NASA Contractor Report-194924

(NASA-CR-194924) ADVANCED INFORMATION PROCESSING SYSTEM: THE ARMY FAULT-TOLERANT ARCHITECTURE DETAILED DESIGN OVERVIEW Final Report (Draper (Charles Stark) Lab.) 138 p

N94-36962

Unclas

G3/62 0013737

Advanced Information Processing System: The Army Fault Tolerant Architecture Detailed Design Overview

Richard E. Harper, Carol A. Babikyan, Bryan P. Butler, Robert J. Clasen, Chris H. Harris, Jaynarayan H. Lala, Thomas K. Masotto, Gail A. Nagle, Mark J. Prizant, Steven Treadwell
THE CHARLES STARK DRAPER LABORATORY, INC., CAMBRIDGE, MA 02139

Contract NAS1-18565
June 1994

**National Aeronautics and
Space Administration
Langley Research Center
Hampton, Virginia 23681-0001**

NASA Contractor Report-194924

Advanced Information Processing System: The Army Fault Tolerant Architecture Detailed Design Overview

Richard E. Harper, Carol A. Babikyan, Bryan P. Butler, Robert J. Clasen, Chris H. Harris, Jaynarayan H. Lala, Thomas K. Masotto, Gail A. Nagle, Mark J. Prizant, Steven Treadwell
THE CHARLES STARK DRAPER LABORATORY, INC., CAMBRIDGE, MA 02139

Contract NAS1-18565
June 1994

National Aeronautics and
Space Administration
Langley Research Center
Hampton, Virginia 23681-0001

This page intentionally left blank.

Executive Summary

Digital computing systems needed for Army programs such as Command and Control (C2) processing and the Computer-Aided Low Altitude Helicopter Flight Program may be characterized by high computational throughput and input/output bandwidth, hard real-time response, high reliability and availability, and maintainability, testability, and producibility requirements. In addition, such a system should be affordable to produce, procure, maintain, and upgrade.

To address these needs the Army Fault Tolerant Architecture (AFTA) is being designed and constructed under a multi-year program comprising the Conceptual Study, Detailed Design and Fabrication, and Demonstration and Validation phases. This report describes the results of the Detailed Design of the AFTA, conducted during Government Fiscal Years (GFYs) 1992 and 1993.

AFTA is a militarized version of the Fault Tolerant Parallel Processor (FTPP) developed by the Charles Stark Draper Laboratory, Inc. AFTA is superior to existing fault tolerant computer technology in several respects. AFTA is a hard-real-time Byzantine resilient parallel processor. Due to its Byzantine resilience, it is capable of tolerating arbitrary failure behavior, as opposed to tolerating only a limited class of faults. It is a scalable parallel processor, which means that processors may be easily added as an application's performance requirements evolve, or as new applications emerge. It supports testability and redundancy management strategies which permit the dynamic reconfiguration of the parallel processing sites into redundant groups to enhance sortie availability and mission reliability. This means that the processing reliability levels can be optimized for given applications to achieve cost-effective fault tolerance. It is an open system, based on industry hardware and software standards and composed largely of Non-Developmental Items. This has the benefits of reducing the cost and risk of development, modification for different missions, and upgrading existing installations. AFTA's fault tolerance is transparent to applications programmers, allowing extensive reuse of existing code as well as simplification of the task of writing new code. Extensive analytical models and predictive verification and validation techniques are provided with AFTA to allow application designers to engineer a configuration for specific missions with a high degree of confidence that the fielded configuration will meet the mission requirements.

AFTA's architectural theory of operation, the AFTA hardware architecture and components, and the architecture of the AFTA Ada run time system (Ada RTS) were defined during a Conceptual Study, as well as a test and maintenance strategy for use in fielded

AFTA installations. A format was developed for representing mission requirements in a manner suitable for first-order AFTA sizing and analysis. Preliminary requirements were obtained for two Army missions: a rotary winged aircraft mission and a ground vehicle mission. An approach to be used in reducing the probability of AFTA failure due to common-mode faults was developed, as well as analytical models for AFTA performance, reliability, availability, life cycle cost, weight, power, and volume. A plan has been developed for verifying and validating key AFTA concepts during the Dem/Val phase, especially those which cannot be cost-effectively validated by accelerated life cycle testing. The analytical models and partial Army mission requirements developed under the Conceptual Study have been used to evaluate AFTA configurations for the two selected Army missions. To assist in documentation and reprocurement of AFTA components, VHDL is used to describe and design AFTA's developmental hardware. Finally, the requirements, architecture, and operational theory of the AFTA Fault Tolerant Data Bus were defined and described.

The AFTA program has now completed the Detailed Design Phase. During this phase, the hardware and software architectures recommended from the Conceptual Study phase were designed in preparation for Brassboard fabrication in the Fabrication, Integration, and Validation phase. Using internal Draper funding, an AFTA Brassboard was fabricated. Under Army funding, the AFTA open systems design philosophy was extended from the hardware to its operating system and programming languages. Specifically, a commercial off-the-shelf Portable Operating System Interface (POSIX)-compliant operating system was ported to the AFTA, and an existing flight-critical Army application was demonstrated on the AFTA Brassboard. The existing code, which was written for a nonredundant system, was ported to the fault tolerant AFTA in under one week.

Table of Contents

List of Figures

List of Tables

1. Introduction

The long-term objective of the AFTA program is to develop and deploy the Army Fault Tolerant Architecture (AFTA) on a variety of Army programs such as the Rotorcraft Pilot's Associate, the Comanche, and Command and Control (C2) applications. Applications such as these may be characterized by a combination of computational intensiveness, real-time response requirements, high reliability and availability requirements, maintainability, testability, and producibility requirements, and sensitivity to life-cycle costs.

The AFTA architecture is based on the Charles Stark Draper Laboratory, Inc. Fault Tolerant Parallel Processor (FTPP). AFTA is a real-time computer possessing high reliability, maintainability, availability, testability, and computational capability. It achieves the first four properties primarily through adherence to a theoretically rigorous theory of fault tolerance known as Byzantine Resilience, through which arbitrary failure modes can be tolerated. It is designed for verifiability and quantifiability of key system attributes with a high degree of confidence, in part due to its theoretically sound basis and in part due to plausible parameterizations of fault tolerance and Operating System overheads. Through the use of parallel processing, AFTA provides the throughput for future integrated avionics and control functions. To be useful for a variety of Army applications, the number and redundancy level of processing sites in AFTA may be varied from one application to another. Two operating systems have been hosted on AFTA. The first is a dedicated Ada run time system, and the second is an industry-standard Portable Operating System Interface (POSIX)-compliant operating system. AFTA is intended to be easy to produce and upgrade through extensive use of Non Developmental Items and compliance with well-accepted electrical, mechanical, and functional standards.

Over the past few years NASA and the Strategic Defense Initiative Office (SDIO) have sponsored the Advanced Information Processing System (AIPS) program at Draper Laboratory. The overall goal of the AIPS program is to produce the knowledgebase necessary to achieve validated distributed fault tolerant computer system architectures for advanced real-time aerospace applications [Har91b]. As a part of this effort, an AIPS engineering model consisting of hardware building blocks such as Fault Tolerant Processors and Inter-Computer (IC) and Input/Output (I/O) networks and software building blocks such as Local System Services, IC and I/O Communications Services was constructed. AFTA can be considered to be a high-throughput AIPS building block which can be interfaced to the AIPS IC network. Section 3.7 of the AFTA Conceptual Study describes the AIPS engineering model in more detail and illustrates how it can be interfaced with AFTA.

The AFTA Detailed Design Phase One Documentation consists of three volumes. Volume I, "Overview Volume," contains the program overview and status, updated performance and reliability models, and a description of the deliverables of this phase of the AFTA development. Volume II, "AFTA Software Documentation," includes the Software Development Plan (SDP), the Ada RTS Software Requirements Specification (SRS), and a directory of the files delivered on digitally readable media. Source code listings are provided for the AFTA Ada Run Time System, the Network Element Simulator, the Performance Measurement and Data Processing software, the Static Code Execution Time Analysis software, and the Network Element Self Test software. Volume III, "AFTA Network Element Hardware Documentation," contains an updated version of Section 4 of the Conceptual Study describing the operational overview of the Network Element and a directory of all files delivered on digitally readable media. Board layouts, schematics, Programmable Array Logic (PAL) equations, VHSIC Hardware Description Language (VHDL) source and testbench code, VHDL testbench inputs and outputs, timing diagrams, microcode, and Field Programmable Gate Array (FPGA) designs are provided for the Network Element.

Because some readers may wish only to read individual volumes, Volumes I-III may contain some redundant information such as references and a glossary of terms and acronyms. In addition, in order for this documentation to be relatively self-contained, review material from the Conceptual Study is incorporated at judicious points.

2. Program Overview and Status

2.1. AFTA Overview

2.1.1. AFTA Hardware Architecture

The AFTA is based on the Fault Tolerant Parallel Processor (FTPP) architecture developed by Draper Laboratory. The FTPP architecture was conceived to satisfy the dual requirements for a computer system of ultra-high reliability and high throughput. To satisfy the first requirement, the FTPP is designed to be resilient to Byzantine faults. To satisfy the throughput requirement, the architecture includes multiple processing elements to provide parallel processing capability. For a detailed description of the FTPP the reader is referred to [Abl88], [Bab90a], [Har87], [Har88a], [Har88b], and [Har91a].

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

A diagram of the physical AFTA configuration is shown in Figure 1. The AFTA cluster consists of 4 or 5 Fault Containment Regions (FCRs). A fault occurring in one FCR can not cause another FCR to malfunction; this is achieved by providing each FCR with independent sources of power, clocking, and dielectric and physical isolation. FCRs reside in Line Replaceable Units (LRUs). FCRs may either be distributed among several LRUs for damage tolerance or integrated into a single LRU if damage tolerance is not an issue. Each FCR contains an NE, 0 to 8 PEs, a PC, and 0 or more IOCs. A minimal AFTA configuration consists of at least four NEs and three PEs; a maximal system would consist of five NEs and forty PEs. Selection of the number of NEs and PEs for a given application is made according to performance, reliability, availability, and other engineering requirements.

Devices in an FCR are interconnected using one or more standardized backplane buses. Depending on the procuring organization, this could be the VMEbus, SAVA SBBUS, PIbus, Futurebus+, or some other bus. The NE's bus-dependent and bus-independent circuitry are intentionally partitioned such that changes in the FCR backplane bus only affect the former, allowing the AFTA concept to transition from one standards suite to another with minimal hardware redesign.

The NEs provide communication between PEs, keep the FCRs synchronized, maintain data consensus among FCRs, and provide dielectric isolation between the FCRs via fiber optic links. The NE implements the protocol requirements for Byzantine resilience [LSP82]. The NE is the only developmental hardware item in AFTA. To facilitate its design, simulation, fabrication, and reprocurement, the NE is described using VHDL.

Each PE consists of a processor, private RAM and ROM, and miscellaneous support devices, such as periodic timer interrupts. The PEs may optionally have private I/O devices, such as Ethernet, RS-232, etc. The processor may be either a general-purpose processor or a special-purpose processor for signal or image processing. Multiple processor types may coexist simultaneously and interoperate in an AFTA implementation.

The IOCs connect AFTA to the outside world, and can be any module that is compatible with the FCR standard backplane bus. Interfaces to communication networks such as the JIAWG HSDB and the AFTA Fault Tolerant Data Bus (FTDB) are also classified as IOCs. Alternatively, for maximum I/O bandwidth, multiple dedicated I/O buses may be used. Both options are shown in Figure 1.

To achieve transparent processor reliability, nonredundant PEs are grouped into Virtual Groups (VGs), depicted in Figures 1 and 2. Byzantine resilient triplexes and quadruplex VGs consist of three and four PEs, respectively. Virtual groups consisting of only one processing site are called simplexes. The ensemble of Network Elements provides a virtual bus abstraction connecting the VGs. This abstraction conceals the multiple NEs and their interconnect, replacing it with a simple bus-oriented abstraction.

As mentioned earlier, two operating systems have currently been hosted on AFTA: a dedicated Ada run time system and an industry-standard POSIX-compliant operating system. Many of the features outlined in the following section apply equally to both the Ada run time system and the POSIX-compliant operating system. Details on the status of the POSIX operating system are provided in Section 5.

