received and the watchdog timer is expired. The watchdog timer is started as soon as a majority is received. If the watchdog timer is disabled, the command will be executed as soon as a majority is observed. The definitions of majority and unanimity depend on the number of vote mask bits set.

The vote mask bits determine whether the respective FCR is to be included by the majority voter. In order from left to right, the bits represent FCRs A, B, C, and D. If none of the bits are set, all FCRs will be included in the vote. In this case, a majority is 3/4 and unanimity is 4/4. If the bit for an FCR is set, that FCR will not be included by the voter, and the definitions of majority and unanimity will change (see section 4.1.4).

The command and message bits are described in section 5.1.4.

The timeout length is controlled by the watchdog timer register (WT), shown in figure 5-13. The 8-bit number stored at this location indicates the number of FTC cycles to wait after receiving a majority in the command voter before asserting the timeout signal.
5.8.2.3. FTSM Control Registers

The decoder mode register (DMR) selects the mode (random, erasure, etc.) to be used when decoding values as they are read from the FTSM. Since each FCR may see different fault manifestations, there must be a separate mode register for each FCR. Each mode register is 16 bits wide, aligned on a 32-bit word boundary. The mode register is composed of four fields; each field is four bits wide. Each field determines the mode for a different byte-sliced decoder. The format of the DMR is shown in figure 5-14.

The defined modes are:

1111 - Random
0000 - Erase A
0001 - Erase B
0010 - Erase C
0011 - Erase D
1000 - Duplex AB
1001 - Duplex AC
1010 - Duplex AD
1101 - Duplex BC
1110 - Duplex BD
1100 - Duplex CD

The random mode indicates that all four FCRs are believed to be functioning correctly. The erasure modes indicate that one of the FCRs is suspect and that its symbol is possibly corrupted. The duplex modes are used when only two FCRs are believed to be working. Duplex mode allows the recovery of data from only two symbols.
6. Analysis of FTP/FTSM

This section analyzes the FTP/FTSM with respect to system performance and reliability. Several design decisions are justified by their impact on one or both of these aspects. The system is also compared to other existing FTP designs.

6.1. Reliability

The purpose of the reliability study is to compare the reliability of the FTP/FTSM with other systems and justify some of the design decisions.

6.1.1. Combinatorial Model

A combinatorial model can be used to make a rough comparison between the FTSM approach to memory reliability and other approaches, such as triplication or single-bit error correction. The model simply calculates the probability that, for a given bit reliability, the error patterns that occur in a memory location are correctable. The models for each system are derived by enumerating all of the possible correctable error patterns.

To analyze the redundancy systems, we will calculate the probability that, for a given bit reliability, $R_b$, a single memory location in the system being analyzed is in a correctable state. The bit reliability is the probability that at any instant, a specific bit has not failed. The reliability is generally a function of time. The function $R_b(t)$ is the probability that at time $t$, the bit is not failed if the bit was not failed at time 0. The probability that a location is in a correctable state is the location reliability, $R_L$. The memory system is composed of $K$ locations, so the memory system reliability is:

$$R_L^K$$

In a triplex memory system with bitwise majority voting, each bit position in a memory location can be treated independently. The overall location reliability is the bit-position reliability raised to the $n$th power, where $n$ is the number of bits in the memory location. For each bit position, the probability of a correctable error pattern is the probability that none of the bits are in error plus the probability that only one of the three bits is in error. The probability that no bits are in error is:

$$R_b^3$$
and the probability that one bit is in error is:

$$3 R_b^3 (1-R_b)$$

Therefore, the reliability of a memory location in a triplicated memory system is:

$$R_{L_{mem}} = \left( R_b^3 + 3 R_b^2 (1-R_b) \right)^n$$

The model for a memory system employing a one-bit error correcting code, such as a Hamming code, can be derived in a similar manner\textsuperscript{34}. A Hamming code requires n data bits plus m parity check bits. The reliability of a memory location is the probability that no bits are in error plus the probability that only one bit is in error, thus the location reliability is:

$$R_{L_{acc}} = R_b^{n+m} + (n+m) R_b^{n+m-1} (1-R_b)$$

The model for the FTSM system is slightly more complicated. The code used by the FTSM has a number of correctable error patterns, and these patterns are dependent upon what mode (random, erasure, or duplex) the FTSM is operated in. For simplicity, only the random mode will be analyzed here. This will give a lower bound for the FTSM reliability, since the modes which are not modeled will enhance reliability. The correctable patterns in the random mode fall into two categories: single symbol errors and double bit errors. Note that these two categories are not mutually exclusive. Some patterns fall into both.

The first case to consider is the possibility that none of the bits in a memory location are in error. For a code with s symbols and m bits in each symbol, the probability of no errors is:

$$R_b^m s$$

To evaluate the single symbol error cases, we will first choose a symbol to be in error. This will simplify the enumeration of the possible patterns. The total probability of exactly one symbol error is the probability that only the chosen symbol is in error times the number of symbols, s (s=4 in the FTSM). The total probability of a correctable single symbol error is:

$$s R_b^{m(s-1)} \sum_{i=1}^{m} \binom{m}{i} R_b^{m-i} (1-R_b)^i$$

To add in the capability of correcting double bit errors, we must consider only the cases where two symbols are involved, since double bit errors residing in the same symbol have
already been accounted for by the single symbol error pattern analysis. Again, we choose ahead of time two symbols to be in error. The total number of possibilities is the probability of two failed bits in two specific symbols times the number of ways to choose two bad symbols. The notation for the number of combinations of two bad symbols is:

$$\binom{s}{2} = \frac{s!}{(s-2)! \cdot 2!}$$

For two given symbols in error, there are \( m^2 \) possible double bit errors. Keep in mind that we are not counting the cases where fewer than two of the bits are in error; these cases have already been accounted for. We know that exactly one bit in each symbol is in error. The total probability of double bit errors which do not line up in the same symbol is:

$$\binom{s}{2} R_b^m (1-R_b)^2$$

Therefore, the overall location reliability is:

$$R_{L_{pm}} = R_b^m + \sum_{i=1}^{m} \binom{m}{i} R_b^{m-i} (1-R_b)^i + \binom{s}{2} R_b^m (1-R_b)^2$$

The combinatorial models presented above were evaluated for different bit reliabilities. The bit reliability was defined as a function of time, specifically:

$$R_b(t) = e^{-\lambda t}$$

where \( \lambda \) is the transient bit failure rate. The estimate for \( \lambda \) was derived from permanent memory failure rates determined in [35] and was estimated to be about 2.3 x 10^{-9}/hour/bit. Transient errors are assumed to be about 100 times as frequent as permanent errors. The combinatorial model treats transient and permanent errors identically, so it can be argued that the transient failure rate is the dominant factor in the combinatorial analysis.

Figure 6-1 shows a graph comparing the combinatorial reliability models for each of the three systems discussed. The probability of system loss is considerably less in the FTSM than it is in either of the other memory systems, demonstrating the inherent advantage of the (4-2) code over other informational redundancy schemes.
Figure 6-1: Combinatorial Reliability Graph

Unfortunately, the combinatorial reliability models are not completely accurate. The models do not consider the effect of memory reads, writes, and scrubs, all of which affect the reliability of a real memory system. Also, the models do not consider any reconfiguration capabilities. In spite of these weaknesses, the combinatorial model does provide a direct comparison on equal terms between various approaches to fault-tolerant memory systems.

6.1.2. Markov Model

The Markov model, shown in figure 6-2, improves the reliability analysis by incorporating more realistic time-varying behavior. This model attempts to study the impact that memory accesses (reads, writes, and scrubs) have on memory system reliability. The model shown is the model for a single memory location in the FTSM. To calculate memory system reliability, the location reliability as determined by the model is raised to the Kth power.

The Markov model consists of a number of states, with each state representing a different error condition. Transitions between states are represented by constant transition rates for an approximation of real behavior. The first state corresponds to the no-errors condition. The second row of states are the single error states. The transition from no errors to 1 transient error occurs with a frequency of $4\lambda_n$, where $\lambda_n$ is the transient symbol failure rate. The permanent failure rate, $\lambda_p$, is the frequency with which permanent symbol faults occur.
\( \lambda_{nt} \) Transient failure rate of memory symbol
\( \lambda_{np} \) Permanent failure rate of memory symbol
\( \mu_{ar} \) Processor read rate
\( \mu_{aw} \) Processor write rate
\( \mu_{scr} \) Scrubbing rate
\( \lambda_{bp} \) Permanent failure rate of memory bit
\( n \) Number of bits per symbol
\( \lambda_{np} = n \cdot \lambda_{bp} \)
\( \lambda_{nt} = 100 \cdot \lambda_{np} \)

**Figure 6-2: Markov Model**

Traditional estimates of the transient error rate vary from 10 to 100 times the permanent error rate\(^{36}\). For this analysis, the transient failure rate is assumed to be 100 times the permanent failure rate, thus yielding a conservative reliability estimate. The symbol failure rates are related to the bit failure rates by:
\[ \lambda_n = m \cdot \lambda_b \]

Recovery from the 1 transient error state occurs by either writing or scrubbing the location. In the FTSM, reading will not recover the location, even though the error will be masked when the value at that location is delivered to the processor. The scrub rate, \( \mu_{scr} \), is a constant equal to the frequency at which scrub operations can be performed times the total number of scrub operations necessary to complete one pass through memory (note that four locations are scrubbed during one scrub operation on the FTSM). The write rate, \( \mu_{aw} \), is highly location dependent and can be expected to vary by many orders of magnitude throughout the memory system\(^{37} \). However, if the write rate is much lower than the scrub rate, the scrub rate will dominate the recovery transition. This illustrates how the scrub rate is able to place a lower bound on the recovery rate from a single transient error.

Recovery from a permanent error is not modeled in this analysis. However, some measure of recovery can be achieved in the actual system by using the erasure and duplex modes. The erasure and duplex reconfiguration modes were eliminated to simplify the model. Since these modes provide considerably less than 100% fault coverage, the reliability gain expected by using them is not likely to be significant. Since the reconfiguration modes can only improve reliability, the results can be considered conservative estimates of the actual system reliability.

The third row of states corresponds to two or more symbol errors. This condition is not correctable by the encoding system. However, recovery from some of these states is possible. If a location has any transient errors, those errors will be eliminated if the location is written to. However, if the location has two or more permanent errors, recovery is not possible, since nothing can eliminate permanent errors.

Performing a read access on a location in one of the multiple error states, represented by the \( \mu_{ar} \) transition, will result in a transition to the fourth row of states which represent system loss. A memory location in a multiple error state does not correspond to system loss, since the processor does not have corrupted data. To cause system loss, the processor must attempt to use the data in the corrupted location. For comparison, the model includes two system loss states. The first is for system loss due strictly to transient faults and the second is for system loss caused by at least one permanent failure. Both states represent system loss.
6.1.3. Parameter Study Using Markov Model

This section studies the effect of various parameters on the reliability of the FTSM predicted by the Markov model presented above. A complete description of the parameters and analysis procedures used to obtain the various analysis results is presented in appendix 9.6. Also included in appendix 9.6 are tables of raw data for some of the analysis results.

6.1.3.1. FTSM Encoded vs. Quadruplicated Memory

Figure 6-3 shows a comparison between the FTSM, which uses an encoding function for fault masking, and a similar shared memory system using a quadruplicated memory with a majority vote function for fault masking. The Markov model was used to model both system, the only significant difference being the number of bits per symbol/copy. All parameters relating to the comparison are the same, including read rate, write rate, scrub rate, and bit failure rate. The graph demonstrates that the probability of system loss is reduced by a factor of 4 by reducing the memory requirement, and therefore the failure rate, by a factor of 2.