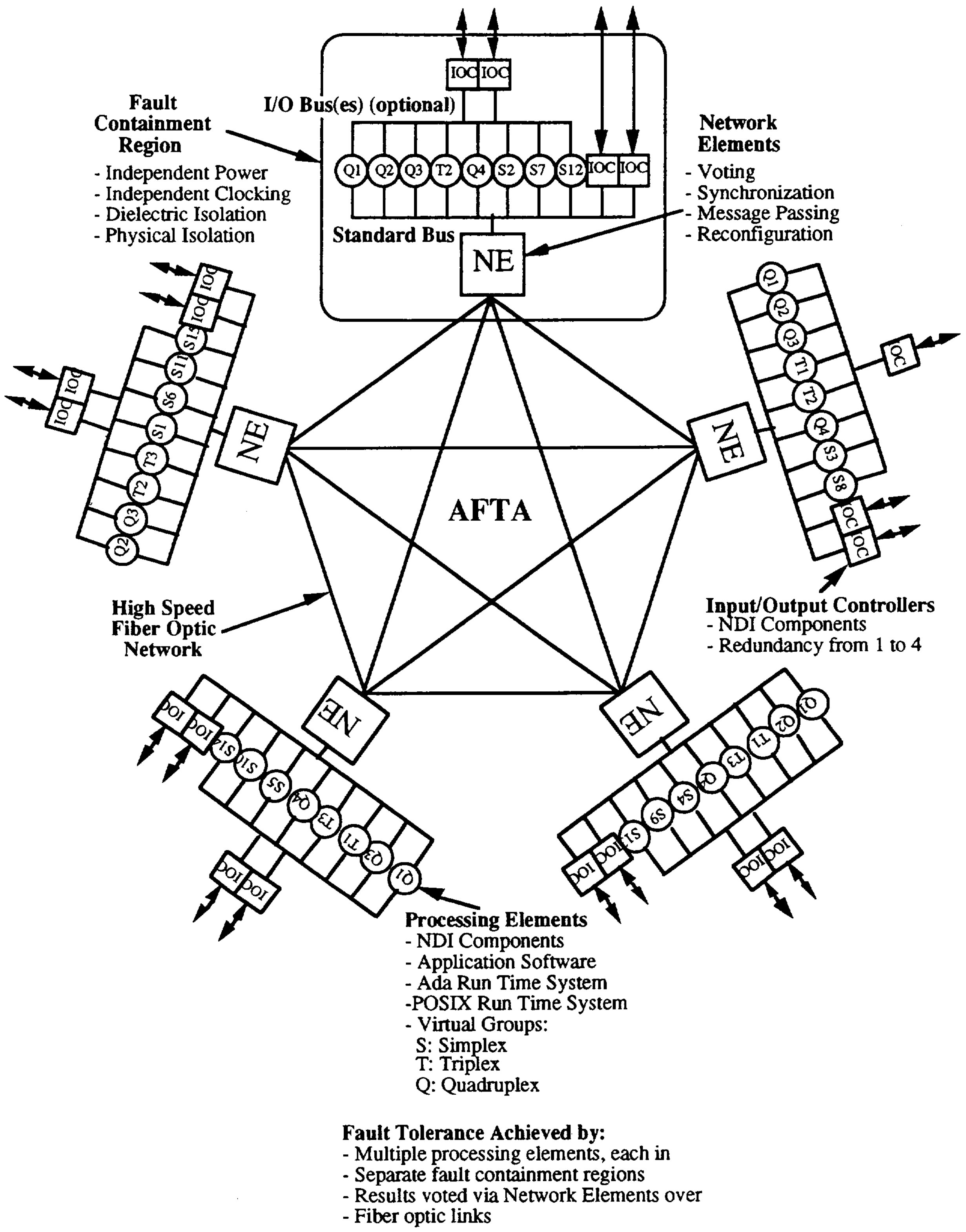

Figure 1. AFTA Physical Architecture

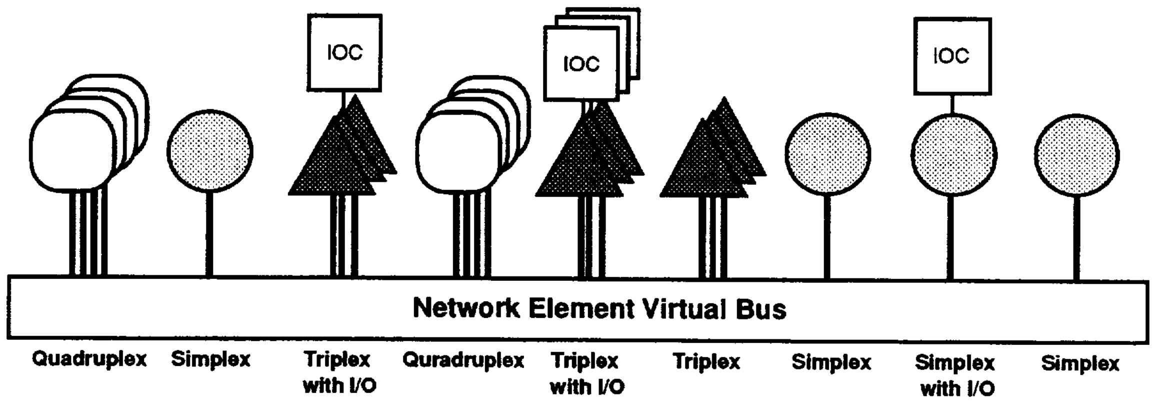

Figure 2. AFTA Virtual Configuration

2.1.2. AFTA Ada Run Time System

The foundation of the Ada run time system for the AFTA consists of a vendor-supplied Ada Run-Time System and Draper-supplied extensions based on recommendations made by the Ada Run-Time Environment Working Group. Additional features are required to manage the plurality of AFTA resources in a manner appropriate to the mission requirements.

AFTA processing is distributed by task, and intertask communication is provided by message passing. High reliability is provided by redundantly executing the tasks on replicated processors. The AFTA hardware and software have been designed to hide the hardware redundancy, hardware faults, and the distributed processing details from the applications programmer.

A system configuration specifies the mapping from tasks to VGs and from VGs to processors. This mapping is maintained by the operating system and is used to isolate the applications programmer from the underlying redundancy and distributed processing mapping. System initialization uses the above mapping to test the hardware components of the system and evaluate whether there are sufficient resources to perform the mission.

AFTA is perhaps best viewed as a layered system. The top layer consists of the applications programs themselves. In an ideal world, these are constructed by the systems engineers without regard for the parallel and redundant nature of AFTA. In reality, the systems engineers must, to some extent, assist in the selection of appropriate task-to-VG mappings, processing site redundancy levels, fault recovery strategies, and other parameters from among those made available by the AFTA architecture.

The next lower layer consists of the AFTA System Services. Several services may be invoked by the applications programmer; these include task scheduling, intertask communication, and input/output. This layer is intended to mask the complexity of AFTA's lower layers from the programmer.

The AFTA Ada RTS supports two different styles of scheduling. The first, known as rate group scheduling, is suitable for task suites in which each task has a well-defined iteration rate and can be validated to have an execution time which is guaranteed to not exceed its iteration frame (the inverse of its iteration rate). Flight control is an example of such a task. The baseline AFTA rate group frames run at 100, 50, 25, and 12.5 Hz; the number and frequencies of frames are easily changed. The second style of scheduling, herein known as "non-rate group scheduling," is used when the iteration rate of a particular task is unknown or undefined. A mission planning algorithm is an example of such a task. Validation of the temporal behavior of such tasks may be difficult. Non-rate group tasks are not allowed to perturb the critical timing behavior of rate-group tasks. This is achieved by scheduling them with a lower priority than rate-group tasks.

The AFTA communication services support intertask communication in the form of asynchronous message passing. A sending task is not required to be cognizant of the VG hosting the destination task–it identifies the destination task via a logical task identifier. Message delivery, correctness, and ordering are guaranteed in the presence of Byzantine faults according to the Byzantine Resilient Virtual Circuit Abstraction [Har87].

The I/O services provide communications between the application program and external devices (sensors and actuators). They execute on any VG which is responsible for I/O and provide source congruency on all input data and voting of all output data. The I/O services provide the user with the ability to group I/O transactions into chains and I/O requests. It also allows the user to schedule both preemptive and non-preemptive I/O. I/O activity is slaved to timer-based interrupts on the VG to reduce jitter. It is expected that many VGs will be accessing I/O devices concurrently to maximize the system's overall I/O bandwidth.

Other important functions of the AFTA System Services are not directly accessible by the applications programmer and are performed in a manner which is largely transparent. These include the traditional functions of preemption of lower priority tasks by higher priority ones, routing intertask messages to remote VGs, disassembling and reassembling long messages, Built-In Testing and fault logging, and fielding software exceptions. Other less traditional functions are Fault Detection, Identification, and Recovery (FDIR), reconfiguration of the parallel resources into redundant computing sites, and interfacing to the NE.

FDIR is composed of local FDIR which executes on each VG and system FDIR which executes on a specially designated VG. Local FDIR has the responsibility for detecting and identifying hardware faults in the PEs of its VG and disabling their outputs using the interlock hardware. In addition, local FDIR reports all link and NE faults to system FDIR and responds to its reconfiguration commands. It is also responsible for transient fault discrimination and for running self tests to detect latent faults. The system FDIR is responsible for the collection of status from the local FDIR and detection, identification and masking of NE faults, and link faults. It resolves conflicting local fault identification decisions, disambiguates unresolved faults, correlates transient faults, and handles VG failures.

When a faulty component has been identified, FDIR initiates an appropriate recovery strategy which attempts to compensate for the loss of a component. The variety of recovery strategies is vast, not only because the policy must be commensurate with the type of component failed but also because of the system requirements and the mission phase. The array of recovery policies includes a strategy to replace a faulty processor with a spare processor, an option to migrate a task when its processor fails, and a policy to quickly mask the incorrect behavior of a failed component.

The next lower layer of AFTA consists of the interprocessor communication network hardware, i.e., the NEs. This hardware implements the interprocessor message passing functions of AFTA. In addition, it implements throughput-critical fault tolerance-specific functions such as voting of messages emanating from redundant processing sites, providing error indications, assisting in synchronizing redundant processing sites, and assisting in arranging the non-redundant parallel processing resources of AFTA into redundant processing sites based on the needs of the application, mission mode, and the fault state of AFTA.

At the lowest layer of interest reside the inter-NE communication links, which provide high-bandwidth, dielectrically isolating, optical communication paths between the AFTA FCRs. The data transmissions over the links also keep the NEs synchronized to within ±80ns using digital phase-locked loop techniques.

2.2. Long-Term AFTA Development Plan

To achieve the AFTA program's long-term objective requires a multi-phased product development, production, and support cycle. A useful model for AFTA's development and deployment cycle is based on that found in MIL-STD-785B, "Reliability Program for Systems and Equipment Development and Production" [MIL-STD-785B].

First, a Conceptual Study phase is performed to ascertain the requirements of anticipated applications and develop concepts suitable for those applications. Quantitative formulations are developed for critical parameters such as performance, reliability, etc., appropriate to the level of detail available from the requirements and the proposed architectural concepts. Deliverables of this phase include a document describing the application requirements, the structure and operational theory of the proposed conceptual solution, analytical models and results used in evaluating the architecture, plans for evaluating and verifying the analytical predictions, and plans for further development phases. This documentation is provided both in hardcopy and digital format.

Next, a Demonstration and Validation (Dem/Val) phase is executed, in which the candidate solution is refined through extensive study and analysis, hardware development, test, and evaluation. In the AFTA program, a prototype of the architecture is designed and constructed from commercially available hardware, and is denoted the *AFTA Brassboard.* This prototype serves as a testbed for evaluation and improvement of the architectural concept, increases confidence in the viability of the architecture, provides information regarding the interaction of system components, and corroborates preliminary analytical and functional models. In the Dem/Val phase, the verifiable attributes of the Brassboard are investigated according to the verification plan described in Section 11 of the Conceptual Study report, and a preliminary Failure Modes and Effects and Criticality Analysis (FMECA) is performed to identify reliability bottlenecks needing attention. The analyses produced under the Conceptual Study phase are refined based on detailed design and empirical data obtained from the Dem/Val phase, and a Full Scale Development plan is constructed. If deployable Non Developmental Items are available for use in the Brassboard, Reliability Development/Growth Testing for these items may be initiated. Deliverables of this phase include one or more copies of the Brassboard, detailed design information such as mechanical drawings, parts lists, schematics, timing analyses, data and control flow diagrams, Interface Control Documents, VHDL, ADA, and Assembler source code, hardware and software documentation, test and evaluation results, refined analytical models, the FMECA, and user/programmer guides. The documentation is provided both in hardcopy and digital format.

Upon satisfactory demonstration, validation, and refinement of the architectural concept, the Full Scale Development phase (FSD) is entered, during which the system and the principal items necessary for its support are designed, fabricated, tested, and evaluated.

The FSD phase begins with the construction of numerous plans. These include Engineering Development Model (EDM) fabrication, incoming/outgoing Quality Assurance, Environmental Stress Screening (ESS), Reliability Development/Growth Testing (RDGT), Failure Reporting And Corrective Action, Validation and Verification, Full-Scale Production (FSP), logistics, Pre-Planned Product Improvement (P^3I), and maintenance plans. A detailed Failure Modes and Effects and Criticality Analysis (FMECA) is performed to identify AFTA reliability bottlenecks. Production acceptance tests such as the Production Reliability Acceptance Test are defined. The Preliminary Design Review, Critical Design Review, and Production Readiness Review are scheduled. Deliverables from the FSD planning phase include the plans and schedule outlined above in hardcopy and digital format. Upon satisfactory completion of the FSD plans, fabrication of the EDM begins. The EDM is as far as possible identical to systems planned for Full Scale Production (FSP); for AFTA, it is constructed of military-qualified components in packages and form factors suitable for installation in the vehicles of interest. The EDM is used to verify the producibility of AFTA, undergo ESS and RDGT, and refine quantitative predictive models of AFTA attributes. Deliverables from the EDM phase include one or more EDM copies, detailed EDM engineering documentation, the FMECA, results from the ESS and RDGT, and Validation and Verification results.

After EDM testing and acquisition of detailed application requirements, the architecture is ready for Full Scale Production (FSP), in which units intended for use in one or more deployments are produced in quantity. While in use in the field, all systems (even AFTA) suffer faults and require continual maintenance, spares, and associated logistics support. During production and deployment a Failure Reporting And Corrective Action plan is exercised to identify failure modes, trace them back to weak components, and, if possible, modify the design, parts, and/or fabrication process to eliminate them. Over the AFTA's fielded life, Pre-Planned Product Improvements (P^3I) may be implemented to increase system capabilities, increase reliability/availability, and reduce support costs. It is generally expected that the field support costs will far exceed all other development and procurement costs. Finally, all systems (even AFTA) become obsolete with time, enter old age and are replaced with newer technology.

The Conceptual Study and Dem/Val phases will now be discussed in chronological order.

2.3. Conceptual Study

The near-term objective of the AFTA program is to demonstrate and evaluate the Army Fault Tolerant Architecture (AFTA) Brassboard within the context of the Computer-Aided Low Altitude Helicopter Flight Program and the Armored Systems Modernization (ASM) Program. The subject program consists of the first two phases in the product development cycle discussed above, namely the Conceptual Study and the Brassboard Demonstration/Validation phases. These two phases are further partitioned into three separate subtasks:

1. Conceptual Study
2. Detailed Design
3. AFTA Brassboard Fabrication and Evaluation

Due to funding limitations the AFTA program Detailed Design was stretched over two years (GFY92 and GFY93), while the Brassboard Fabrication was completed on schedule under separate funding.

The approximate schedule for these phases is given in Figure 3:

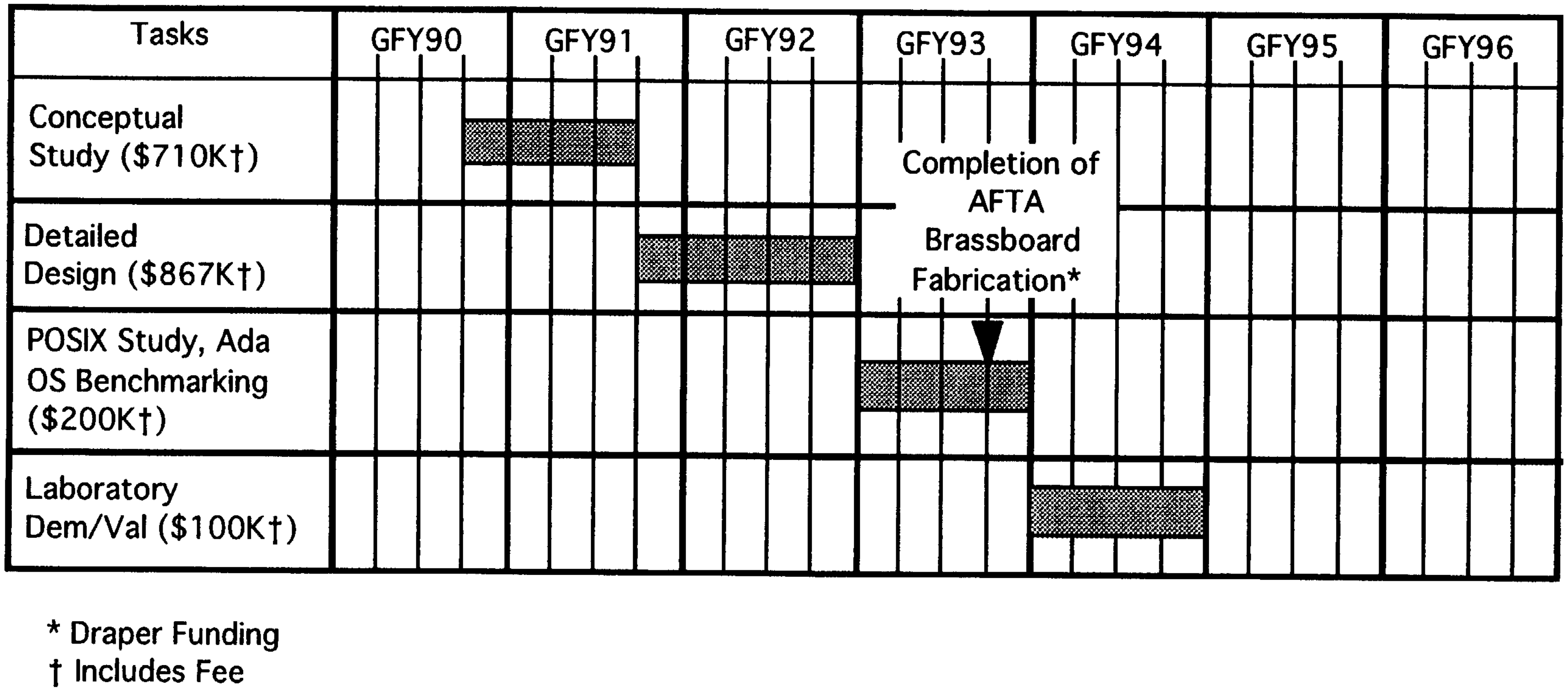

Figure 3. Near-Term Schedule for the AFTA Program

The Conceptual Study comprised the Requirements Definition, Requirements Acquisition, Engineering Description, Analytical Modeling, Verification Plan, Architecture Configuration, the C2 Loaner, and the Fault Tolerant Data Bus subtasks.

In the Requirements Definition phase, we defined a format for requirements that the application designer may place upon the computational system. Relevant requirements data include reliability, maintainability, availability, testability (RMAT), performance requirements, operational environment, mission scenario, and maintenance strategy.

In the Requirements Acquisition phase, available data were obtained for the Army missions of interest from C2SID, CECOM-C^3, and RAMECES. These requirements were determined by the Computer-Aided Low Altitude Helicopter Flight and the Ground Maneuver Systems Fault Tolerant Navigation Processor programs. For brevity these applications are henceforth referred to as the "TF/TA/NOE" (for Terrain-Following/Terrain-Avoidance/Nap-of-the-Earth) and the "Ground Vehicle" applications, respectively. The requirements were transformed where possible into the format defined in the Requirements Definition phase.

In the Engineering Description phase, a detailed description was generated of the components of AFTA and how they are assembled and operated. The engineering description is sufficiently detailed to provide the analytical models with parameters such as throughput, memory, intertask communication bandwidth and latency, input/output bandwidth and latency, weight, power, size, volume, and component failure rate as a function of the architecture configuration chosen for a given Army application. In addition, the engineering description provides details on how to develop software for AFTA and operational details on fault tolerance and recovery schemes.

In the Analytical Modeling phase, analytical models were constructed to predict whether a given AFTA configuration will meet the requirements as specified in the Requirements Acquisition phase. These models are parameterized so as to be useful in estimating the characteristics of the Brassboard as well as multiple deployable AFTA configurations.

The Verification Plan phase comprises the construction of a plan for demonstrating that the analytical models predict system characteristics with reasonable accuracy. This plan is executed in the Dem/Val phase.

In the Architecture Configuration phase, the AFTA architectural parameters were adjusted to realize conceptual architectures for the two Army missions: the helicopter TF/TA/NOE mission and the Ground Vehicle mission. The analytical models developed in the Analytical

Modeling phase are used to predict AFTA reliability, availability, weight, power, volume, and Life Cycle Costs for these missions.

In the C2 loaner subtask, the AFTA Cluster 2 (C2) was delivered to C2SID for evaluation and familiarization with AFTA technology. The C2 is a quadruply redundant uniprocessor version of the AFTA and hosts the same basic Ada Run Time System and software development environment as AFTA. The AFTA software development environment was purchased and delivered to C2SID in the Conceptual Study phase to jump start the AFTA application software development process. In a related effort, the testability of the C2 Network Element (NE) was evaluated by writing and demonstrating self-test software; the lessons learned from this exercise will be used to improve the testability of the AFTA Network Element.

Common-mode faults are those which occur in more than one copy of a redundant computation due to a common source. Thus, they can defeat redundancy-based fault tolerance techniques such as those used in AFTA. A methodology for detecting and recovering from common-mode faults in AFTA was developed. In addition, a plan for verifying the effectiveness of the common-mode fault tolerance techniques comprising the methodology was formulated.

As a separate but related effort, an FTDB was developed to provide a fault tolerant networking system for AFTA and other digital systems, including the Silicon Graphics display processor, the Merit Technologies MT-1 VME system, the US Air Force Real-Time AI System (RTAIS), sensor and image processors, and flight and engine controls. The objective of the fault-tolerant data bus effort is to provide highly reliable end-to-end communications between the above systems. The conceptual design of the FTDB covered many aspects of network design, including media technology, media access control, topology, routing, OSI protocol stacks, and fault detection and recovery. In addition to these traditional network topics, the FTDB also encompasses techniques from the area of fault-tolerance, including Byzantine resilience and authentication protocols.

2.4. AFTA Brassboard Demonstration and Validation

Following the completion and evaluation of the Conceptual Study phase the Brassboard Dem/Val phase begins. The first year of Dem/Val comprises the Detailed Design phase, while the second year comprises the Fabrication, Integration, and Validation phase. The program is currently at the completion of the Detailed Design phase. An AFTA has been constructed and demonstrated on two applications, partially under separate funding.

2.4.1. Detailed Design

The intent of the detailed design phase is to design the hardware and software architectures recommended from the Conceptual Study phase, in preparation for Brassboard fabrication in the Fabrication, Integration, and Validation phase. It comprises work in the following areas.

The design of the Brassboard AFTA Network Element is completed. The design of the backplane-independent components of the NE is described in VHDL at the behavioral level. A comprehensive set of NE self-tests is designed, and a software simulation of the NE is constructed for use in Operating System and other AFTA software development efforts.

The basic AFTA Ada run time system (Ada RTS) is designed, documented, and key system functions are prototyped. The Ada RTS includes task scheduling, intertask communication, input/output services, and Fault Detection, Identification, and Recovery functions.

The quantitative models of AFTA's reliability, availability, weight, power, volume, failure rate, life-cycle cost, and other parameters are refined as design and application mission details become available.

The schedulable milestones for the Detailed Design phase are listed below. All of this work has been completed as of the end of the Detailed Design phase.

Hardware:

1. Complete the detailed design of the AFTA Brassboard Network Element. This includes schematics, netlists, PAL equations, microcode, timing diagrams, parts lists, and board layouts.
2. Complete the Network Element Simulator.
3. Complete the fabrication and testing of a single AFTA Brassboard Network Element.
4. Complete the VHDL behavioral model of the Network Element Scoreboard and Data Path Board.

Basic Ada Run Time System:

1. Complete the Software Development Plan.
2. Complete the Ada RTS Software Requirements Specification.
3. Develop a debugging/development support environment.
4. Perform the Detailed Design of the Ada run time system.