![Graph showing comparison between FTSM and quadruplicated memory]

Figure 6-3: Reliability of Encoded vs. Quad Memory

This analysis also demonstrates that transient errors are much less likely to cause system loss than permanent errors, even at short mission times. This characteristic is caused by the memory scrubber. The scrubber can correct transient errors if they are caught early enough so that an uncorrectable condition does not occur. The failure rate of a memory bit is about $2.3 \times 10^{-9}$ per hour. The failure rate is dominated by transient errors, which are estimated to be 100 times as frequent as permanent errors. The scrubber in the current design has a
maximum frequency of about 360 scrubs per hour. This means that every location is scrubbed every ten seconds. Since the scrubbing operation occurs much more often than transient errors, the effect of transient errors on system reliability is effectively eliminated.

6.1.3.2. Reliability vs. Memory Size

The size of the memory system has a dominant effect on the reliability of the FTP. The graph in figure 6-4 shows the effect of memory size on the reliability of the FTSM. As expected, the reliability decreases as the memory size increases. By studying the data in appendix 9.6, it is apparent that the system loss probability is directly proportional to the memory size.

![Graph showing the relationship between system loss probability and memory size.](image)

**Figure 6-4: Reliability vs. Memory Size**

6.1.3.3. Reliability vs. Scrub Rate

Program locality of reference indicates that most locations in a memory system are accessed very infrequently over short-periods of operation. A similar skewed workload distribution is also expected for long-term operation. Previous studies indicate that the workload distribution can have an effect on the reliability of a self-checking memory system. Memory access rates are assumed to be highly non-linear across the address space, with a few locations receiving many accesses and most locations being referenced very rarely. The graph in figure 6-5 shows how the access rate distribution might look for a real system, and a simple modeling estimate for the distribution. The value $f$ indicates the fraction of "fast"
memory locations which have the high access rate. Note that this value will probably be small (< 1%) for systems with a large amount of memory.

![Graph of Memory Access Distribution](image)

**Figure 6-5: Memory Access Distribution**

The graph in figure 6-6 shows the reliability of the FTSM memory system (without scrubbing) as a function of the percentage of so-called "fast" memory. This graph demonstrates that the reliability of the memory system is bounded by the reliability of the "slow" memory for all but unreasonably high percentages of "fast" memory. To improve the reliability of the FTSM system, the reliability of the "slow" memory must be increased. The

![Graph of Reliability vs. % Fast Memory](image)

**Figure 6-6: Reliability vs. % Fast Memory**
Memory scrubber accomplishes this task by periodically accessing every location in memory at a fixed rate, placing a lower-bound on the access rate, and thereby the reliability, of the "slow" memory.

The memory scrubber is included to reduce the probability of "slow" memory locations accumulating errors that become uncorrectable. The scrubber functions by reading a location in memory, exchanging, decoding, re-encoding, and re-writing the value to memory. The scrubber requires eleven clock cycles to complete the scrub of four memory locations, but six of these cycles are appended to the end of a processor memory access, so only five can actually interfere with processor throughput. A scrubbing regulator keeps the scrubber from operating too often. The period of the scrubbing regulator can be increased to reduce scrubbing impact on performance or decreased to increase reliability.

The effect of scrubbing rate on system reliability is shown in figure 6-7 for an FTSM with 0% "fast" memory. The "fast" memory was ignored since it has little effect on reliability (see appendix 9.6 for actual access rates). Both transient and permanent system loss probabilities are shown. Scrubbing has little effect on permanent errors, other than their early detection, so the permanent system loss probability is nearly constant with respect to scrubbing rate. The "knee" in the transient system loss probability graph occurs when the scrubbing rate is equal to the memory access (write) rate. If the write rate is higher than the scrubbing rate, the scrubbing operation has little effect on the reliability, whereas a higher scrubbing rate dominates the reliability effects.

![Figure 6-7: Reliability vs. Scrub Rate](image)

86
Several interesting conclusions can be drawn from this graph. First, scrubbing is more important for increasing short-term system reliability than for long-term reliability. This is easily understood since transient errors, which are correctable by the scrubber, dominate short-term reliability, whereas uncorrectable permanent errors dominate long-term reliability. Also, relatively slow scrub rates are sufficient for a measurable increase in short-term reliability. At 10 hours, the system loss probability can be reduced to the limits of permanent failures with a scrub rate of 30 per hour. The hardware implemented scrubber of the FTSM is assumed to have a scrub rate of 360 per hour with a worst-case performance penalty of 1.2% (assuming the processor collides with every scrub cycle).

The required scrub rate may be low enough that the scrubbing function can be implemented in software instead of hardware. Rough estimates indicate that a software scrubber task implemented in a 25 Hz system with a 1% performance penalty, which would run for 400 μs at 40 ms intervals, would be able to scrub approximately 256 bytes every task cycle. The entire 4 Mbytes of memory could be scrubbed 5.5 times per hour. This scrub rate is not quite sufficient to improve the reliability at 10 hours, though it is sufficient for anything over 100 hours. Also, it assumes a rather modest iteration rate. The inclusion of the hardware scrubber may be useful, especially in systems with high iteration rates where a software scrubber would impose extremely high performance penalties. However, the increased hardware complexity may make the hardware scrubber undesirable in other circumstances.

The issue of hardware or software implementation of the scrubber is indeterminate without more information about the actual application in which the FTP/FTSM is to be used. Some systems may be able to implement a scrubber in software to obtain a significant reliability improvement. The performance of other systems may be degraded too much by a software scrubber, requiring the scrubbing function to be implemented in hardware. The hardware scrubber should never be a significant cause of performance degradation since the currently selected scrub rate of 360/hour, which is more than fast enough for significant reliability improvement, imposes less than 2% penalty on the FTP/FTSM throughput. However, the permanent failure rate of the latter system may be increased due to the increased complexity of the hardware. The analysis presented above, while inconclusive, can be used as the basis for a qualitative engineering tradeoff decision on this issue.
6.2. Performance

The primary interest in the performance analysis of fault-tolerant computers is the performance penalty imposed by the fault-tolerant mechanism. One way of analyzing the performance of a fault-tolerant computer is to compare the time required to complete a specific task on a comparable simplex processor to the time required to complete the same task on the fault-tolerant processor, complete with source congruency and output consensus operations. Another gauge of performance is the overhead imposed on a task by the fault-tolerance functions.

An accepted benchmark for both types of analyses is one iteration of a control loop, since FTPs are often used for real-time control systems. A typical application for an imbedded real-time system is flight control. The control loop for this application, shown in figure 6-8, consists of three parts. During the first part, data from input sensors is read by the host FCR and exchanged via the source congruency exchange. The actuator outputs are computed during the second part. The third phase involves voting the actuator outputs to ensure consensus and transmitting the output values to the actuators.

![Control Loop Iteration for Flight Control](image)

Figure 6-8: Control Loop Iteration for Flight Control

Each step in the control loop shown above can be classified as belonging either to the fault-tolerant task or to the computational task, as indicated by the labels A and B, respectively. Any step which would be executed by a simplex computer performing the control task is defined to be part of the computational task, even if the step involves non-computational operations, such as I/O. The function of the computational task is unchanged on different
architectures. The fault-tolerance task is composed of those steps which only apply to fault-tolerant computers. The fault-tolerance task varies depending on the functions available on the FTP being analyzed. The fault-tolerant task time is denoted by $T_A$ and the computational task time is defined as $T_B$. The task completion time for a design $Y$ is the sum of the computational task time and the fault-tolerance task time:

$$T_Y = \sum T_{A_Y} + \sum T_{B_Y}$$

Each FTP analyzed in this section is compared to a baseline design to permit direct comparison of different FTP architectures. This baseline design is a simplex processor, so it only executes the computational task. The task completion time for the baseline design is defined as $T_0$.

Direct comparison of competing architectures can be made by comparing the task completion time on any given design $Y$ to the task completion time on the baseline system. This comparison can be made using the task completion ratio:

$$TCR = \frac{T_Y}{T_0}$$

Another interesting parameter is the overhead imposed by the fault-tolerance. While overhead is not necessarily a valid metric for direct comparison, it is an indication of how much of the processor power is tied up with fault-tolerance functions. Overhead is defined as the ratio of the completion time of the fault-tolerance task to the overall task completion time:

$$OH = \frac{T_{A_Y}}{T_Y} = \frac{T_{A_Y}}{T_{A_Y} + T_{B_Y}}$$

6.2.1. Baseline System Performance Analysis

The baseline system for this performance comparison is a 12.5 MHz 68020 processor with zero wait-state memory. This system was chosen because all of the designs to be compared use 68000 architecture processors, making direct comparison much easier. The NEFTP and VLSI-FTP both use 68020s, and the FTP/FTSM uses a 68030. No significant difference in throughput per processor clock is expected between the 68020 and the 68030 core microprocessors since the instruction set architectures are similar\textsuperscript{39, 40}.
To compare throughput of each of the FTP architectures with the baseline system, a step-by-step translation will be presented which will account for all of the significant differences between the baseline system and the FTP being analyzed.

6.2.2. NEFTP Performance Analysis

The current incarnation of the NEFTP uses processing elements which are identical to the baseline system. The only translation needed to obtain the NEFTP from the baseline system is the addition of the network element.

The control loop as it might be coded on the NEFTP is shown in figure 6-9. The NEFTP has three different functions which are performed during one iteration of the control loop. The FROMx source congruency is used to exchange single-source data from input sensors hosted by each processor. The VOTE function is used to vote actuator output values. SYNC is used to ensure processor synchronization. The penalties imposed by executing each of these functions are tabulated in appendix 9.6. The times given for FROMx and VOTE are for packet sizes of 240 bytes.

![Figure 6-9: Control Loop for NEFTP](image)

The completion time for the computational task is unchanged from the baseline, since the processing elements are identical to the baseline system. The fault-tolerance task as shown in figure 6-9 is composed of four FROMx operations, one VOTE, and two SYNCs. The total time for completion of the fault-tolerance task is 2.21 ms. Note that for a given packet size, the fault-tolerance task time is a constant. The total task completion time for the NEFTP is:

\[ T_{\text{NEFTP}} = T_0 + 2.21\text{ms} \]
6.2.3. VLSI-FTP Performance Analysis

There are two translations from the baseline design to the VLSI-FTP. The first is an increase in clock frequency. The second translation is the interstage exchange mechanism which implements the necessary interchannel data exchange functions.

The computational task time in the VLSI-FTP is different than that in the baseline design because of the difference in clock speed. The VLSI-FTP uses 16 MHz 68020 processors, instead of the 12.5 MHz processors in the baseline system.

\[ T_{B_{\text{VLSI-FTP}}} = T_0 \frac{12.5 \text{ MHz}}{16 \text{ MHz}} \]

The VLSI-FTP implementation of the control loop is shown in figure 6-10. The processors in the VLSI-FTP are micro-synchronized using a free-running fault-tolerant clock composed of four phase-locked channels, so no explicit SYNC operation is necessary. However, synchronization does contribute to the fault-tolerance overhead. A fault-tolerant clock is nominally 18 cycles of a 16 MHz clock. Two adjustments to the processor clock reduce the number of processor clocks per FTC cycle to 16. The result is that only \(8/9\) of the 16 MHz clock cycles are available for processor computation. The \(1/9\) cycles that are lost to synchronization need to be included with the fault-tolerance task completion time, since synchronization is a part of the fault-tolerance functionality. The following equation represents the time required to complete the computational and synchronization tasks:

![Figure 6-10: Control Loop for VLSI-FTP](image-url)
\[ T_B + T_{\text{sync}} = \frac{9}{8} T_B \]

The time to complete the computation plus the synchronization is 9/8 the time required to complete only the computation. By solving for \( T_{\text{sync}} \), the synchronization time can be determined and included with the calculation of \( T_A \).