5. Code and test a key subset of Ada run time system functions with the Network Element Simulator.

6. Test a key subset of Ada run time system functions with the existing AFTA Brassboard Network Element.

Quantitative Models:

1. Update the quantitative models of AFTA based on evolving engineering data and mission details.

2.4.2. Fabrication, Integration, Validation

In the Fabrication, Integration, Validation Phase one or more Brassboard AFTAs are assembled. The AFTA's Network Elements (NEs) are fabricated and tested, the Processing Elements (PEs), Input/Output Controllers (IOCs), backplanes, and Power Conditioners (PCs) are purchased, and the Operating System (OS) software is completed.

After fabrication and integration of the components the Brassboard is delivered to the Army for demonstration and validation. For the demonstration, a representative application is ported to AFTA. Subsequently, the critical parameters of AFTA are evaluated according to the verification plan described in Section 11 of the Conceptual Study.

The following parameters are measured, both with and without injected faults in relation to the TF/TA NOE application:

1. Delivered throughput per processing site
2. Available memory per processing site
3. Effective intertask communication bandwidth
4. Effective I/O bandwidth
5. Iteration rate of a task
6. Reliability
7. Availability
8. Testability
9. Cost per unit of service
10. Weight, power, and volume

The fault recovery and common mode fault tolerance capabilities specified by C2SID will also be demonstrated.

Deliverables of this phase include one or more copies of the Brassboard, detailed test and evaluation results, and refined analytical models. The documentation is provided both in hardcopy and digital format.

2.5. Documents Used and Generated Under This Contract

The documents used under the AFTA Detailed Design phase are listed in Section 6, "References." The documents generated under this contract are listed below.

1. NASA Contractor Report 189632, Volumes I and II, "Advanced Information Processing System: The Army Fault Tolerant Architecture Conceptual Study," July 1992.
2. AFTA Detailed Design Phase Documentation, Volumes I (NASA Contractor Report 194924, "Advanced Information Processing System: The Army Fault Tolerant Architecture Detailed Design Overview"), II, and III, June 1994.
3. "System Performance Modeling and Analysis of a Fault-Tolerant Real-Time Parallel Processor," R. J. Clasen, Master of Science Thesis, Northeastern University, May 1993.

2.6. Overview of Detailed Design Phase Deliverables

The primary deliverables of the AFTA Detailed Design phase is a set of engineering documents which are adequate to facilitate the construction of the AFTA Brassboard. The documents represent the state of the design prior to Brassboard fabrication and, by necessity, do not reflect design updates resulting from the Brassboard fabrication, validation, and verification. The documentation consists of three volumes.

Volume I, "Overview Volume," contains the program overview and status, updated performance and reliability models, and an description of the deliverables of this phase of the AFTA development.

Volume II, "AFTA Software Documentation," contains the AFTA Software documentation, including the Software Development Plan (SDP), the Software Requirements Specification (SRS), and a directory of the files delivered on digitally readable media. Source code listings are provided for the AFTA Ada Run Time System, the Network Element Simulator, the Performance Measurement and Data Processing software, the Static Code Execution Time Analysis software, and the Network Element Self Test software.

Volume III, “AFTA Network Element hardware Documentation,” contains an updated version of Section 4 of the Conceptual Study describing the operational overview of the Network Element and a directory of all files delivered on digitally readable media. Board layouts, schematics, Programmable Array Logic (PAL) equations, VHDL source and testbench code, VHDL testbench inputs and outputs, timing diagrams, microcode, and Field Programmable Gate Array (FPGA) designs are provided for the Network Element.

3. Updated Reliability Model

The AFTA reliability and availability are strong functions of the policies which are used to manage the redundant AFTA resources. A design change in the FDIR software arising out of the Detailed Design phase may have an impact on the reliability obtained using one of these options. Before describing the design change and its quantitative impact on AFTA reliability, the two redundancy management options selected for analysis under the AFTA program will be reviewed.

3.1. Review of Two AFTA Redundancy Management Options

The reliability and availability of an AFTA implementation is a function of the number of FCRs and PEs, the VG redundancy levels, the mission environment, the operational and maintenance scenario, and fault recovery procedures. Two classes of AFTA fault recovery options were analyzed in the AFTA Conceptual Study, and each one has a different impact on the overall AFTA reliability and availability.

The first class of options, of which the *graceful degradation* and *Network Element masking* in Section 5.6.6 of the Conceptual Study Final Report are examples, are appropriate for an operational mode in which little if any time is available for fault recovery. In this case, a faulty component in a redundant VG or an NE is immediately disabled upon detection, with no lengthy fault recovery attempted. No effort is made to discriminate between transient and permanent faults for the purpose of performing on-line recovery, in effect treating all faults as permanent until a more relaxed operational regime is entered. This option has the advantage of incurring no dropout of functionality, but has the disadvantage of irreversibly reducing the redundancy level of the faulted VG and hastening its demise due to redundancy exhaustion. Therefore it may be viewed as being best suited for short missions having fast real-time constraints, such as real-time control of mission-critical helicopter functions.

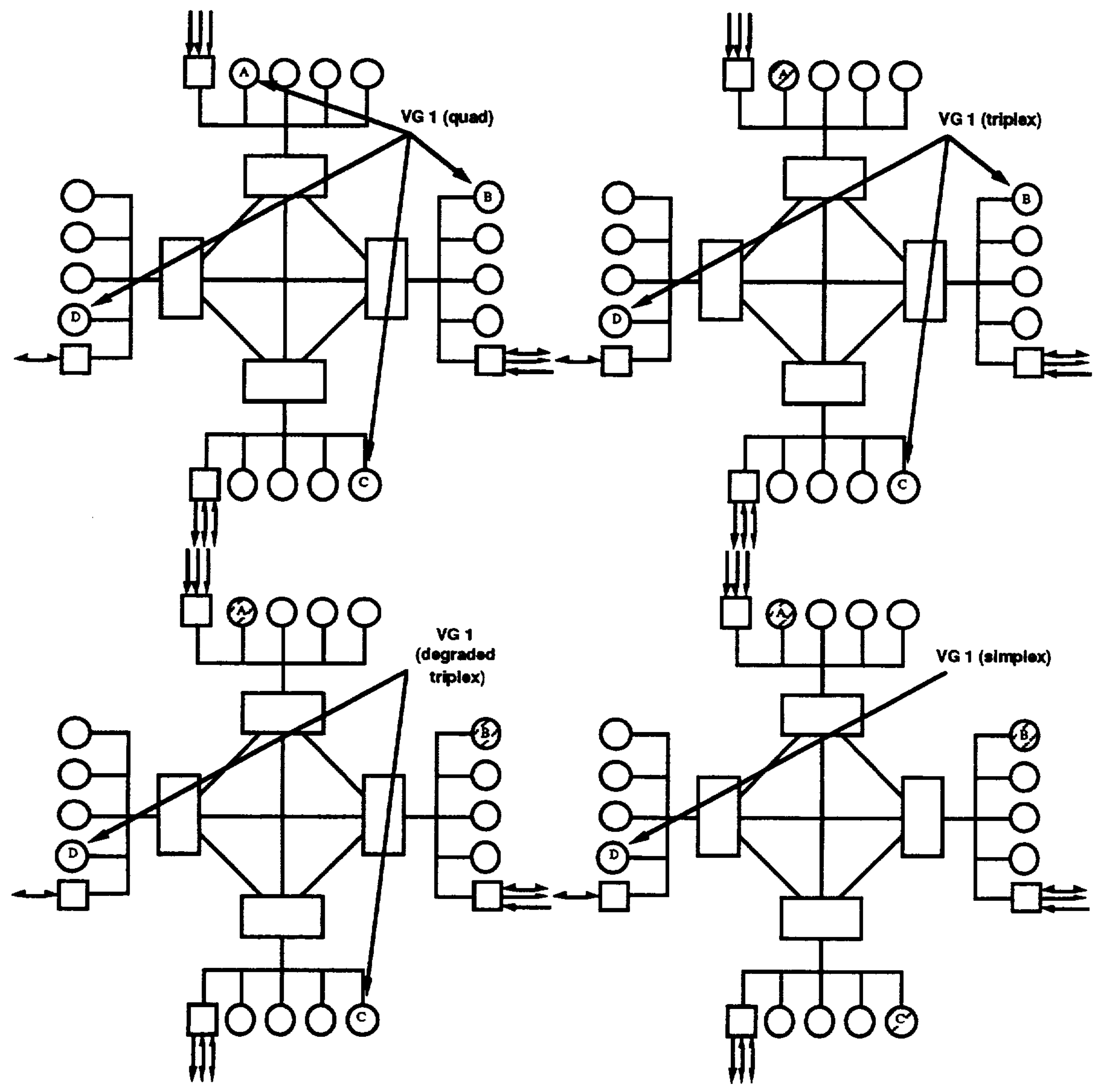

Figure 4. Graceful Degradation of Quadruplex VG_1

Figure 4 illustrates this fault recovery option: after the first failure of member A of quadruply-redundant VG_1, the faulted member is disabled, reducing VG_1's redundancy level to triplex. A second failure of one of VG_1's members, say B, reduces its redundancy level to "degraded triplex." For a degraded VG, the Network Element's main data path packet voter masks the input from the faulted member and does not include it in the vote. The Scoreboard, however, continues to consider a degraded VG's faulted channel when calculating the VG's voted Output Buffer Not Empty (known as OBNE, an indication that the VG has a packet to be transmitted from its Output Buffer) and voted Input Buffer Not Full (IBNF, an indication that the VG is capable of receiving at least one packet in its Input

Buffer)[†]. This is to allow a faulted member of a degraded VG to remain in synchronization with its parent VG to facilitate recovery operations. This capability is more robust and useful for degraded quadruplex VGs than for degraded triplex VGs.

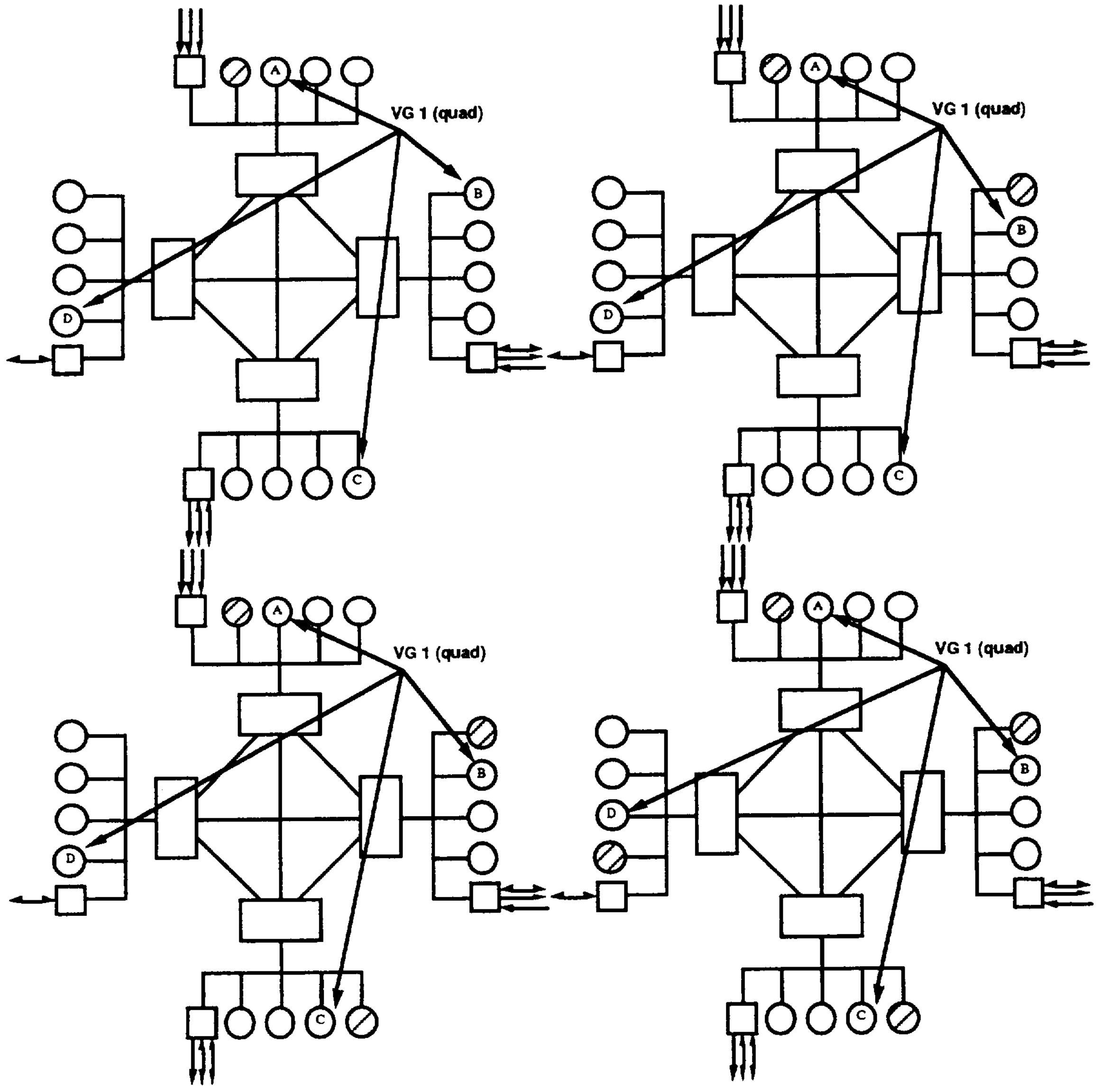

Figure 5. Processor Replacement Redundancy Management for Quadruplex VG_1

A third failure in VG_1, say of member C, reduces its redundancy level to simplex, and a fourth failure results in the loss of the functionality supported by VG_1. The probability of successfully transitioning from a faulted degraded triplex VG to a nonfaulty simplex is significantly less than unity, and is represented by the "duplex coverage," c_D.