FROMx and VOTE functions take the same amount of time to complete, since all message exchanges pass from processors to interstages and back to the processors. The effective interconnect width is 16 bits. 2.25 \( \mu \)s are required to exchange one 16 bit word, so each 240 byte FROMx and VOTE requires 270 \( \mu \)s. The total fault-tolerance task completion time including the loss to synchronization is:

\[ T_{A_{\text{VLSFTP}}} = \frac{1}{8} T_0 \frac{12.5 \text{ MHz}}{16 \text{ MHz}} + 1.35 \text{ ms} \]

### 6.2.4. FTP/FTSM Performance Analysis

Translation of the baseline design to the FTP/FTSM requires three steps. First is an increase in clock frequency. The second is the addition of the FTSM and its associated fault-tolerance functions. Finally, a data cache is added to boost performance.

The 68030 processor in the FTP/FTSM runs at 25 MHz, which is twice the frequency of the baseline. The core 68030 is assumed to have approximately the same performance per clock cycle as a 68020, since the architecture and instruction set are nearly identical. Consequently, the raw performance of the processor in the FTP/FTSM will be doubled compared to the baseline.

The two major fault-tolerance functions which impact the performance of the FTP/FTSM are the class I and class II data exchanges. Each function affects performance in a different way.

Class II exchanges are an explicit operation, invoked by the executing task. Therefore, the class II performance penalty is a constant for a given data exchange size, as it is in the NEFTP. Each complete class II exchange requires 17 processor clock cycles (7 for the write, 10 for the read) for a total of 680 ns per 32-bit word. 240 bytes of data from each FCR would require a total of 163.2 \( \mu \)s to exchange.

Class I exchanges are implicitly invoked by the processor when performing an access to the FTSM. Since class I writes are completed in the minimum amount of time, only class I reads contribute to the degradation of performance.
Synchronization has very little impact on the performance of the FTP/FTSM. The FTC period is nominally about 5 μs long, with a processor clock adjustment of 20 ns. The total performance impact is about 0.4%, so the synchronization overhead will be neglected in the performance analysis of the FTP/FTSM.

The control loop for the FTP/FTSM is shown in Figure 6-11. Since the FTP/FTSM is micro-synchronized, no explicit SYNC operation is necessary. Also, the fault masking properties of the VOTE function are furnished by the class I exchange. Since the class I exchange is invoked implicitly, a step analogous to the VOTE step in the NEFTP and the VLSI-FTP is not necessary. However, the class I is initiated throughout the computational task, so part of the time spent performing the computational task should actually be attributed to the fault-tolerance task.

![Figure 6-11: Control Loop for FTP/FTSM](image)

The determination of $T_{FTP/FTSM}$ begins by analyzing the amount of time the baseline system takes to complete the computational task. Since only memory reads affect FTP/FTSM performance during the computational task, we can divide the task time into time spent doing memory reads and time spent doing other things. For the baseline computer, this time can be expressed as:

$$T_0 = T_0 r_0 + T_0 (1-r_0)$$

where $r_0$ is the fraction of processor time devoted to memory reads.

The performance penalty imposed by class I reads appears to the processor to be the result of slow memory. The processor requires a minimum number of cycles to complete a read cycle.
This value is denoted $n_m$ and is equal to 3 for the 68020/68030. The FTSM system imposes a penalty of $p$ wait states to every memory read. The total number of cycles required to complete a read access from the FTSM is $(p + n_m)$ cycles. For the FTSM system with read penalty, the task completion time is:

$$T_p = \frac{12.5\text{MHz}}{25\text{MHz}} \left( T_0 r_0 \left( \frac{p+n_m}{n_m} \right) + T_0 (1-r_0) \right)$$

An independent analysis can be made of the impact of the data cache on the performance of the baseline computer. By application of the well-known cache equation\(^\text{41}\), the task completion time for the baseline design with cache is:

$$T_{0_c} = \left( T_0 r_0 \left( \frac{q n_c + (1-q) n_m}{n_m} \right) + T_0 (1-r_0) \right)$$

where $n_c$ is the cache access time (equal to 1 for the 68030 on-chip caches) and $q$ is the hit ratio of the cache system. Combining the last two equations gives the total time necessary to complete the computational task:

$$T_{B_{\text{FTSM}}} = \frac{1}{2} \left( T_0 r_0 \left( \frac{p+n_m}{n_m} \right) \left( \frac{q n_c + (1-q) n_m}{n_m} \right) + T_0 (1-r_0) \right)$$

This equation includes the actual computation time plus part of the fault-tolerance time. By separating the first term to yield:

$$T_{B_{\text{FTSM}}} = \frac{1}{2} \left( T_0 r_0 \left( \frac{p}{n_m} \right) \left( \frac{q n_c + (1-q) n_m}{n_m} \right) + T_0 r_0 \left( \frac{n_m}{n_m} \right) \left( \frac{q n_c + (1-q) n_m}{n_m} \right) + T_0 (1-r_0) \right)$$

the partition can be readily seen. The first term corresponds to the time spent performing the penalty cycles imposed by the FTSM. This time should be counted with the fault-tolerance task time. The remaining terms make up the computational task time.

Combining the two fault-tolerance tasks gives:

$$T_{A_{\text{FTSM}}} = \frac{1}{2} T_0 r_0 \left( \frac{p}{n_m} \right) \left( \frac{q n_c + (1-q) n_m}{n_m} \right) + 163.2 \mu s$$

for the fault-tolerance task completion time for the FTP/FTSM. The computational task completion time is:
\[ T_{B_{FTP/FTSM}} = \frac{1}{2} \left( T_0 r_0 \left( \frac{q n c^q (1-q) n_m}{n_m} \right) + T_0 (1-r_0) \right) \]

The parameters in these equations must be determined before making a performance comparison. Some of the parameters are dependent on the application program and can not be determined exactly without evaluation of a real application. For these cases, estimates were made. Appendix 9.6 lists each parameter, an estimated value, and the conditions under which the parameters were determined.

### 6.2.5. Performance Analysis Results

The graph in figure 6-12 show the overhead consumed by the fault-tolerance task for each FTP architecture at a given control loop iteration rate. The iteration rate is the reciprocal of the total task completion time, \( T_Y \). Note that since \( T_{AY} \) varies between the different designs, \( T_{BY} \) is not the same on different designs for a given iteration rate. Also, each design has a different raw throughput. A design with high throughput and high overhead may be able to do more in one iteration at some frequencies than another design with low throughput and low overhead. For these reasons, the overhead should not be used for direct comparison between different designs. However, it does give an indication of how much processor power is available on a given design.

![Graph showing Overhead vs. Iteration Rate](image-url)

**Figure 6-12: Overhead vs. Iteration Rate**

The overhead in the FTP/FTSM at low frequencies is much greater than that in the NEFTP or the VLSI-FTP, because the penalty for fault masking is paid on every memory read.
However, at high frequencies, the FTP/FTSM excels, because the constant fault-tolerance overhead, which dominates the overhead at high frequencies, is much less in the FTP/FTSM than in either the NEFTP or the VLSI-FTP. Consequently, the FTP/FTSM is more appropriate for applications requiring higher iteration rates.

Direct throughput comparisons can be made using the task completion ratios, the ratio of task completion time on a specific design to the task completion time on the baseline design. Figure 6-13 shows the task completion ratios for the three designs presented in this performance analysis. The x-axis is \( T_0 \), the task completion time on the baseline design, and the y-axis is the task completion ratio. The TCR for the baseline is always one, by definition.

![Figure 6-13: Task Completion Ratios](image)

The TCR curve for the FTP/FTSM shows that its performance is about 1.7 times that of the baseline design for \( T_0 \) greater than 10 ms. The NEFTP and the VLSI-FTP both approach the raw performance of their respective simplex processors for tasks greater than 50 ms. For tasks between 50 ms and 1 ms in duration, the performance of the FTP/FTSM deteriorates at a much lower rate than for either of the other two designs. At 50 ms, the FTP/FTSM is about 1.7 times as fast as the NEFTP, whereas at 1 ms, the FTP/FTSM is 4 times as fast. This agrees with the observation made from the overhead graph. The FTP/FTSM is slightly better than the other designs at low frequencies, primarily due to enhanced processor performance. At high frequencies, the FTP/FTSM is much better since the constant component of the fault-tolerance task is much shorter.
The overhead in the FTP/FTSM is fairly high compared to other designs such as the NEFTP and the VLSI-FTP. The overhead limits the throughput of the FTP/FTSM at slow iteration rates to less than twice the NEFTP throughput, even though the FTP/FTSM uses a processor with twice the performance of the processor in the NEFTP. However, the NEFTP overhead begins to increase very rapidly at iteration rates above 50 Hz, whereas the overhead in the FTP/FTSM increases more slowly. As a result, we again conclude that the FTP/FTSM should have much better performance than the NEFTP at higher iteration rates.

It should be noted that the performance parameters for the VLSI-FTP and the NEFTP are actual empirical data measured on existing architectural implementations. The parameters, therefore, are reflective only of these specific implementations, and not of inherent architectural limitations. Indeed, analysis of the NEFTP indicates that the performance of that design can be improved by several architectural and software optimizations.\textsuperscript{42}

6.2.6. Performance Improvement of the FTP/FTSM

To improve the performance of the FTP/FTSM even further, the following factor must be minimized:

$$\left( \frac{p+n_m}{n_m} \right) \left( \frac{q n_c + (1-q) n_m}{n_m} \right)$$

This factor is the amount of performance improvement/degradation imposed on the FTP/FTSM as compared to a simplex 68030 (without a data cache). If the factor is equal to one, the performance of the FTP/FTSM will equal the baseline (25 MHz 68030) design. If it is less than one, the performance will be improved, and if it is greater than one, the performance will be degraded.

Minimization of the factor is accomplished by reducing the memory access penalty, \( p \), and increasing the cache hit ratio, \( q \). The access penalty of four clock cycles is probably about the smallest penalty that can easily be obtained. The IFC link will always have some latency, although a latency of one clock cycle instead of two may be possible. The processors must be able to tolerate non-zero skew, so the one cycle penalty for synchronization cannot be eliminated. Also, the decode operation takes some time to complete. The cache hit ratio could be increased with the addition of a cache external to the 68030, but the cache access time would also be increased since an external bus cycle would be required.
Another way to reduce the effect of fault-tolerance on performance is to reduce $r_0$. This parameter is highly dependent on the processor design and programming style. It may be possible to minimize memory accesses by using a RISC design with a load/store architecture, a large register set, and an optimizing compiler.
7. Future Enhancements

This thesis presents the FTP/FTSM architecture design as a new type of architecture for fault-tolerant computers. The preceding chapters discuss the fundamental architecture design and a few implementation-specific topics. This chapter presents an overview of a few possible improvements that could be made to the FTP/FTSM architecture.

7.1. Multiprocessor/Parallel Processor Architectures

An ever-increasing need for additional computer throughput has stimulated the design of computer systems using multiple processors, both in multiprocessor and parallel processor architectures. However, conventional multiprocessor and parallel processor architectures are unsuitable for use in situations where high reliability is required. The desire for high performance in fault-tolerant computers has led to the design of several multiprocessor and parallel processor designs, including MAFT, FTMP, and FTPP.

A multiprocessor architecture could be built which would allow multiple processors to access the fault-tolerant shared memory system. This configuration is shown in figure 7-1. Each FCR contains an equal number of processing sites and FTSM quadrants. The processing sites are arbitrarily grouped into quadrupled redundant processing groups. Each processing group submits to deterministic arbitration for access to the shared bus.