[†] See Section 4 of the Conceptual Study Final Report for a discussion of this terminology.

When a fault recovery time on the order of a second of two is permissible, a wider range of fault recovery options are available. Representatives of this class of options are listed in Section 5.6.6 of the Conceptual Study Final Report as *processor resynchronization, processor reintegration, processor replacement, processor replacement with initialization, task migration,* and *Network Element resynchronization.* All of these recovery options are characterized by their capability to seek and find components sufficient to maximize the likelihood of forming a desired configuration of redundant VGs, followed by either initializing or copying the state of the newly reintegrated component into agreement with the surviving members of the faulted VG. As is mentioned earlier, this process, while maximizing the effective use of the reconfigurable AFTA components, consumes one to two seconds to perform. As an example of such a strategy in the context of the previous example, we reconsider the case of a processor replacement fault recovery option applied to VG_1[†]. After a failure of member A of VG_1, VG_1's redundancy level can be restored by switching in (say) the PE adjacent to member A. After the second failure of member B, a spare processor may be reintegrated, again restoring VG_1's quadruplex redundancy level, and so on and so forth (Figure 5). This can continue until all the spares allocated to repairing VG_1 are exhausted, at which point the VG_1 fault recovery policy may revert to the graceful degradation policy described above, or another policy may go into effect.

The more leisurely fault recovery options in this class are more suited to less stressful real-time operational regimes and missions, such as during the hiatus phase of the flight mission where availability is to be maximized, or during a long ground mission where one or two second dropouts are a reasonable tradeoff for significant mission longevity enhancement.

3.2. Modifications Arising from Detailed Design Phase One

Section 9 of the Conceptual Study presented formulations of the probability that AFTA can perform its intended functions, i.e., form the requisite number of functioning VGs, when managed according to the two fault recovery policies outlined above. These formulations remain unchanged as a result of the evolution of the detailed design.

Mission times, environments, and failure rates were also presented in the Conceptual Study. These also remain unchanged. In the Conceptual Study, the AFTA component failure rates were calculated assuming that the hiatus environment corresponds to the Ground, Fixed (GF) environment, and the mission environment corresponds to the Rotary Wing Aircraft (AR) environment, both described in MIL-HDBK-217E. These values are un-

[†] Different VGs may have different fault recovery options, and the same VG's fault recovery option can vary over the course of a mission.

changed and are repeated below for convenience. Mission times for the helicopter mission ranged from 1 to 8 hours, a Minimum Dispatch Complement of 6 VGs was assumed, and a duplex coverage of 0.50 is used.

Component	GF failure rate, per hour	AR failure rate, per hour
PE	1.92E-5	6.58E-5
NE	4.08E-5	1.85E-4
PC	1.59E-5	5.40E-5
FCR Back-plane Bus	1.92E-6	6.58E-6

Table 1. AFTA Component Failure Rates for Helicopter Mission Scenario

Volume III of the Detailed Design Phase One Documentation describes how the Fault Detection, Identification, and Recovery (FDIR) design has been changed to facilitate implementation, testing, validation, and temporal determinism. In the new design, all fault diagnoses and recovery acts are executed or controlled by a single System Virtual Group, as opposed to the Conceptual Study design in which each VG was responsible for its own diagnosis and, in most cases, recovery. The net result of this FDIR design modification is that the mean time between error manifestation and recovery is increased from a value of 10 milliseconds to a larger, currently undetermined, value. It is expected that this value will be less than one second. The analytical parameter which represents the inverse of the recovery time is called the recovery rate and denoted μ in the AFTA dependability models. The following sections analyze the effect of varying this parameter on overall AFTA mission reliability and availability. The variation will be performed over a wide enough range to ensure that the final AFTA recovery rate is covered by the analysis.

3.3. Effect of Recovery Rate on AFTA Mission Reliability

During a mission having fast hard real-time constraints, the AFTA VGs are managed under a "graceful degradation" redundancy management policy. Under this policy, system failures may occur as a result of the arrival of a second fault while AFTA is in the process or recovering from a preceding fault. The probability of this occurrence is proportional to the mean time required for recovery, or, equivalently, inversely proportional to the recovery rate μ.

To illustrate the effect of an the recovery rate on AFTA mission reliability, the formulation for the reliability of the AFTA under this policy is repeated below from the Conceptual Study.

Let $\Xi(\lambda, \mu, t, r)$ represent the reliability at mission time t of a VG having processor failure rate λ, fault recovery rate μ, and redundancy level r, assuming PE faults only. This is the probability of occurrence of all operational states (redundancy levels of 1, 2, 3, or 4) of the VG minus the probability that the VG fails due to near-coincident PE faults.

$$\Xi(\lambda, \mu, t, r) = \begin{cases} \sum_{i=1}^{r}\left[c_i \binom{r}{i} \left(e^{-\lambda t}\right)^{i} \left(1-e^{-\lambda t}\right)^{r-i}\right] - \dfrac{r(r-1)\lambda^2 t}{\mu} & , r > 0 \\ 0 & , r \leq 0 \end{cases}$$

where c_i is the probability that a VG of redundancy level i+1 can successfully degrade to a VG of redundancy level i. The second term in the above equation is the approximate probability that the VG suffers catastrophic failure due to near-coincident PE faults.

If r>1, then

$$c_i = \begin{cases} c_D, i=1 \\ 1.0, i=2 \\ 1.0, i=3 \\ 1.0, i=4 \end{cases}$$

If r=1, then

$$c_D = 1.0$$

The parameter c_D ranges from 0.5 to 0.90, depending upon the level of effort put into tolerating faults in duplex VGs. A safe assumption is usually $c_D = 0.50$, since at worst the redundancy management function can, upon detecting a fault in a duplex VG, randomly guess which one is faulty and mask it out.

Let nelist(VG_i) represent the set of FCRs which contain at most one channel of VG_i. For example, if quadruply redundant VG_1 has members in FCRs 0, 1, 3, and 4, then nelist(VG_1) = {0, 1, 3, 4}.

The conditional VG reliability becomes

$$R(VG_i \mid \text{no FCR faults}) = \Xi(\lambda_{PE}, \mu_{PE}, t, redlev_i)$$

$$R(VG_i \mid FCR_j \text{ faulty}) = \begin{cases} \Xi(\lambda_{PE}, \mu_{PE}, t, \text{redlev}_i), j \notin \text{nelist}(VG_i) \\ \Xi(\lambda_{PE}, \mu_{PE}, t, \text{redlev}_i - 1), j \in \text{nelist}(VG_i) \end{cases}$$

and

$$R(VG_i \mid \text{FCRs } j, k \text{ faulty}) = \begin{cases} \Xi(\lambda_{PE}, \mu_{PE}, t, \text{redlev}_i), j \notin \text{netlist}(VG_i \text{ and } k \notin \text{nelist}(VG_i), \\ \Xi(\lambda_{PE}, \mu_{PE}, t, \text{redlev}_i - 1), j \notin \text{netlist}(VG_i \text{ and } k \in \text{nelist}(VG_i), \\ \Xi(\lambda_{PE}, \mu_{PE}, t, \text{redlev}_i - 1), j \in \text{netlist}(VG_i \text{ and } k \notin \text{nelist}(VG_i), \\ \Xi(\lambda_{PE}, \mu_{PE}, t, \text{redlev}_i - 2), j \in \text{netlist}(VG_i \text{ and } k \in \text{nelist}(VG_i) \end{cases}$$

This formulation for the conditional VG reliability is used to compute p_{GD}:

$$p_{GD} = \left[\prod_{VG_i \in F_j \; F_j \in S} R(VG_i \mid \text{no FCR faults}) \right] \Pr(\text{no FCR faults})$$

$$+ \sum_{n=1}^{NNEs} \left[\prod_{VG_i \in F_j \; F_j \in S} R(VG_i \mid \text{FCR } n \text{ faulty}) \right] \Pr(\text{FCR } n \text{ faulty})$$

$$+ K \sum_{n=1}^{NNEs} \sum_{m=1.\, m \neq n}^{NNEs} \left[\prod_{VG_i \in F_j \; F_j \in S} R(VG_i \mid \text{FCRs } n \text{ and } m \text{ faulty}) \right] \Pr(\text{FCR } n \text{ faulty}) \Pr(\text{FCR } m \text{ faulty})$$

Lengthy fault reconfiguration times can result in a significant probability of AFTA failure due to a second fault occurring while AFTA is recovering from a prior fault. The AFTA reliability and availability models compute the probability of failure due to near-coincident faults and attrition to allow estimation of their relative importance and the consequent need for intensive verification of reconfiguration time. Table 2 illustrates the probabilities that a single VG suffers failure due to attrition and near-coincident faults for the helicopter mission described in the Conceptual Study. A 100Hz recovery rate was used in this calculation, and the Minimum Dispatch Complement was assumed to be 6 VGs. Note that the two contributors to VG failure are of commensurate magnitude when quadruplex VGs are used for short mission times, and therefore statistically significant verification of reconfiguration time becomes an issue in this case. In general, large variations in the AFTA mission duration or reconfiguration time may force this quantity into prominence.

VG Redundancy Level	Probability of VG Failure due to Attrition	Probability of VG Failure due to Near-Coincident Faults
1 Hour Helicopter Mission		
Triplex	6.49E-09	7.21E-14
Quadruplex	7.14E-13	1.44E-13
2 Hour Helicopter Mission		
Triplex	2.59E-08	1.44E-13
Quadruplex	4.84E-12	2.88E-13

Table 2. VG Failure Probability Due to Attrition and Near-Coincident Faults - 100Hz Recovery Rate as Assumed in Conceptual Study

The following table shows the sensitivity of VG failure probability to varying recovery rate over two orders of magnitude. The 100Hz column repeated from the Conceptual Study to allow comparison with this design baseline.

Recovery Rate ⇒	100 Hz		10 Hz		1 Hz	
VG Redundancy Level ⇓	Attrition	Near-Coincident Faults	Attrition	Near-Coincident Faults	Attrition	Near-Coincident Faults
1 Hour Helicopter Mission						
Triplex	6.49E-09	7.21E-14	6.49E-09	7.21E-13	6.49E-09	7.21E-12
Quadruplex	7.14E-13	1.44E-13	7.14E-13	1.44E-12	7.14E-13	1.44E-11
2 Hour Helicopter Mission						
Triplex	2.59E-08	1.44E-13	2.59E-08	1.44E-12	2.59E-08	1.44E-11
Quadruplex	4.84E-12	2.88E-13	4.84E-12	2.88E-12	4.84E-12	2.88E-11

Table 3. VG Failure Probability Due to Attrition and Near-Coincident Faults - 100Hz, 10Hz, and 1Hz Recovery Rate

3.4. Effect of Recovery Rate on AFTA Mission Availability

Variations in the fault recovery rate has no effect on AFTA Mission Availability.

4. Updated Performance Model

The Detailed Design phase resulted in the implementation of a hard real-time Ada run time system which includes a multitasking rate group dispatcher, interprocess message passing functions, FDIR, and Input/Output Services. In order to construct a plausible performance model it is necessary to empirically measure the execution time of these functions. When their execution times vary, it is necessary to determine the parameters which affect this variance and determine the sensitivity of execution time to such parameters.

This section includes preliminary performance measurements of the rate group dispatcher, context switch times, message queuing and retrieving (with and without FDIR), and FDIR functions. All measurements were taken on an AFTA RTS which was interfaced to the Network Element Simulator (as opposed to hardware-implemented Network Elements). Before the detailed performance measurements are presented, an review of the basic AFTA run time system functions is provided.

4.1. Overview of AFTA Ada RTS Scheduling

The AFTA is designed for hard real-time applications. A rate group scheduler has been selected as the primary scheduling paradigm for the Ada RTS. This section discusses the rationale for the selection of this paradigm and illustrates its use to achieve hard real-time response for periodic and aperiodic hard real-time tasks.

Hard real-time schedulers must ensure that task executions, inter-task interactions, and interactions between the tasks and the outside world are predictable and deterministic, with guaranteeable worst-case response time. The means for validating this guaranteed response time must be an integral part of the scheduling paradigm. The scheduling paradigm should exhibit formal tractability to facilitate its formal specification and verification to reduce the occurrence of scheduler design and implementation errors. The scheduler should enforce the notion of "separation of concerns" to permit the combinatorially explosive validation of a complex application to be accomplished via the more tractable option of validating its constituent parts and their interactions. Guaranteeing these properties is often in direct conflict with programming and maintenance ease. An engineering tradeoff must be performed keeping in mind the disastrous ramifications of failure to meet a hard real-time deadline and the high life-cycle cost of software maintenance.

Relevant developments influencing the design rationale of the AFTA scheduler include Rate Monotonic Scheduling [Liu73], the MARS (MAintainable Real-time System) [Kop89], the

Reliable Computing Platform (RCP) [DiV90], and the NASA Space Transportation System General Purpose Computer (GPC) ([Car84], [Han89]).

4.1.1. Task Scheduling on a Single Virtual Group

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

In a rate group paradigm tasks executing on each VG in the AFTA are characterized by an iteration rate. In the AFTA, these rates are nominally 100, 50, 25, and 12.5 Hz, corresponding to rate group identifiers R4, R3, R2, and R1, respectively. A rate group frame duration is the inverse of the rate group iteration rate; thus the R4, R3, R2, and R1 frames are 10, 20, 40, and 80 ms in duration, respectively. All frame boundaries are determined by crystal oscillator-controlled interrupts. The frequencies and number of rate group frames are readily changed as the application dictates. Frames executing on different VGs in the AFTA need have no particular phase relationship with each other, although a desired phase relationship among certain frames may be enforced in some applications using a multi-VG rate group phasing method described below.

Within a particular rate group frame, tasks are scheduled using a nonpreemptive static schedule. When scheduled, a task executes to self-suspension. The exact time of execution of a particular task in the rate group frame will be in general unknown to the application programmer, and interactions between RG tasks and other entities occur only at RG boundaries, similar to the MARS temporal encapsulation concept. Instead, AFTA guarantees that all tasks within a rate group will be executed in the order specified by the application programmer sometime within the appropriate rate group frame. Figure 6 illustrates the basic idea of a single rate group.

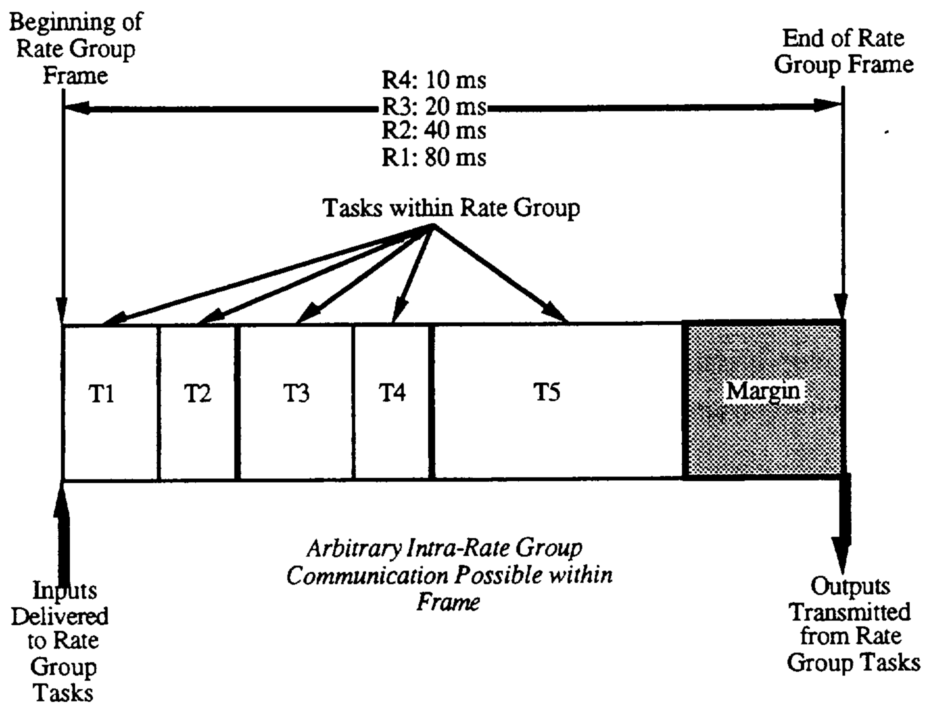
Beginning of Rate Group Frame
R4: 10 ms
R3: 20 ms
R2: 40 ms
R1: 80 ms
End of Rate Group Frame
Tasks within Rate Group
T1
T2
T3
T4
T5
Margin
Arbitrary Intra-Rate Group Communication Possible within Frame
Inputs Delivered to Rate Group Tasks
Outputs Transmitted from Rate Group Tasks

Figure 6. Rate Group Frame - Programming Model

To achieve multi-rate group execution on a VG, lower frequency rate group tasks are interrupted on a periodic basis to allow the higher-frequency rate groups to execute (Figure 7). The interruption process is transparent to the application programmer.

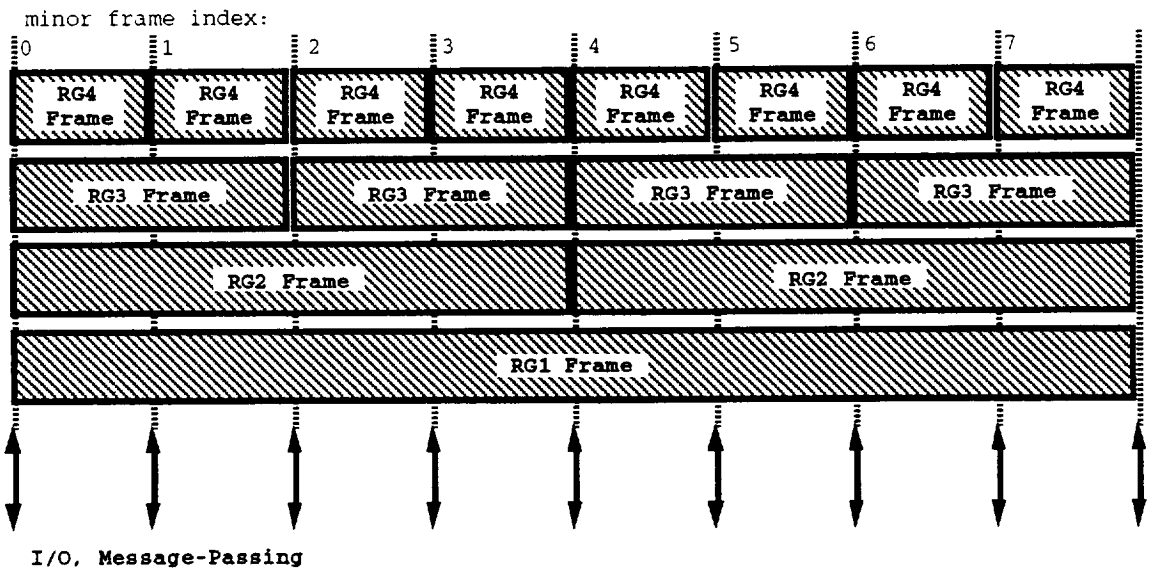

Figure 7. Architecture of RG Frames on a Single VG

Task overruns are detected by the rate group dispatcher at the end of each RG frame. Since all tasks within a frame nominally execute to self-suspension, the rate group dispatcher can detect a frame overrun by checking the suspension status of tasks which should have completed an iteration in the preceding rate group frame. Note that since the task which caused the overrun may itself have completed in the frame yet caused a subsequently-scheduled task to overrun, this technique does not conclusively identify which task is responsible for the overrun. Identification of the culprit task is achieved by comparison of the actual measurement of each task's execution time with its predicted execution time (note that this information is already needed for construction of the task schedule). Several overrun handling options exist and must be selected on a task-specific basis. Examples include aborting or restarting the culprit task, or resuming the preempted task from its preemption point at the start of its next RG frame.

Rate group scheduling may be viewed as a compromise between dynamic preemptive and static non-preemptive scheduling. Within a rate group, a static nonpreemptive schedule is followed. Higher frequency rate groups preempt lower frequency rate groups in a variant of rate monotonic scheduling [Liu73] modified for task suites having harmonic iteration frequencies. Because they interact only on frame boundaries, the set of rate groups may be viewed as a temporally encapsulated set of nonpreemptive tasks which may be formally treated independently.

4.1.2. Intertask Communication

All communication to tasks within a rate group is delivered and made available to the rate group tasks at the beginning of their rate group frame. All communication emanating from tasks within a rate group is queued within the rate group frame and transmitted at the end of that rate group frame. All messages not read by a RG task by the end of its frame can either be retained or deleted, with appropriate notification given to the recipient task. All communication emanating from a non-rate-group task is queued and transmitted on the frame boundary immediately after the one in which either (1) all copies of the task have requested transmission of the message, or (2) a majority of the copies of the task have requested transmission of the message and a user-defined timeout has expired.

4.1.3. Overview of Minor Frame

A simplified description of the sequence of events occurring within a minor frame of a single VG is depicted in Figure 8. The frame begins with a Frame Timer interrupt which is generated by a crystal oscillator resident on each member of the VG. Immediately after the Frame Timer interrupt, the VG synchronizes its members using a synchronizing act as described in the AFTA Conceptual Study Final Report, and sets up the Frame Timer interrupt for the next minor frame. This reduces the skew with which the members of the VG receive the next Frame Timer interrupt to the Network Element's post-synchronization skew plus the crystal oscillators' drift over the frame.

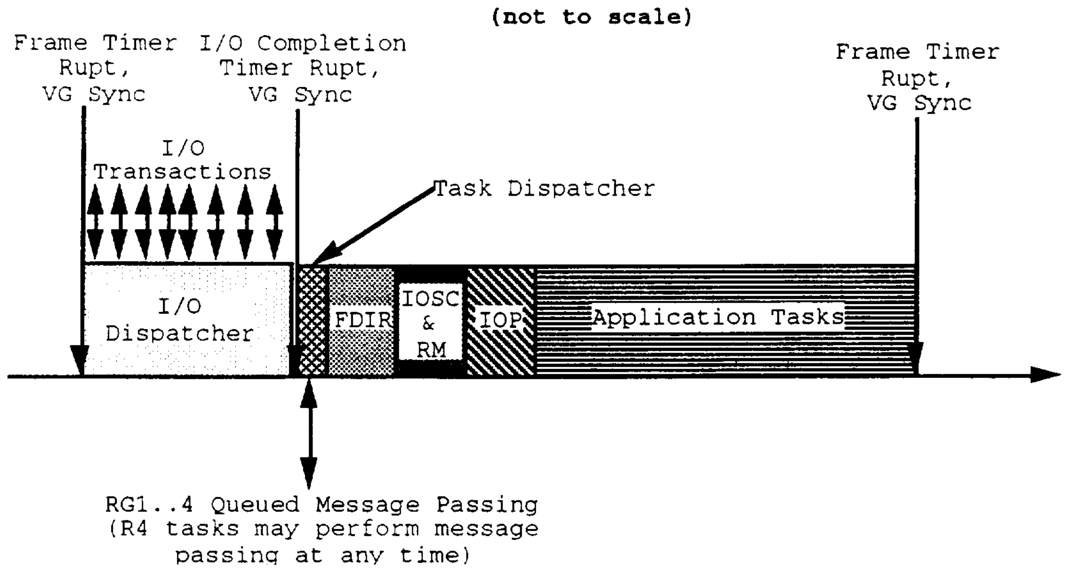
(not to scale)
Frame Timer Rupt, VG Sync
I/O Completion Timer Rupt, VG Sync
Frame Timer Rupt, VG Sync
I/O Transactions
Task Dispatcher
I/O Dispatcher
FDIR
IOSC & RM
IOP
Application Tasks
RG1..4 Queued Message Passing (R4 tasks may perform message passing at any time)

Figure 8. Overview of Minor Frame

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

After the I/O Completion interrupt another VG synchronization is performed by the RG dispatcher, and messages previously queued for transmission by rate group tasks which completed in the prior frame are transmitted to the Network Element. Messages are also read from the NE to the VG at this time. The FDIR task is scheduled after message passing, followed by the I/O Source Congruency and Redundancy Manager and I/O Processing tasks. The I/O tasks are responsible for transmitting single-source input data from one member of the VG to the others, I/O Controller/Device error processing, and deriving and formatting a known good copy of redundant input data for delivery to the destination application task. The I/O Processing task is also responsible for transmitting predetermined input data from one VG to another.