This configuration enhances the performance of the system by providing multiple processing sites with access to shared resources, such as memory and I/O. Reliability could also be improved since the arbitrary grouping of redundant groups would allow the inclusion of spare units, either processors, memory, or I/O, in each FCR. The problem with this design is that as more redundant groups of processors are attached to the shared bus, contention for the shared bus will increase. Bus arbitration and contention is the limiting factor in any multiprocessor design. Therefore, the total number of processing groups may not be able to exceed three or four in practice.
Figure 7-1: FTP/FTSM Multiprocessor Architecture

The multiprocessor architecture could be used as the basis for a parallel processor. The use of a quadruplicated redundant cluster for parallel processing has been discussed in [47]. The clusters for the fault-tolerant parallel processor, FTPP, are designed to be functionally synchronized. However, there is no reason why a micro-synchronized cluster couldn't be used instead. Further investigation into this possibility is necessary to determine possible advantages to using a FTP/FTSM cluster instead of the traditional functionally synchronized cluster.

7.2. Input/Output Processing

One of the limitations of the current FTP/FTSM design is the interface to I/O. Since the CP, the main processor in the FTP/FTSM, is clock deterministic, all I/O devices must either be clock deterministic or must communicate to the CP through a synchronizing interface, such as a dual-port RAM. The technique chosen for the FTP/FTSM is to use another processor, the IOP, which accesses I/O devices directly and communicates with the CP through the DPRAM. The CP does not have direct control of I/O devices, but instead sends I/O commands to the IOP. The IOPs in the current design are simplex processors executing different programs.
Another possible configuration for the IOP, shown in figure 7-2, is to use another FTP, such as the NEFTP, to perform the IOP functions. The NEFTP lends itself well to this type of application since it is a functionally synchronized design and uses a standard interface bus, namely the VMEbus. Since the NEFTP has the full capabilities of a Byzantine resilient FTP, input data can be exchanged through the source congruency operation before being transferred to the CP via the common I/O dual-port RAM. Since the source congruency can be performed by the NEFTP concurrently with CP execution, some performance improvement can be expected. Also, since the NEFTP is a full-fledged fault-tolerant computer, it can take over execution from the FTP/FTSM in the event of unrepairable faults in the FTP/FTSM.

![Diagram showing FTP/FTSM (CP) and NEFTP (IOP)]

**Figure 7-2:** Using the NEFTP as an IOP

### 7.3. External Caches

The primary performance penalty imposed on the FTP/FTSM by the fault-tolerant mechanisms is the additional wait-states required for performing reads from the FTSM, as demonstrated by the performance analysis in section 6.2. The performance analysis also demonstrated the benefit of using a data cache to minimize reads from the FTSM. The performance benefit provided by the data cache is highly dependent on the hit ratio, which must be increased to gain a performance benefit.
The hit ratio is dependent upon a number of variables, including the cache size, line size, and set size. It is also highly dependent on the memory access patterns of the applications being executed. The only practical way to determine a reasonable hit ratio for a particular architecture is through memory access simulation of a typical application program.

The 68030 has two internal caches, an instruction cache and a data cache. Each cache contains 16 lines of 4 entries, each entry being one aligned 32-bit word. The data cache supports 1, 2, 3, and 4 byte operations from the processor. Each entry in the caches is independently replaceable using 4 byte (longword) bus cycles to memory. The caches also support a 4 longword memory bus cycle called a cache burst. The cache burst is used to fill an entire cache line in a single indivisible cycle. The purpose of the cache burst is to increase the hit ratio by prefetching values expected to be used by the processor in the near future. In most systems, including the FTP/FTSM, a cache burst cycle takes much less time than four independent longword cycles.

An access to the internal caches can be completed in a single clock cycle. In contrast, a synchronous bus cycle (a enhancement to the 68000 architecture introduced on the 68030) requires a minimum of two clocks, and an asynchronous cycle (the standard bus cycle for the 68000-68020) requires a minimum of three clocks. With a good hit ratio, the performance of the 68030 can be improved by almost a factor of three with the use of the internal caches.

Unfortunately, the internal caches are so small that a good hit ratio ($q>0.9$) may be unobtainable with some applications. To enhance the performance of the FTP/FTSM, a larger external cache system may be required. The external cache would be used in conjunction with the internal caches so that if a miss occurs in the internal cache, the address will be looked up in the external cache before requesting a fetch from the FTSM. The external cache could be designed so that if a hit occurs, the external cycle can be completed in only two clock cycles. Also the cache should be able to support cache burst cycles, both for filling the 68030 internal caches through processor bus cycles and for filling its own lines during memory bus cycles. Since the dimensions of the external cache are unconstrained by the design of the 68030, the line size can be increased so that a burst access to the FTSM would request more entries (8 to 16) instead of the four requested by the 68030 internal caches. The FTSM is designed so that the length of cache burst cycles can be increased with little modification.

Two important considerations apply to both internal and external caches. The first is the need to periodically refresh the cache. An entry in the processor cache is assumed to be less reliable than a location in the FTSM, since all of the reliability enhancements applied to the
FTSM can not be used on the processor caches. Therefore, the cache entries should be periodically refreshed from the FTSM to ensure their integrity. This process is facilitated by a writethrough cache design. Since a writethrough cache always updates the copy in memory, a cache entry can be invalidated without a copyback cycle. The cache entries only need to be invalidated and allowed to refill through normal program execution.

The processor cache must also be considered part of the processor state that must be realigned before reintegration of a processor following a transient fault. As noted before, memory locations in the FTSM do not require realignment, thus saving a majority of the reintegration time compared to most conventional FTPs. However, some realignment is necessary, including processor registers and caches. The cache realignment is very simple. Since the caches are of the writethrough design, the FTSM contains up-to-date values for every location. The only step required to realign the caches is to invalidate all entries in all caches on all processors. Since all caches are empty, they can refill themselves, thereby providing realignment, through natural program execution.

7.4. Event-Driven I/O

The current FTP/FTSM design does not provide for I/O interrupts to the CP. The IOP can be event-driven, since it is not required to be clock deterministic or synchronized. However, the CP must poll the IOP to obtain status and input since there is no provision for single-source hardware initiated interrupts. The amount of time spent polling the IOP may be considerable. In a real-time control system, this wasted time may be intolerable.

The solution is to allow the CP to be event-driven. When the IOP has data ready for, or is ready to accept data from, the CP, it should interrupt the CP. After requesting a function from the IOP, the CP can resume processing until the IOP is ready to deliver the request.

The problem is that interrupts from the IOP must be treated as single-source data items. An interrupt must be exchanged using a source congruency operation and synchronized between FCRs before being used to interrupt the CP.

A mechanism has been provided in the FTP/FTSM to exchange and vote commonly sourced exceptions, all of which are currently invoked by software. The mechanism is part of the implementation of several necessary functions, including initial synchronization and reconfiguration. The mechanism could be modified to provide a source congruency function for single-source events.
8. Conclusion and Recommendations

The FTP/FTSM presented in this thesis represents an alternative architecture for Byzantine resilient computers. The primary benefits of the FTP/FTSM over other Byzantine resilient architectures, such as the VLSI-FTP or the NEFTP, are the elimination of memory realignment time due to the shared memory design, the improvement in short-term reliability obtained by the reduced memory requirement and the hardware implemented memory scrubber, the reduced fault-latency due to the continual and implicit fault-masking, and the improved performance resulting from the efficient exchange primitives.

The performance of the FTP/FTSM compares favorably with other FTP designs. The overhead due to fault-tolerance is greater than in other designs, especially at low iteration rates. However, the overhead is offset by an increase in raw processor throughput. The result is that the FTP/FTSM can outperform other designs even at low iteration rates. For iteration rates greater than 200 Hz, the overhead in the VLSI-FTP and the NEFTP is greater than the overhead in the FTP/FTSM. Consequently, the FTP/FTSM is well suited for applications requiring high iteration rates. The FTP/FTSM could theoretically operate at iteration rates as high as 6 kHz, although the overhead would be so high that little computation could be performed. A more reasonable limit might be 2 kHz, where the overhead is about 50%.

The reliability of the FTP/FTSM is also comparable to that of other fault-tolerant computers. The reliability of the FTP/FTSM at short mission times (around 100 hours in duration) is especially good, due to the elimination of transient errors as a significant source of potential system loss. At long mission times (on the order of 10000 hours) reliability without the possibility of reconfiguration is somewhat improved as a result of the reduction in memory as compared to a quad-FTP. However, some FTPs, the VLSI-FTP in particular, have more flexibility for reconfiguration around existing faults than does the FTP/FTSM. The VLSI-FTP has complete fail-op/fail-op/fail-safe capability. The FTP/FTSM can reconfigure around faults by changing the mode of the decoding operation from random to erasure, or erasure to duplex. However, the erasure mode is not guaranteed to mask all faults and the duplex mode will not mask any faults. Therefore, the FTP/FTSM is not fail-safe after detection of a single fault. The FTP/FTSM can therefore be either fail-safe or fail-op/fail-catastrophic.

The reliability models used to analyze the reliability of the FTP/FTSM are not completely accurate models of the system reliability. The combinatorial model does not take into account
the effect of memory accesses on the reliability of the memory system. The Markov model
does not consider any double-bit errors during the random mode, nor does it consider the use
of erasure or duplex modes. All of these factors influence the reliability estimates of the
FTP/FTSM. By further investigation, the Markov model could be improved to incorporate
the missing features, thus generating a more accurate model of the FTP/FTSM system
reliability.

The design presented in this thesis is a complete conceptual level design with some
consideration for implementation issues. The construction of a prototype is beyond the scope
of this report. A prototype is desirable for proof-of-concept demonstration, accurate
performance analysis, and code development. However, a design review of the conceptual
design and a complete implementation design should be performed before building a
prototype model.

The conceptual design makes the assumption that certain hardware devices will be available
in sufficient quantities at reasonable prices for the construction of the prototype. If availability
of these parts is not as expected, the performance of the design may have to be compromised.
The performance analysis in section 6.2 determined that the primary performance limitation
of the FTP/FTSM is the latency of the class I read operation. The latency of the operation
used for the performance analysis, four processor wait states, was determined by assuming
that the following devices are available: static memory devices with an access (read or write)
time less than 30 ns, a serial communication link with a bandwidth of 500 Mbits/sec and a
latency less than 80 ns, and a pipelined ASIC implementation of the FTSM encoder/decoder
with a decode time of less than 40 ns.

Software development is another area for future developments. A real-time operating system
with basic functions including a task scheduler, a memory allocator, and device drivers is
needed. Parts of the operating system for the NEFTP could possibly be used. However,
changes would be required to the message passing primitives (most of these would not exist
on the FTP/FTSM), the fault detection and isolation algorithms, and the I/O drivers.

The FTP/FTSM is proposed as an alternative architecture for a Byzantine resilient computer.
While the FTP/FTSM has a few disadvantages compared to other FTPs, such as the NEFTP
or VLSI-FTP, it also has a number of advantages over other Byzantine resilient architectures
which makes it worth considering for certain applications.
9. Appendices
9.1. Galois Fields

The (4,2) code is based on a Galois field containing 16 elements. The finite field contains a zero element, a unary element (one), and a primitive element (α). All other elements in the finite field can be expressed as a power of the primitive element. The field elements are generated using an irreducible polynomial in α. A list of each member of GF(2^4) is presented below. The left column shows each element expressed as a power of the primitive element. The column on the right gives the hexadecimal notation representing the polynomial expansion for each element. For more information on Galois fields, see [48] and [49].