After the I/O tasks execute, the application tasks are scheduled and execute according to the rate group scheduling paradigm until the next Frame Timer interrupt occurs.

4.1.4. Aperiodic Hard Real-Time Task Scheduling

The AFTA scheduler supports the execution of event-triggered hard real-time aperiodic tasks by statically assigning the processing associated with each given event with an RG. An appropriate RG is determined a priori by the maximum allowable time between the occurrence of the event and the VG's output response. Events may of course occur at any time. The AFTA Input/Output System Service (IOSS) is scheduled at the beginning of each frame and is responsible for reading the status of any events to be processed in subsequent frames. Thus there is at most one minor frame's latency between the time of an event's occurrence and the time at which the IOSS processes the event for delivery to the destination task. An event processing task may be assigned to rate groups 1 through 4, in some cases preempting iterative tasks as outlined below. Multiple event processing tasks may be scheduled on a VG. The following table illustrates the maximum event response time as a function of the RG containing the event processing task.

Rate Group	Maximum Event Response Latency, # Minor Frames	Maximum Event Response Latency, ms (10 ms Minor Frame)
1	16	160
2	8	80
3	4	40
4	2	20

Table 4. Maximum Event Response Latency vs. Rate Group

Figure 9 shows an event occurring in frame 0 of a rate group schedule. If the event processing task is in R4, then the response from the event is delivered at the end of frame 1. If the task is in R3, the response is delivered at the end of frame 3. If the task is in R2, the response is delivered at the end of frame 7, and if the task is in R1, the response is delivered at the end of frame 7 of the subsequent major frame (not shown in the figure).

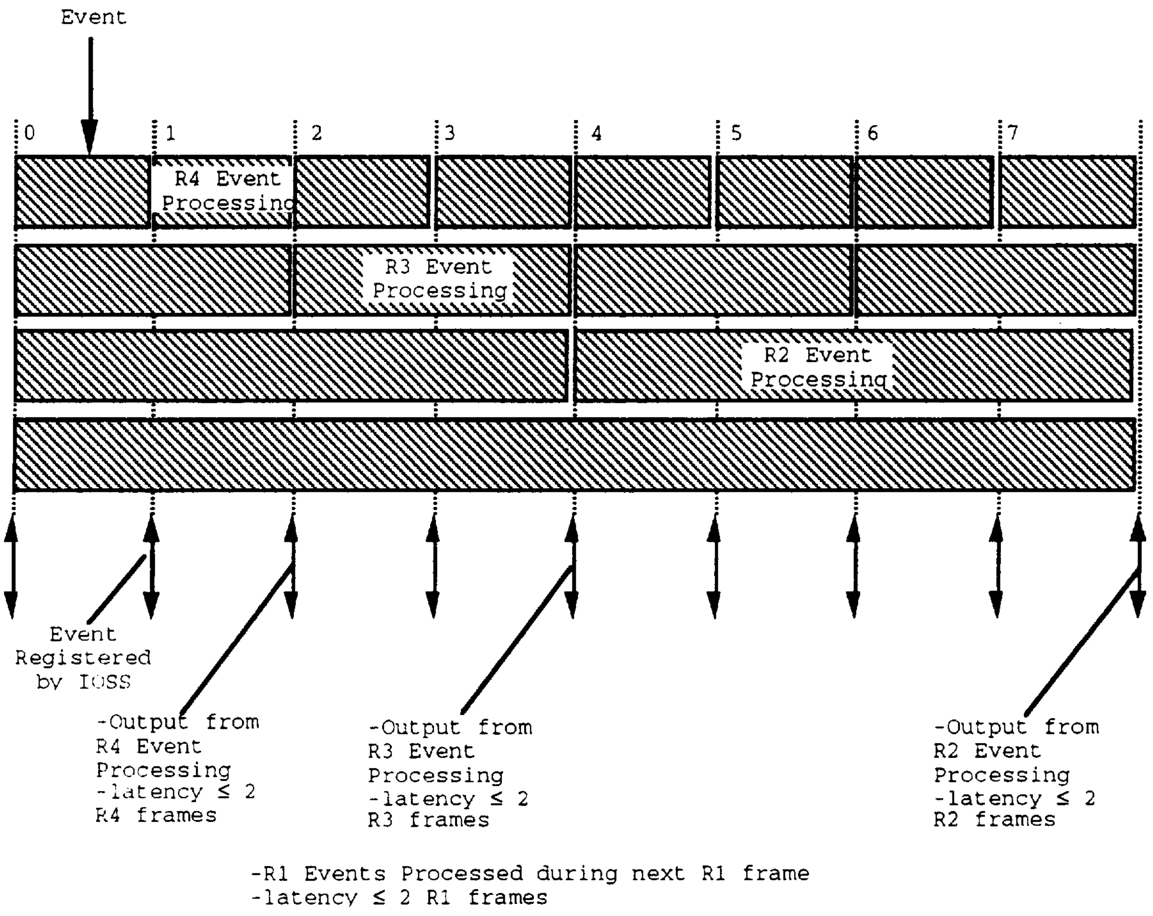
Event
0
1
2
3
4
5
6
7
R4 Event Processing
R3 Event Processing
R2 Event Processing
Event Registered by IOSS
-Output from R4 Event Processing -latency ≤ 2 R4 frames
-Output from R3 Event Processing -latency ≤ 2 R3 frames
-Output from R2 Event Processing -latency ≤ 2 R2 frames
-R1 Events Processed during next R1 frame -latency ≤ 2 R1 frames

Figure 9. Scheduling of Event-Triggered Hard Real-Time Aperiodic Tasks

Hard real-time event processing tasks are assigned to rate groups and are scheduled based on the occurrence of the events they are designed to handle. The arrival of a high-priority event and the consequent scheduling of an event processing task may perturb the timing of periodic tasks. Several options exist for gracefully scheduling event-triggered hard real-time aperiodic tasks.

One may validate the task suite's execution time upper-bound in the presence of all "valid" event combinations. The advantage of this approach is predictability and validatability for foreseen event suites. The programmer need not worry about frame slippage due to event-based preemption. The disadvantages are potential poor processor utilization, undefined or unpredictable behavior should an unforeseen event suite occur, and lengthy validation.

Alternatively, depending on the event to be processed, one can deschedule one or more selected periodic tasks of equal *or higher* iteration rate. After event processing completion, the descheduled iterative tasks must be rescheduled for resumption. It is critical that, regardless of the selected option, periodic and aperiodic hard real-time task aggregate execution times must be validated to meet all real-time constraints.

4.1.5. Aperiodic Non-Real-Time Task Scheduling

Aperiodic tasks which do not have hard real-time constraints are executed after all rate group tasks (including aperiodic hard real-time tasks) have been executed. There may be several non-real-time aperiodic tasks running on a VG and they may be scheduled arbitrarily (unprioritized round-robin, multi-level prioritized, etc.).

4.1.6. Execution of RGs on Multiple VGs

Due to the parallel nature of AFTA, different VGs will execute different RG task suites. Mapping a multi-VG multi-RG task suite onto multiple VGs can be performed in a straightforward manner using application task-to-parallel processor mapping technology embodied in an integrated schedule generation and analysis tool. Many such tools have been built and are commercially available. Task suites are expected to change as a function of the mission mode and system state. This will give rise to multiple mappings. Each such mapping must be created using the schedule generation and analysis tool. Moreover, the valid transition sequences from one such mapping to another must be carefully defined and implemented so as to continuously meet real-time requirements during the transition period. We note that most tool designs do not appear to handle the generation and evaluation of transitions from one task mapping to another with respect to continuously meeting real-time constraints.

The rate group *phasing* describes the relationship between rate group frames on different VGs in an AFTA. This phasing can be selected to minimize nondeterminism due to contention for the shared Network Element communications media as described below.

Within the task configuration table, each task is assigned to execute in some rate group. The rate group determines the frequency at which the task will be executed, and the resulting rate group frame boundaries delimit the execution cycle of the task in accordance with temporal encapsulation. Tasks assigned to the same rate group will execute at the same frequency regardless of their hosting VG, but there may be a time difference between the start of their first and each subsequent rate group frame if the tasks are executing on different VGs. This phasing could, for example, be caused by the completion of system initialization at different times on different VGs. An example phasing of the frames for tasks in a given rate group on multiple VGs is shown in Figure 10.

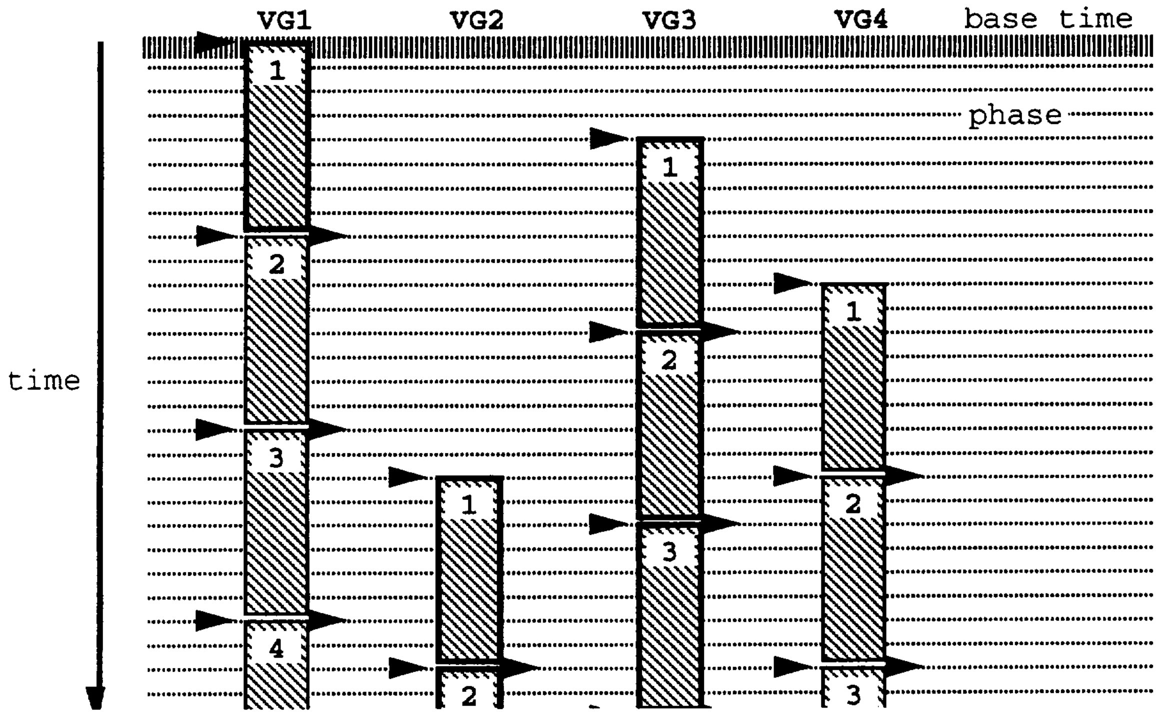

Figure 10. Phasing of RG Frames on Multiple VGs

In the example, the first rate group frame on VG1 starts at some "base time" and the first rate group frames on the remaining VGs are delayed. The interval between the base time and the start of its first rate group frame is a VG's phase delay. The phase delay is important because it determines the relationship between the frame in which messages are sent by one VG and the frame in which they are received by another. It also effects the degree and predictability with which the different VGs contend for the Network Elements and other physical resources. Predictability with respect to a single VG is enhanced by the message passing restrictions in the rate group tasking paradigm. In the paradigm, a task's queued messages are only sent and its received messages are only made available at its corresponding rate group frame boundary. This is indicated in the figure by the arrows at the frame boundaries. An example from the figure is the messages transmitted after the first frame from VG1. They will be received at the start of the first frame on VG2 and VG4, but will not be received until the start of the second frame on VG3. This relationship of sending frame to receiving frame will remain constant for subsequent frames if the phasing does not change.