Elements of GF(2^4) Generated from \( \alpha^4 + \alpha + 1 \)

<table>
<thead>
<tr>
<th>Powers of a</th>
<th>Polynomial Representation</th>
<th>Binary (hex) Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>= 0</td>
<td>0</td>
</tr>
<tr>
<td>( \alpha^0 )</td>
<td>= 1</td>
<td>1</td>
</tr>
<tr>
<td>( \alpha^1 )</td>
<td>= ( \alpha )</td>
<td>2</td>
</tr>
<tr>
<td>( \alpha^2 )</td>
<td>= ( \alpha^2 )</td>
<td>4</td>
</tr>
<tr>
<td>( \alpha^3 )</td>
<td>= ( \alpha^3 )</td>
<td>8</td>
</tr>
<tr>
<td>( \alpha^4 )</td>
<td>= ( \alpha + 1 )</td>
<td>3</td>
</tr>
<tr>
<td>( \alpha^5 )</td>
<td>= ( \alpha^2 + \alpha )</td>
<td>6</td>
</tr>
<tr>
<td>( \alpha^6 )</td>
<td>= ( \alpha^3 + \alpha^2 )</td>
<td>C</td>
</tr>
<tr>
<td>( \alpha^7 )</td>
<td>= ( \alpha^3 + \alpha + 1 )</td>
<td>B</td>
</tr>
<tr>
<td>( \alpha^8 )</td>
<td>= ( \alpha^2 + 1 )</td>
<td>5</td>
</tr>
<tr>
<td>( \alpha^9 )</td>
<td>= ( \alpha^3 + \alpha )</td>
<td>A</td>
</tr>
<tr>
<td>( \alpha^{10} )</td>
<td>= ( \alpha^2 + \alpha + 1 )</td>
<td>7</td>
</tr>
<tr>
<td>( \alpha^{11} )</td>
<td>= ( \alpha^3 + \alpha^2 + \alpha )</td>
<td>E</td>
</tr>
<tr>
<td>( \alpha^{12} )</td>
<td>= ( \alpha^3 + \alpha^2 + \alpha + 1 )</td>
<td>F</td>
</tr>
<tr>
<td>( \alpha^{13} )</td>
<td>= ( \alpha^3 + \alpha^2 + 1 )</td>
<td>D</td>
</tr>
<tr>
<td>( \alpha^{14} )</td>
<td>= ( \alpha^3 + 1 )</td>
<td>9</td>
</tr>
<tr>
<td>( \alpha^{15} )</td>
<td>= 1</td>
<td></td>
</tr>
</tbody>
</table>
The Galois field contains two binary operations defined over the elements of the finite field. These operations have the following properties:

- The operations are closed over the finite field.
- The operations are associative
- The operations are commutative
- The zero element is the additive identity element
- The unary element is the multiplicative identity element
- All elements have an additive inverse; each element is its own additive inverse.
- All elements have a multiplicative inverse except the zero element.

The additive and multiplicative tables are shown below.

### Additive Table

<table>
<thead>
<tr>
<th>+</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>7</td>
<td>6</td>
<td>9</td>
<td>8</td>
<td>B</td>
<td>A</td>
<td>D</td>
<td>C</td>
<td>F</td>
<td>E</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>4</td>
<td>5</td>
<td>A</td>
<td>B</td>
<td>8</td>
<td>9</td>
<td>E</td>
<td>F</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>B</td>
<td>A</td>
<td>9</td>
<td>8</td>
<td>F</td>
<td>E</td>
<td>D</td>
<td>C</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>8</td>
<td>9</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>4</td>
<td>7</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>D</td>
<td>C</td>
<td>F</td>
<td>E</td>
<td>9</td>
<td>8</td>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>7</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>E</td>
<td>F</td>
<td>C</td>
<td>D</td>
<td>A</td>
<td>B</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>F</td>
<td>E</td>
<td>D</td>
<td>C</td>
<td>B</td>
<td>A</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>9</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>8</td>
<td>B</td>
<td>A</td>
<td>D</td>
<td>C</td>
<td>F</td>
<td>E</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>B</td>
<td>8</td>
<td>9</td>
<td>E</td>
<td>F</td>
<td>C</td>
<td>D</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td>A</td>
<td>9</td>
<td>8</td>
<td>F</td>
<td>E</td>
<td>D</td>
<td>C</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>8</td>
<td>9</td>
<td>A</td>
<td>B</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>C</td>
<td>F</td>
<td>E</td>
<td>9</td>
<td>8</td>
<td>B</td>
<td>A</td>
<td>5</td>
<td>4</td>
<td>7</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>E</td>
<td>F</td>
<td>C</td>
<td>D</td>
<td>A</td>
<td>B</td>
<td>8</td>
<td>9</td>
<td>6</td>
<td>7</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>E</td>
<td>D</td>
<td>C</td>
<td>B</td>
<td>A</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

### Multiplicative Table

| * | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
| 2 | 0 | 2 | 4 | 6 | 8 | A | C | E | 3 | 1 | 7 | 5 | B | 9 | F | D |
| 3 | 0 | 3 | 6 | 5 | C | F | A | 9 | B | 8 | D | E | 7 | 4 | 1 | 2 |
| 4 | 0 | 4 | 8 | C | 3 | 7 | B | F | 6 | 2 | E | A | 5 | 1 | D | 9 |
| 5 | 0 | 5 | A | F | 7 | 2 | D | 8 | E | B | 4 | 1 | 9 | C | 3 | 6 |
| 6 | 0 | 6 | C | A | B | D | 7 | 1 | 5 | 3 | 9 | F | E | 8 | 2 | 4 |
| 7 | 0 | 7 | E | 9 | F | 8 | 1 | 6 | D | A | 3 | 4 | 2 | 5 | C | B |
| 8 | 0 | 8 | 3 | B | 6 | E | 5 | D | C | 4 | F | 7 | A | 2 | 9 | 1 |
| 9 | 0 | 9 | 1 | 8 | 2 | B | 3 | A | 4 | D | 5 | C | 6 | F | 7 | E |
| A | 0 | A | 7 | D | E | 4 | 9 | 3 | F | 5 | 8 | 2 | 1 | B | 6 | C |
| B | 0 | B | 5 | E | A | 1 | F | 4 | 7 | C | 2 | 9 | D | 6 | 8 | 3 |
| C | 0 | C | B | 7 | 5 | 9 | E | 2 | A | 6 | 1 | D | F | 3 | 4 | 8 |
| D | 0 | D | 9 | 4 | 1 | C | 8 | 5 | 2 | F | B | 6 | 3 | E | A | 7 |
| E | 0 | E | F | 1 | D | 3 | 2 | C | 9 | 7 | 6 | 8 | 4 | A | B | 5 |
| F | 0 | F | D | 2 | 9 | 6 | 4 | B | 1 | E | C | 3 | 8 | 7 | 5 | A |
9.2. Bus Cycles

Class I Write

A class I write occurs if the /WRITE signal is asserted and the function codes (FC) indicate an access to user program, user data, supervisor program, or supervisor data space.

1. Bus master asserts /ALERT
   Bus master presents address on address/data lines
   Bus slave latches address, SPC0-1, SIZ1-0, and /WRITE at the end of (1)

2. Bus master asserts /BCQUAL, qualifying write
   Bus master presents data on address/data lines
   Bus slave asserts /STERM
   Bus master detects /STERM low at the end of (2)

3. Data symbols written into SRAM
   Address register cleared at the end of (3)
   Bus master negates /BCQUAL, data, SPC1-0, and SIZ1-0 by the end of (3)

Class I Read

A class I read occurs if the /WRITE signal is not asserted and the function codes (FC) indicate an access to user program, user data, supervisor program, or supervisor data space.

1. Bus master asserts /ALERT
   Bus master presents address on address/data lines
   Bus slave latches address, SPC0-1, SIZ1-0, and /WRITE at the end of (1)

2. Bus master asserts /BCQUAL, qualifying read
   Bus master releases address/data lines
   Data symbols read out of SRAM
   Address register cleared at the end of (2)

3. Data symbols broadcast to other FCRs

4.

5.

6. Data symbols from other FCRs appear at FIFOs by the beginning of (6)
   Bus slave asserts /STERM
   Bus master detects /STERM low at the end of (6)
   Syndrome latch is enabled

7. Bus slave drives decoded data onto address/data lines
   Bus master latches data at the end of (7)
   Data symbols discarded from the end of FIFOs
   Bus master negates /BCQUAL, SPC1-0, and SIZ1-0 by the end of (7)

Class II Write

A class II write occurs if the /WRITE signal is asserted and the function codes (FC) indicate an access to the user-defined reserved space (FC=3).

1. Bus master asserts /ALERT
   Bus master presents address on address/data lines
   Bus slave latches address, SPC0-1, SIZ1-0, and /WRITE at the end of (1)
   SPC signals indicate a Class II cycle

2. Bus master asserts /BCQUAL, qualifying write
Bus master presents data on address/data lines
3. Source FCR broadcasts data symbols to recipient FCRs; recipient FCRs send null data
   Data symbols for source FCR present on M-Data lines
4.
5.
6. Data symbols from source FCR appear at FIFOs by the beginning of (6)
   Bus slave asserts /STERM
   Bus master detects /STERM low at the end of (6)
   Data symbols for recipient FCRs present on M-Data lines
   Data symbols written into SRAM
   Address register cleared at the end of (6)
7. Data symbols discarded from the end of FIFOs
   Bus master negates /BCQUAL, data, SPC1-0, and SIZ1-0 by the end of (7)

Class II Read

A class II read occurs if the /WRITE signal is not asserted and the function codes (FC)
indicate an access to the user-defined reserved space (FC=3).

1. Bus master asserts /ALERT
   Bus master presents address on address/data lines
   Bus slave latches address, SPC0-1, SIZ1-0, and /WRITE at the end of (1)
   SPC signals indicate a Class II cycle
2. Bus master asserts /BCQUAL, qualifying read
   Bus master releases address/data lines
   Data symbols read out of SRAM
   Address register cleared at the end of (2)
3. Data symbols broadcast to other FCRs
4.
5.
6. Data symbols from other FCRs appear at FIFOs by the beginning of (6)
   Symbol from source FCR selected for reflection
   Source symbol re-broadcast by recipient FCRs; source FCR sends null data
   Data symbols from all FCRs loaded into Class II delay register
7. Data symbols discarded from the end of FIFOs
8.
9.
10. Data symbols from recipient FCRs appear at FIFOs by the beginning of (10)
    Bus slave asserts /STERM
    Bus master detects /STERM low at the end of (10)
    Syndrome latch is enabled
11. Bus slave drives decoded data onto address/data lines
    Bus master latches data at the end of (11)
    Data symbols discarded from the end of FIFOs
    Bus master negates /BCQUAL, SPC1-0, and SIZ1-0 by the end of (11)
Cache Burst Access (read)

A cache burst is a special form of the class I read. A cache burst occurs if the /WRITE signal is not asserted, the function codes (FC) indicate an access to user program, user data, supervisor program, or supervisor data space, and the cache burst request (/CBREQ) signal is asserted.

1. Bus master asserts /ALERT
   Bus master presents address on address/data lines
   Bus slave latches address, SPC0-1, SIZ1-0, and /WRITE at the end of (1)
2. Bus master asserts /BCQUAL, qualifying read
   Bus master releases address/data lines
   Data symbols for Word 1 read out of SRAM
   Address register incremented at the end of (2)
3. Data symbols for Word 1 broadcast to other FCRs
   Data symbols for Word 2 read out of SRAM
   Address register incremented at the end of (3)
4. Data symbols for Word 2 broadcast to other FCRs
   Data symbols for Word 3 read out of SRAM
   Address register incremented at the end of (4)
5. Data symbols for Word 3 broadcast to other FCRs
   Data symbols for Word 4 read out of SRAM
   Address register cleared at the end of (5)
6. Data symbols for Word 4 broadcast to other FCRs
   Data symbols for Word 1 appear at FIFOs by the beginning of (6)
   Bus slave asserts /STERM
   Bus master detects /STERM low at the end of (6)
   Syndrome latch is enabled
7. Bus slave drives Word 1 onto address/data lines
   Bus master latches data at the end of (7)
   Data symbols for Word 1 discarded from the end of FIFOs
   Data symbols for Word 2 appear in FIFOs by the beginning of (7)
   Bus slave asserts /STERM
   Bus master detects /STERM low at the end of (7)
   Syndrome latch is enabled
8. Bus slave drives Word 2 onto address/data lines
   Bus master latches data at the end of (8)
   Data symbols for Word 2 discarded from the end of FIFOs
   Data symbols for Word 3 appear in FIFOs by the beginning of (8)
   Bus slave asserts /STERM
   Bus master detects /STERM low at the end of (8)
   Syndrome latch is enabled
9. Bus slave drives Word 3 onto address/data lines
   Bus master latches data at the end of (9)
   Data symbols for Word 3 discarded from the end of FIFOs
   Data symbols for Word 4 appear in FIFOs by the beginning of (9)
   Bus slave asserts /STERM
   Bus master detects /STERM low at the end of (9)
   Syndrome latch is enabled
10. Bus slave drives Word 4 onto address/data lines
    Bus master latches data at the end of (10)
    Data symbols for Word 4 discarded from the end of FIFOs
    Bus master negates /BCQUAL, SPC1-0, and SIZ1-0 by the end of (10)
Aborted Access (no /AS)

An aborted access occurs if the 68030 starts a memory cycle by the assertion of the /ALERT signal but fails to qualify the cycle by asserting /BCQUAL during the next clock period. An aborted access is the result of a hit in the 68030 internal caches.

1. Bus master asserts /ALERT
   Bus master presents address on address/data lines
   Bus slave latches address, SPC0-1, SIZ1-0, and /WRITE at the end of (1)
2. Bus master does not assert /BCQUAL, cycle is not qualified
   Bus master releases address/data lines
   Address register cleared at the end of (2)

Split Cache Burst Access

A split cache burst occurs if the FTC goes low before all four longwords of a cache burst cycle have been transmitted.

1. Bus master asserts /ALERT
   Bus master presents address on address/data lines
   Bus slave latches address, SPC0-1, SIZ1-0, and /WRITE at the end of (1)
2. Bus master asserts /BCQUAL, qualifying read
   Bus master releases address/data lines
   Data symbols for Word 1 read out of SRAM
   Address register incremented at the end of (2)
3. Data symbols for Word 1 broadcast to other FCRs
   Data symbols for Word 2 read out of SRAM
   Address register incremented at the end of (3)
4. FTC is low; data symbols for Word 2 can’t be broadcast until FTC returns high
   Data symbols for Word 3 read out of SRAM
5. FTC is still low; FTSM transmitter is on hold
6. Data symbols for Word 2 broadcast to other FCRs
   Data symbols for Word 3 read out of SRAM
   Address register incremented at the end of (6)
   Data symbols for Word 1 appear at FIFOs by the beginning of (6)
   Bus slave asserts /STERM
   Bus master detects /STERM low at the end of (6)
   Syndrome latch is enabled
7. Data symbols for Word 3 broadcast to other FCRs
   Data symbols for Word 4 read out of SRAM
   Address register cleared at the end of (7)
   Bus slave drives Word 1 onto address/data lines
   Bus master latches data at the end of (7)
   Data symbols for Word 1 discarded from the end of FIFOs
8. Data symbols for Word 4 broadcast to other FCRs
9. Data symbols for Word 2 appear in FIFOs by the beginning of (9)
   Bus slave asserts /STERM
   Bus master detects /STERM low at the end of (9)
   Syndrome latch is enabled
10. Bus slave drives Word 2 onto address/data lines
    Bus master latches data at the end of (10)
    Data symbols for Word 2 discarded from the end of FIFOs
    Data symbols for Word 3 appear in FIFOs by the beginning of (10)
    Bus slave asserts /STERM
Bus master detects /STERM low at the end of (10) 
Syndrome latch is enabled 

11. Bus slave drives Word 3 onto address/data lines 
Bus master latches data at the end of (11) 
Data symbols for Word 3 discarded from the end of FIFOs 
Data symbols for Word 4 appear in FIFOs by the beginning of (11) 
Bus slave asserts /STERM 
Bus master detects /STERM low at the end of (11) 
Syndrome latch is enabled 

12. Bus slave drives Word 4 onto address/data lines 
Bus master latches data at the end of (12) 
Data symbols for Word 4 discarded from the end of FIFOs 
Bus master negates /BCQUAL, SPC1-0, and SIZ1-0 by the end of (12) 

Block Scrub Cycle 

The scrub cycle is used to refresh four longwords of memory in one indivisible cycle. 

1. Controller starts scrub cycle 
/ BUSY signal asserted by bus slave to signal bus master that a bus cycle can not be started 
Address register loaded with next scrub address at the end of (1) 

2. Data symbols for Word 1 read out of SRAM 
Address register incremented at the end of (2) 

3. Data symbols for Word 1 broadcast to other FCRs 
Data symbols for Word 2 read out of SRAM 
Address register incremented at the end of (3) 

4. Data symbols for Word 2 broadcast to other FCRs 
Data symbols for Word 3 read out of SRAM 
Address register incremented at the end of (4) 

5. Data symbols for Word 3 broadcast to other FCRs 
Data symbols for Word 4 read out of SRAM 
Address register incremented at the end of (5); AR now points to Word 1 

6. Data symbols for Word 4 broadcast to other FCRs 
Data symbols for Word 1 appear at FIFOs by the beginning of (6) 
Decoded Word 1 appears at end of decoder at the end of (6) 
Syndrome latch is enabled 

7. Data symbols for Word 1 discarded from the end of FIFOs 
Encoded Word 1 appears at end of encoder at the end of (7) 
Data symbols for Word 2 appear in FIFOs by the beginning of (7) 
Decoded Word 2 appears at end of decoder at the end of (7) 
Syndrome latch is enabled 

8. Data symbols for Word 1 written to SRAM 
Data symbols for Word 2 discarded from the end of FIFOs 
Encoded Word 2 appears at end of encoder at the end of (8) 
Data symbols for Word 3 appear in FIFOs by the beginning of (8) 
Decoded Word 3 appears at end of decoder at the end of (8) 
Syndrome latch is enabled 
Address register incremented at the end of (8) 

9. Data symbols for Word 2 written to SRAM 
Data symbols for Word 3 discarded from the end of FIFOs 
Encoded Word 3 appears at end of encoder at the end of (9) 
Data symbols for Word 4 appear in FIFOs by the beginning of (9) 
Decoded Word 2 appears at end of decoder at the end of (9) 

114
Syndrome latch is enabled
Address register incremented at the end of (9)

10. Data symbols for Word 3 written to SRAM
    Data symbols for Word 4 discarded from the end of FIFOs
    Encoded Word 4 appears at end of encoder at the end of (10)
    Address register incremented at the end of (10)

11. Data symbols for Word 4 written to SRAM
    Address register cleared at the end of (11)
    /BUSY signal deasserted by bus slave to signal bus master that a bus cycle can be started
AC Electrical Characteristics:

The following parameters must be determined during the detailed design phase of the FTP/FTSM. Since most of the parameters are implementation dependent, it would be useless to determine them before component selection. Some of the parameters are demonstrated on the figure below.

1. /ALERT setup time
2. /ALERT hold time
3. /BCQUAL setup time
   /STERM high to /BCQUAL high
4. Address setup time
5. Address hold time
6. Data setup time
7. Data hold time
   My-DATA setup time (for data transfer from PC to FTSM)
   My-DATA hold time (for data transfer from PC to FTSM)
   Clk to Data
8. Clk to My-ADDR
9. Clk to My-DATA (for data transfer from FTSM to PC)
10. Clk to /STERM low
11. Clk to /STERM high
   Memory access (read) time
   Memory access (write) time

[Diagram of timing chart with annotations 1 to 13]
**Bus Pin Assignments**

Shown below are the tentative pin assignments for the synchronous bus. The synchronous bus is implemented on a 64-pin backplane using a 3-column, 96-pin Eurocard connector. The middle row of pins, except for power and ground connections, is reserved for the 32-bit extension to the VMEbus. Signals preceded by a slash (/) are asserted low. All other signals are asserted high.

<table>
<thead>
<tr>
<th>Pin</th>
<th>Row A</th>
<th>Row B</th>
<th>Row C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A/D00</td>
<td>+5v</td>
<td>A/D01</td>
</tr>
<tr>
<td>2</td>
<td>A/D02</td>
<td>GND</td>
<td>A/D03</td>
</tr>
<tr>
<td>3</td>
<td>A/D04</td>
<td>Reserved (VME)</td>
<td>A/D05</td>
</tr>
<tr>
<td>4</td>
<td>A/D06</td>
<td>Reserved (VME)</td>
<td>A/D07</td>
</tr>
<tr>
<td>5</td>
<td>A/D08</td>
<td>Reserved (VME)</td>
<td>A/D09</td>
</tr>
<tr>
<td>6</td>
<td>A/D10</td>
<td>Reserved (VME)</td>
<td>A/D11</td>
</tr>
<tr>
<td>7</td>
<td>A/D12</td>
<td>Reserved (VME)</td>
<td>A/D13</td>
</tr>
<tr>
<td>8</td>
<td>A/D14</td>
<td>Reserved (VME)</td>
<td>A/D15</td>
</tr>
<tr>
<td>9</td>
<td>A/D16</td>
<td>Reserved (VME)</td>
<td>A/D17</td>
</tr>
<tr>
<td>10</td>
<td>A/D18</td>
<td>Reserved (VME)</td>
<td>A/D19</td>
</tr>
<tr>
<td>11</td>
<td>A/D20</td>
<td>Reserved (VME)</td>
<td>A/D21</td>
</tr>
<tr>
<td>12</td>
<td>A/D22</td>
<td>GND</td>
<td>A/D23</td>
</tr>
<tr>
<td>13</td>
<td>A/D24</td>
<td>+5v</td>
<td>A/D25</td>
</tr>
<tr>
<td>14</td>
<td>A/D26</td>
<td>Reserved (VME)</td>
<td>A/D27</td>
</tr>
<tr>
<td>15</td>
<td>A/D28</td>
<td>Reserved (VME)</td>
<td>A/D29</td>
</tr>
<tr>
<td>16</td>
<td>A/D30</td>
<td>Reserved (VME)</td>
<td>A/D31</td>
</tr>
<tr>
<td>17</td>
<td>GND</td>
<td>Reserved (VME)</td>
<td>GND</td>
</tr>
<tr>
<td>18</td>
<td>Reserved</td>
<td>Reserved (VME)</td>
<td>GND</td>
</tr>
<tr>
<td>19</td>
<td>Reserved</td>
<td>Reserved (VME)</td>
<td>GND</td>
</tr>
<tr>
<td>20</td>
<td>/WRITE</td>
<td>Reserved (VME)</td>
<td>GND</td>
</tr>
<tr>
<td>21</td>
<td>SPACE0</td>
<td>Reserved (VME)</td>
<td>SIZE0</td>
</tr>
<tr>
<td>22</td>
<td>SPACE1</td>
<td>GND</td>
<td>/ALERT</td>
</tr>
<tr>
<td>23</td>
<td>Reserved</td>
<td>Reserved (VME)</td>
<td>SIZE1</td>
</tr>
<tr>
<td>24</td>
<td>Reserved</td>
<td>Reserved (VME)</td>
<td>GND</td>
</tr>
<tr>
<td>25</td>
<td>GND</td>
<td>Reserved (VME)</td>
<td>/STERM</td>
</tr>
<tr>
<td>26</td>
<td>GND</td>
<td>Reserved (VME)</td>
<td>PROCCLK</td>
</tr>
<tr>
<td>27</td>
<td>GND</td>
<td>Reserved (VME)</td>
<td>/BCQUAL</td>
</tr>
<tr>
<td>28</td>
<td>Unused</td>
<td>Reserved (VME)</td>
<td>Reserved</td>
</tr>
<tr>
<td>29</td>
<td>Unused</td>
<td>Reserved (VME)</td>
<td>/CBREQ</td>
</tr>
<tr>
<td>30</td>
<td>Unused</td>
<td>Reserved (VME)</td>
<td>/WAIT</td>
</tr>
<tr>
<td>31</td>
<td>Reserved</td>
<td>GND</td>
<td>/BUSY</td>
</tr>
<tr>
<td>32</td>
<td>Reserved</td>
<td>+5v</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
9.3. Controller Microcode

Access to the FTSM from the synchronous bus is controlled by two finite-state machine controllers. The pseudo-high-level code for these controllers is given below:

FSM A:

/* inputs */
/ALERT-alert signal from synchronous bus (asserted low)
/WRITE-write signal from SBUS
SPACE1-Defines address space for current access
SPACE0-
/CBREQ-Cache burst request
/BCQUAL-Bus cycle qualify
SCRUBOK-scrub cycle approved by scrub regulator
SCRUBEN-scrubbing enabled by processor (from proc. control reg)
FTC-Fault-tolerant clock
DR-Data ready (from FSM B)

/* outputs */
SOURCE-selects proc. (0) or scrubber (1) for loading addr. reg.
/STEM-synchronous termination signal for SBUS
MYBUS-enables data to be driven onto FTSM bus from encoder
LOADAR-loads address register
CLEARAR-clears address register
INCAR-increments address register (for cache burst or scrub)
REGEN-enables IFC controller input register
MUX2-selects data from encoder (rather than MYBUS) to broadcast
RD-Return data, signals FSM B that data is to be returned for
   further processing (scrub cycle or Class IIs)
RTOW-changes the address register from read mode to write mode
INCSCR-Increment scrubber to next scrub block

IDLE:

/* Class I write */

if (ALERT & WRITE & /SPACE1 & /SPACE0)
{
    SOURCE=0;
    :STEM;
    :MYBUS;
    CLEARAR;
    goto IDLE;
}

/* Class I read */

else if (ALERT & WRITE & /SPACE1 & /SPACE0)
{
    SOURCE=0;
    :if (/BCQUAL)
{  CLEARAR;
    goto IDLE;
}

/* Check for cache burst request */

else if (BCQUAL & CACHE)
{
    REGEN;
    INCAR;
:    while (/FTC);
    INCAR;
    REGEN;
:    while (/FTC);
    INCAR;
    REGEN;
:    while (/FTC);

/* append scrub cycle if approved by regulator */

/* single longword access */

else if (BCQUAL & /CACHE)
{
    REGEN;

/* append scrub cycle if approved */

    if (SCRUBEN & SCRUBOK)
    {
        LOADAR;
        REGEN;
        SOURCE=1;
        goto X;
    }
    else
    {
        CLEARAR;
        REGEN;
        goto IDLE;
    }
}

/* single longword access */

else if (BCQUAL & /CACHE)
{
    REGEN;

/* append scrub cycle if approved */

    if (SCRUBEN & SCRUBOK)
    {
        LOADAR;
        SOURCE=1;
        goto X;
    }
    else
    {
        CLEARAR;
        goto IDLE;
    }
}
} }

} }

/* Class II write cycle */
else if (ALERT & WRITE & /SPACE1 & SPACE0)
{
  SOURCE=0;
  : REGEN;
  MUX2;
  RD;
  : while (/DR);
  MYBUS;
  CLEARAR;
  goto IDLE;
}

/* Class II read cycle */
else if (ALERT & /WRITE & /SPACE1 & SPACE0)
{
  SOURCE=0;
  : REGEN;
  CLEARAR;
  RD;
  : while (/DR);
  MYBUS;
  REGEN;
  goto IDLE;
} else
{
  stay;
}

/* Scrub routine */

X:  WAIT;
    while (/FTC);
    : INCAR;
    REGEN;
    RD;
    : WAIT;
    while (/FTC);
    INCAR;
    REGEN;
    RD;
    : WAIT;
    while (/FTC);
    INCAR;
    REGEN;
    RD;
    : WAIT;
    while (/FTC);
    INCAR;
    REGEN;
    RD;
    : WAIT;
while (/FTC);
INCAR;
REGEN;
RTOW;
RD;
: WAIT;
while (/DR);
INCAR;
MYBUS;
: WAIT;
while (/DR);
INCAR;
MYBUS;
: WAIT;
while (/DR);
INCAR;
MYBUS;
: WAIT;
while (/DR);
CLEARAR;
INCSR;
MYBUS;
goto IDLE;

FSM B:

/* Inputs */
DOINGSCRUB—Signal from FSM A indicating scrub in progress
DOINGSCRUB=WAIT
DAV—Data Valid. Signal from IFC controller indicating data is ready at the FIFO outputs

/* Outputs */
DR0—Data ready signal that will reach FSM A this cycle
DR2—Data ready signal delayed 2 cycles before reaching FSM A
Note: DR signal into FSM A is the OR of the above
CLASS2—Selects special class II hardware
LOADC2—loads class II delay registers from FIFO outputs
LATCHSYN—latch syndrome (if non-zero)
SHIFTOUT—remove data at the end of FIFOs
SELIBUS—

IDLE: if (DAV & /SPACE1 & /SPACE0 & /WRITE)
{
    STERM;
    LATCHSYN;
    : SHIFTOUT;
PBUS;
while (DAV & /SPACE1 & /SPACE0 & /WRITE)
{
    STERM;
    LATCHSYN;
    stay;
}
if (/DAV | SPACE1 | SPACE0 | WRITE) goto IDLE;
else if (DR & DOINGSCRUB & Rddly)
{
    LATCHSYN;
    DR2;
    SELIBUS;
    Y:  SHIFTOUT;
    while (DAV & DOINGSCRUB & Rddly)
    {
        LATCHSYN;
        DR2;
        SELIBUS;
    }
    goto IDLE;
}
else if (DAV & DOINGSCRUB & Rddly)
{
    LATCHSYN;
    DR2;
    SELIBUS;
    goto Y;
}

/* Special routine for Class II read */
else if (DAV & /SPACE1 & SPACE0 & /WRITE)
{
    DR0;
    LOADC2;
    CLASS2;
:  SHIFTOUT;
:  while (/DR);
    CLASS2;
    STREAM;
    LATCHSYN;
:  SHIFTOUT;
    PBUS;
    goto IDLE;
}

/* Special routine for Class II write */
else if (DAV & /SPACE1 & SPACE0 & WRITE)
{
    DR0;
    CLASS2;
    STREAM;
:  SHIFTOUT;
    goto IDLE;
}
9.4. Memory Controller/Address Register Logic

This section details the logic design of the memory controller and address register for the FTSM.

Address Decoder

The address decoder determines, from the low bits of the address, the SIZE signals, and the /WRITE signal, which bytes of a specific longword are being accessed. The address decoder also contains the logic to directly control the memory modules.

/* Select signals */
/* LLB = lower-lower byte (lowest significant byte) */
/* LMB = lower-middle byte */
/* UMB = upper-middle byte */
/* UUB = upper-upper byte (most significant byte) */

LLB = A1*A0 + A0*SIZE1*SIZE0 + /SIZE1*/SIZE0 + A1*SIZE1
LMB = A1*/A0 + /A1*/SIZE1*/SIZE0 + /A1*SIZE1*SIZE0 +
     /A1*/A0*/SIZE0
UMB = /A1*A0 + /A1*/SIZE0 + /A1*SIZE1
UUB = /A1*/A0

/* Chip select signals. One for each byte in a longword */
/* by one for each of four memory modules. Total:16 */

CSLLB0 := (LLB + /WRITE) *MEMSEL*/A21*/A20*LOADAR*/CLEARAR +
         CSLLB0*/CLEARAR
CSLMB0 := (LMB + /WRITE) *MEMSEL*/A21*/A20*LOADAR*/CLEARAR +
         CSLMB0*/CLEARAR
CSUMB0 := (UMB + /WRITE) *MEMSEL*/A21*/A20*LOADAR*/CLEARAR +
         CSUMB0*/CLEARAR
CSUUB0 := (UUB + /WRITE) *MEMSEL*/A21*/A20*LOADAR*/CLEARAR +
         CSUUB0*/CLEARAR

CSLLB1 := (LLB + /WRITE) *MEMSEL*/A21*A20*LOADAR*/CLEARAR +
         CSLLB1*/CLEARAR
CSLMB1 := (LMB + /WRITE) *MEMSEL*/A21*A20*LOADAR*/CLEARAR +
         CSLMB1*/CLEARAR
CSUMB1 := (UMB + /WRITE) *MEMSEL*/A21*A20*LOADAR*/CLEARAR +
         CSUMB1*/CLEARAR
CSUUB1 := (UUB + /WRITE) *MEMSEL*/A21*A20*LOADAR*/CLEARAR +
         CSUUB1*/CLEARAR

CSLLB2 := (LLB + /WRITE) *MEMSEL*A21*/A20*LOADAR*/CLEARAR +
         CSLLB2*/CLEARAR
CSLMB2 := (LMB + /WRITE) *MEMSEL*A21*/A20*LOADAR*/CLEARAR +
         CSLMB2*/CLEARAR
CSUMB2 := (UMB + /WRITE) *MEMSEL*A21*/A20*LOADAR*/CLEARAR +
         CSUMB2*/CLEARAR
CSUUB2 := (UUB + /WRITE) *MEMSEL*A21*/A20*LOADAR*/CLEARAR +
         CSUUB2*/CLEARAR

130
CSLLB3 := (LLB + /WRITE) *MEMSEL*\texttt{A21*A20*LOADAR*/CLEARAR} + CSLLB3*/CLEARAR
CSLMB3 := (LMB + /WRITE) *MEMSEL*\texttt{A21*A20*LOADAR*/CLEARAR} + CSLMB3*/CLEARAR
CSUMB3 := (UMB + /WRITE) *MEMSEL*\texttt{A21*A20*LOADAR*/CLEARAR} + CSUMB3*/CLEARAR
CSUUB3 := (UUB + /WRITE) *MEMSEL*\texttt{A21*A20*LOADAR*/CLEARAR} + CSUUB3*/CLEARAR

/* Write Enable signals. One for each of four memory modules */

WE0 := WRITE*MEMSEL*/\texttt{A21*/A20*LOADAR*/CLEARAR} + WE0*/CLEARAR
WE1 := WRITE*MEMSEL*/\texttt{A21*A20*LOADAR*/CLEARAR} + WE1*/CLEARAR
WE2 := WRITE*MEMSEL*\texttt{A21*/A20*LOADAR*/CLEARAR} + WE2*/CLEARAR
WE3 := WRITE*MEMSEL*\texttt{A21*A20*LOADAR*/CLEARAR} + WE3*/CLEARAR

Address Register

The address register simply latches the low order bit of the address (A19-A2) to provide stable address signals to the memory modules. In addition, the lower two bits (A3-A2) can be incremented modulo 4 to support cache burst accesses and memory scrub cycles.

/* Ax represents incoming address lines from S-BUS */
/* AMx represents address lines to memory modules */

AM0 := AM0*/LOADAR*/INCAR + \texttt{A2*LOADAR} + \texttt{AM0*/LOADAR*INCAR}
AM1 := AM1*/LOADAR*/INCAR + \texttt{A3*LOADAR} + (AM1*/AM0 + \texttt{AM1*AM0})*LOADAR*INCAR
AM2 := A4
AM3 := A5
AM4 := A6
AM5 := \texttt{A7}
AM6 := A8
AM7 := A9
AM8 := A10
AM9 := A11
AM10 := A12
AM11 := A13
AM12 := A14
AM13 := A15
AM14 := A16
AM15 := A17
AM16 := A18
AM17 := A19

Memory Scrubber

The memory scrubber is designed to step through memory starting with the lowest address in the FTSM (0xC000000), incrementing by 4 longword locations everytime INCSCR is asserted. The scrubber resets to zero when it reaches address 0xFFF000, so that the region reserved for class II exchanges is not scrubbed.
9.5. Fault-Tolerant Clock

Fault-Tolerant Clock Block Diagram
FTC Sequencer
Clocked by 50MHz clock (GENCLK)
Processor Clock (PROCCLK) Generator
Clocked by 50MHz clock (GENCLK)

Local Fault-Tolerant Clock (MYFTC) Generator
Clocked by 50 MHz clock (GENCLK)
Phase Detector
Clocked by 50MHz clock (GENCLK)
FTC Edge Voter
9.6. Analysis Procedures and Parameters

The reliability and performance analyses presented in section 6 depend on a number of parameters. Each analysis procedure is briefly presented here along with all important parameters pertaining to the analysis.

Note: all memory sizes are for the apparent memory size, not the total amount of memory required. For example, a quadruplicated memory system with 1 Megabyte in each channel, for a total of 4 Megabytes, would be shown as a 1 Mbyte system.

Combinatorial Reliability Analysis

section 6.1.1
Compares the reliability of three different types of error-correcting memory systems. Probability of system failure is a function of bit reliability (relations presented in section 6.1.1). Bit reliability is a function of time (also presented in 6.1.1).

Relevant parameters:
Memory size $K=1$ Mbyte
Data word width $n=8$
Number of check bits for ECC $m=5$
Number of bits per symbol for FTSM $s=4$
Bit failure rate $\lambda_b=2.3 \times 10^{-9}$/hour

Markov Analysis of FTSM vs. Quadruplicated Memory

section 6.1.3.1
Compares the dynamic reliability of using a (4,2) code for informational redundancy in the FTSM as compared to using quadruplicated copies. The Markov model presented in section 6.1.2 is solved by an iterative technique using a FORTRAN computer program on a VAX computer.

Relevant parameters:
Memory size $K=4$ Mbytes
Transient bit failure rate $\lambda_{bt}=2.3 \times 10^{-9}$/hour
Permanent bit failure rate $\lambda_{bp}=2.3 \times 10^{-11}$/hour
Memory access rate for writes $\mu_{aw}=3.6 \times 10^5$/hour
Memory access rate for reads $\mu_{ar}=3.6 \times 10^6$/hour
Memory scrub rate $\mu_{scr}=360$/hour
Bits per symbol (FTSM) $n=4$
Bits per copy (Quad) $n=8$

Data:

<table>
<thead>
<tr>
<th>time</th>
<th>FTSM trans</th>
<th>FTSM perm</th>
<th>QUAD trans</th>
<th>QUAD perm</th>
</tr>
</thead>
<tbody>
<tr>
<td>2e+0</td>
<td>2.05e-14</td>
<td>7.468e-11</td>
<td>8.199e-14</td>
<td>2.987e-10</td>
</tr>
<tr>
<td>1e+1</td>
<td>1.025e-13</td>
<td>1.867e-9</td>
<td>4.1e-13</td>
<td>7.468e-9</td>
</tr>
<tr>
<td>1e+2</td>
<td>1.025e-12</td>
<td>1.867e-7</td>
<td>4.1e-12</td>
<td>7.468e-7</td>
</tr>
<tr>
<td>1e+3</td>
<td>1.025e-11</td>
<td>1.867e-5</td>
<td>4.1e-11</td>
<td>7.468e-5</td>
</tr>
<tr>
<td>1e+4</td>
<td>1.025e-10</td>
<td>1.865e-3</td>
<td>4.1e-10</td>
<td>7.439e-3</td>
</tr>
</tbody>
</table>
Markov Analysis of Reliability vs. Memory Size

section 6.1.3.2
Uses the Markov model to determine the relation between memory size and reliability in the FTSM. Transient and permanent errors are considered separately. The system loss probability is determined for mission times of 100 and 10,000 hours.

Relevant parameters:
Memory size
Transient bit failure rate
Permanent bit failure rate
Memory access rate for writes
Memory access rate for reads
Memory scrub rate
Bits per symbol

K=4 Mbytes
$\lambda_{bt}=2.3 \times 10^{-9}/$hour
$\lambda_{bp}=2.3 \times 10^{-11}/$hour
$\mu_{aw}=3.6 \times 10^{-2}$/hour
$\mu_{ar}=3.6 \times 10^{-1}$/hour
$\mu_{sc} = 360$/hour
n=4

Data:

<table>
<thead>
<tr>
<th>read rate</th>
<th>write rate</th>
<th>scrub rate</th>
<th>mem size</th>
<th>trans @100</th>
<th>perm @100</th>
<th>trans @10k</th>
<th>perm @10k</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.36</td>
<td>0.036</td>
<td>360</td>
<td>4e+6</td>
<td>9.999e-10</td>
<td>1.775e-7</td>
<td>1.026e-7</td>
<td>1.864e-3</td>
</tr>
<tr>
<td>0.36</td>
<td>0.036</td>
<td>360</td>
<td>2e+6</td>
<td>5e-10</td>
<td>8.875e-8</td>
<td>5.128e-8</td>
<td>9.325e-4</td>
</tr>
<tr>
<td>0.36</td>
<td>0.036</td>
<td>360</td>
<td>1e+6</td>
<td>2.5e-10</td>
<td>4.438e-8</td>
<td>2.564e-8</td>
<td>4.664e-4</td>
</tr>
<tr>
<td>0.36</td>
<td>0.036</td>
<td>360</td>
<td>5e+5</td>
<td>1.25e-10</td>
<td>2.219e-8</td>
<td>1.282e-8</td>
<td>2.332e-4</td>
</tr>
<tr>
<td>0.36</td>
<td>0.036</td>
<td>360</td>
<td>2e+5</td>
<td>4.999e-11</td>
<td>8.875e-9</td>
<td>5.128e-9</td>
<td>9.329e-5</td>
</tr>
<tr>
<td>0.36</td>
<td>0.036</td>
<td>360</td>
<td>1e+5</td>
<td>2.5e-11</td>
<td>4.438e-9</td>
<td>2.564e-9</td>
<td>4.665e-5</td>
</tr>
<tr>
<td>0.36</td>
<td>0.036</td>
<td>360</td>
<td>5e+4</td>
<td>1.25e-11</td>
<td>2.219e-9</td>
<td>1.282e-9</td>
<td>2.332e-5</td>
</tr>
<tr>
<td>0.36</td>
<td>0.036</td>
<td>360</td>
<td>2e+4</td>
<td>4.999e-12</td>
<td>8.875e-10</td>
<td>5.128e-10</td>
<td>9.329e-6</td>
</tr>
</tbody>
</table>

Markov Analysis of Reliability vs. % Fast Memory

section 6.1.3.3
Uses the Markov model to demonstrate the dominant effect that rarely accessed locations have on memory system reliability. Only system loss due to transients is analyzed.

Relevant parameters:
Memory size
Transient bit failure rate
Permanent bit failure rate
Memory access rate for writes (fast mem)
Memory access rate for reads (fast mem)
Memory access rate for writes (slow mem)
Memory access rate for reads (slow mem)
Memory scrub rate
Bits per symbol

K=4 Mbytes
$\lambda_{bt}=2.3 \times 10^{-9}/$hour
$\lambda_{bp}=2.3 \times 10^{-11}/$hour
$\mu_{aw}=3.6 \times 10^{5}$/hour
$\mu_{ar}=3.6 \times 10^{6}$/hour
$\mu_{aw}=3.6 \times 10^{-2}$/hour
$\mu_{ar}=3.6 \times 10^{-1}$/hour
$\mu_{sc} = 360$/hour
n=4
Markov Analysis of Reliability vs. Scrub Rate

section 6.1.3.2
Uses the Markov model to determine the optimum memory scrub rate. Low access rates are used since the previous analysis indicates that memory locations with high access rates will not contribute significantly to the probability of system loss. The system loss probability is determined for mission times of 10, 100 and 10,000 hours.

**Relevant parameters:**
- Memory size: \( K = 4 \) Mbytes
- Transient bit failure rate: \( \lambda_{bt} = 2.3 \times 10^{-9} \) hour
- Permanent bit failure rate: \( \lambda_{bp} = 2.3 \times 10^{-11} \) hour
- Memory access rate for writes: \( \mu_{aw} = 3.6 \times 10^{-3} \) hour
- Memory access rate for reads: \( \mu_{ar} = 3.6 \times 10^{-2} \) hour
- Memory scrub rate: \( \mu_{sc} = 360 \) hour
- Bits per symbol: \( n = 4 \)

Performance Analysis: Overhead vs. Iteration Rate

section 6.2.5
Presents the traditional measure of performance: overhead imposed by fault-tolerance as a percentage of total processor throughput. Analysis is done for three different FTP designs. Iteration rate is determined by finding the inverse of the total task completion time for each design. Note: all exchange times given are for 240 bytes.

**Relevant parameters:**
- Clock Frequency of Baseline design: \( F_{\text{baseline}} = 12.5 \) MHz
- Clock Frequency of NEFTP: \( F_{\text{NEFTP}} = 12.5 \) MHz
- Clock Frequency of VLSI-FTP: \( F_{\text{VLSI-FTP}} = 16 \) MHz
- Clock Frequency of FTP/FTSM: \( F_{\text{FTP/FTSM}} = 25 \) MHz
- Class I on NEFTP: \( 464 \times 10^{-6} \) sec
- Class II on NEFTP: \( 426 \times 10^{-6} \) sec
- Synchronization exchange on NEFTP: \( 21 \times 10^{-6} \) sec
- Class I on VLSI-FTP: \( 240 \times 10^{-6} \) sec
- Class II on VLSI-FTP: \( 240 \times 10^{-6} \) sec
- Synchronization factor on VLSI-FTP: \( 1/8 \)
- Class I on FTP/FTSM: \( 0 \)
- Class II on FTP/FTSM: \( 40.8 \times 10^{-6} \) sec
- Synchronization on FTP/FTSM: \( \sim 0 \)
- Minimum bus cycles for 68030: \( n_m = 3 \)
- Number of cycles for internal caches: \( n_c = 1 \)
- Penalty (in cycles) imposed by FTSM: \( p = 4 \)
- Fraction of cycles involved in mem reads: \( r_0 = 0.3 \)
- Cache hit ratio: \( q = 0.95 \)
Performance Analysis: Task Completion Ratios

section 6.2.5
Presents a uniform performance metric which can be used to directly compare competing designs. The task completion ratio compares the time to complete a specific task on a specific design to the time to complete the same task on a baseline design.

Relevant parameters:
same as above
9.7. Glossary

BGU-Bus Guardian Unit. A device which attempts to prevent a faulty processor from seizing control of a shared bus and thereby causing catastrophic failure.

Byzantine resilience-A design methodology for fault-tolerant computers in which no assumptions are made about the possible failure modes.

CP-Computational Processor. In the FTP/FTSM, the CP is connected directly to the FTSM system. Each CP in the FTP/FTSM is micro-synchronized

fault-masking function-A function which resolves an informationally redundant data set into one copy of the original data, given that a non-correctable error pattern is not present in the input data set.

FCR-Fault Containment Region. A physical circuit boundary with protection at all points of entry from other FCRs to prevent fault propagation between FCRs.

Galois field-A set whose subsets are a set with a finite number of numerical elements, an additive operation and a multiplicative operation.

informational redundancy-A method of storing or transmitting data such that the informational data can be recovered in the presence of some types of errors. Examples: triplicated copies and error-correcting codes.

IOP-Input/Output Processor. Processor responsible for I/O operations. The IOP in the FTP/FTSM is connected to the CP through a dual-port memory

longword-By Motorola convention, a data object composed of 32 bits. (see word)


word-A generic reference to a data object of arbitrary size. By Motorola convention, a word refers to an object composed of 16 bits (2 bytes). Other conventions use word to refer to an object of 32 bits.
9.8. References


20 Krol, 1982.


35 Dzwonczyk, et. al., p. 83, 1989.


42 Abler, p. 86-88,90-92.