The time management service (embodied in the rate group dispatcher task) has been designed to achieve a fixed phase between the RGs of different VGs by locking a VG's phase to the system time maintained by the Network Element. There still remains inherent float because of the variability of the interval from the start of the frame to when any given message will be sent or read. This float is increased when VGs which share a Network Element have the same phase delay or their delays differ by an integer number of minor frames. This is because the VGs are then forced to compete for access to the Network Element to send and read their messages at their frame boundary. For this reason the simplest phasing of a zero phase delay for all VGs is not recommended. The phase field in the VG configuration table is provided to specify the desired phase delay for each VG.

4.2. Performance Models

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

Because of their importance to AFTA system performance, analytical models for both the operating system overhead and Network Element contention are developed. This section presents descriptions of each model.

4.2.1. Operating System Overhead Model

This section gives a general overview of the model for the overhead associated with AFTA operating system tasks. Figure 11 reviews the operating system tasks associated with each minor frame.

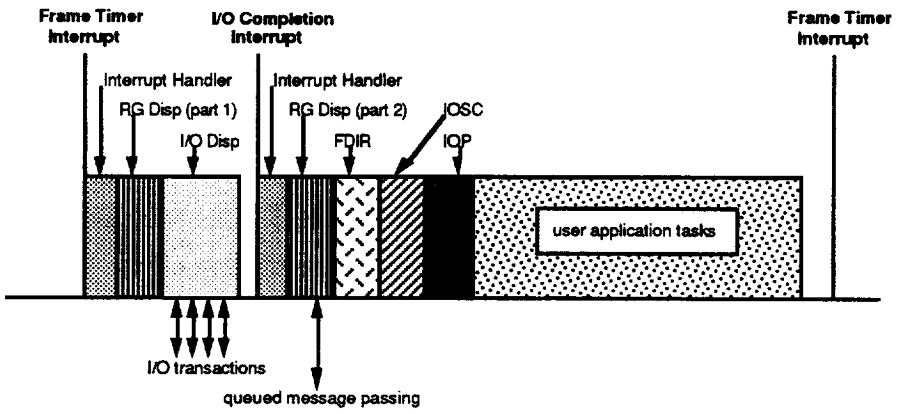

Figure 11. Overview of Minor Frame

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

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

A description of each of these eight overheads follows.

4.2.1.1. Interrupt Handler (IH_1) Overhead

The overhead associated with the first interrupt handler (IH_1) is given by the following equation:

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

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

4.2.1.2. Rate Group Dispatcher - Part One (RGD_1) Overhead

The time needed to execute the first part of the rate group dispatcher (RGD_1) can be summarized with the following equation:

$$RGD_1 = \textit{(time to update congruent time)} + \textit{(time to check for } RGD_2 \textit{ overrun)} + \textit{(time to check for task overruns)} + \textit{(time to set up next RG interval)} + \textit{(time to schedule IOD)}$$

With the exception of checking for task overruns, all the components of RGD_1 are constant. Checking for task overruns is a function of the number of tasks which were scheduled to suspend themselves during the previous minor frame.

4.2.1.3. IO Dispatcher (IOD) Overhead

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

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

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

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

4.2.1.4. Interrupt Handler (IH_2) Overhead

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

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

Even though both instances of the interrupt handler are modeled by the same equation, in general the overheads associated with IH_1 and IH_2 are different. This is because the time to scoop messages will vary with the number of packets present in the receive queue for the processor. Typically, the time interval between the occurrence of IH_1 and IH_2 is less than the time duration from IH_2 to the next occurrence of IH_1. This implies that more packets may arrive in the receive queue during the interval from IH_2 to IH_1, and therefore the message scoop time should generally be longer for IH_1 than IH_2.

4.2.1.5. Rate Group Dispatcher - part two (RGD_2) Overhead

The execution time for the second part of the rate group dispatcher (RGD_2) can be quantified as follows:

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

Most of the constituents of RGD_2 listed above involve simple housekeeping chores and have constant execution times. The three areas of interest are the time to send queued messages, the time to update queues, and the time to schedule rate group tasks. The *time to send queued messages* is a function of the number of tasks that suspended themselves during the previous minor frame and the number of message packets that each task had enqueued since the last time its queue was sent. The time to send queued messages also varies with the amount of contention for NE service. The OS overhead model assumes no contention; the effect of contention on the `send_queue` time is explored in Section 4.2. The *time to update a task's queue* is a function of the number of packets received and the number of packets read since the last time the queue was updated. The *time to schedule the RG tasks* is a function of the number of RG tasks that are to be scheduled this minor frame.

4.2.1.6. Fault Detection Identification and Recovery (FDIR) Overhead

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

FDIR = (time to enqueue message to System FDIR task)

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

4.2.1.7. IO Source Congruency Manager (IOSC) Overhead

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

IOSC = (time to exchange input data among VG members)

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

4.2.1.8. IO Processing Task (IOP) Overhead

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

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

Note that there are four IOP tasks, one for each rate group.

4.2.1.9. OS Overhead Summary

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

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

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

$$OH = Constant + f(number\_of\_tasks) + g(number\_of\_message\_packets) + h(amount\_and\_type\_of\_IO)$$

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

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

4.2.2. Contention Model

The second model developed to analyze AFTA system performance examines the contention among PEs for NE services. Each processor sends its queued message packets during the second part of the rate group dispatcher. If several PEs are sending packets at the same time, they must wait for the NE, which services the PEs in round-robin fashion.

This contention results in performance delays because the PEs busy-wait (i.e., continuously poll the NE to see if it is ready) for each packet to be serviced before enqueueing the next one. Since performance delays can be critical in real-time systems, it is important to understand the effects of this contention on system performance by developing an analytical model.

This section describes the model developed to analyze the delay times associated with contention. This contention model can be used to determine the busy-wait delay for each PE as a function of the phasing of the eight PEs.

4.2.2.1. The Model

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

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

4.2.2.1.1. Processing of Message Packets

Tasks that wish to send messages must first decompose each message into 64-byte packets and place them in a queue in the PE's local memory. When the rate group dispatcher (part two) executes during the following minor frame, these message packets are sent to the NE one at a time during execution of the `send_queue` procedure. The queued packets are sent to the PE's transmit queue, which is located in a dual-port RAM memory shared between the PE and the NE. This is shown in Figure 12. Each of the possible eight PEs connected to a NE has its own transmit queue. The packets are sent one at a time because the capacity of the queue is only one packet. The PE can not send a second packet to the transmit queue until the first one has been removed by the NE. If the PE has more than one packet to send and the transmit queue is full, the PE must wait until the NE empties the transmit queue before the PE can transfer the next packet to the transmit queue.

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

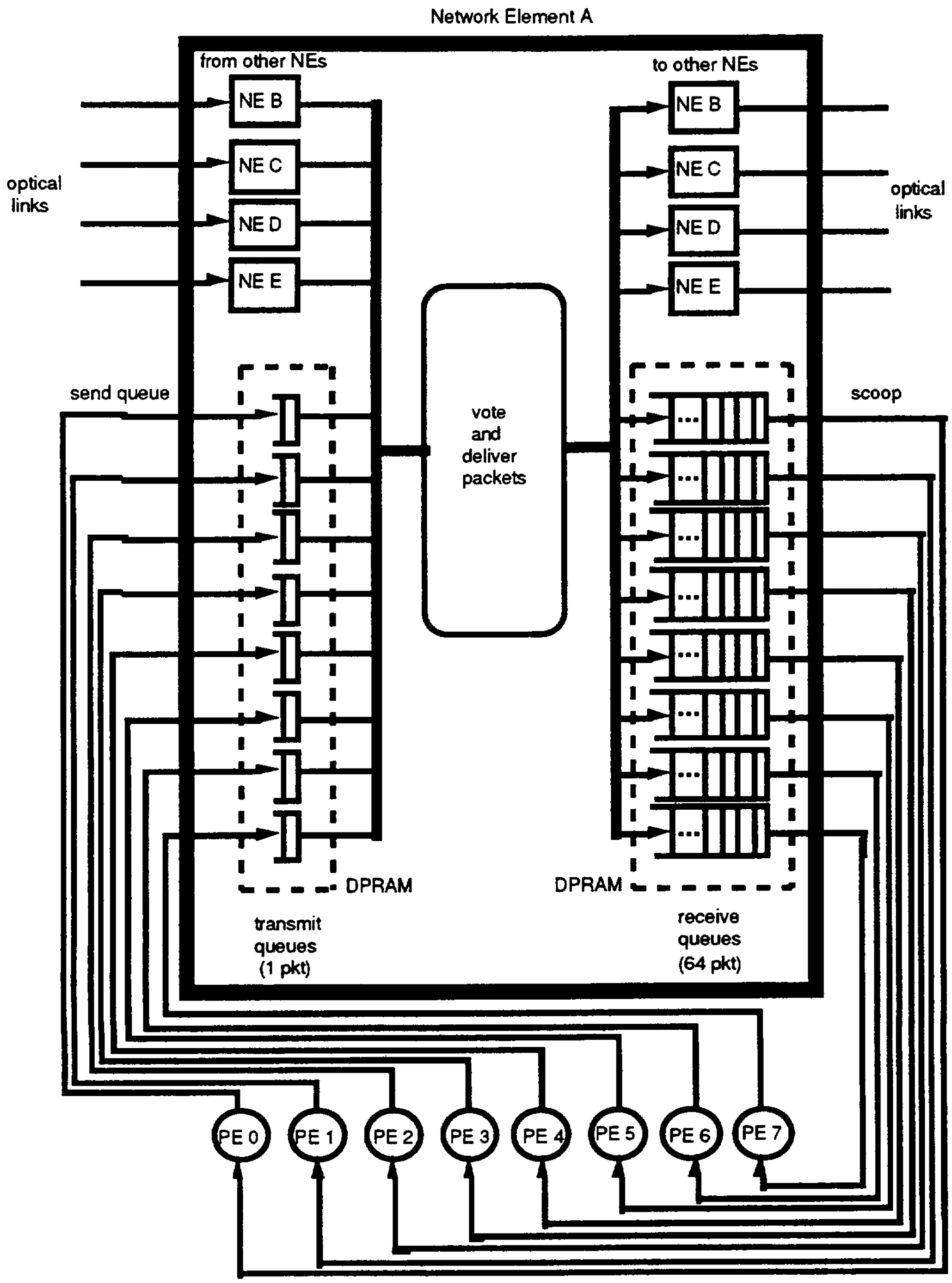

Figure 12. Message Packet Processing

Once the NE is aware of the arrival of a packet, it can begin to process it. The processing done by the NE depends on the class of the packet being transmitted. A Class 0 message requires minimal processing time since no data is involved. Class 1 messages (voted messages) are typically the most common type of message. Processing a Class 1 packet involves receiving redundant copies of the packet from the other PEs (connected to different NEs) in the virtual group. These copies are voted, syndrome information is attached, and the voted copy is delivered to the destination PE. A Class 2 packet (source congruency message) undergoes a two-round exchange with the other NEs before voting and delivery.

After processing a packet, the NE delivers the packet by placing it in the appropriate receive queue, as shown in Figure 12. Each PE has its own receive queue located in the dual-port RAM shared among the NE and its eight PEs. The capacity of each receive queue is 64 packets. Packets are transferred to the destination PE via a `scoop` call, and the packets are reassembled into messages when the destination VG executes a `retrieve_message` system call.

4.2.2.1.2. Contention for NE Services Among Two or More PEs

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

An example demonstrating how the busy-wait time can vary is given in Figure 13. The time needed for a PE to transfer a packet from its local memory to the transmit queue is constant (approximately 57 μsec). As shown in Figure 13, the NE will be informed that PE_0's transmit queue is full once the NE has processed the SERP containing this information. In the figure, the transmit queue was filled at time t1, and the SERP processing was completed at time t2. Once the SERP is processed, the NE is able to process the packet, and the PE is then able to transfer the next packet when the packet processing is finished and the transmit queue is cleared at time t3. Thus, PE_0 had to wait from time t1 to time t3. Notice that once PE_0 has filled the transmit queue a second time, it has a much longer delay before it can transfer a third packet. This is because when the queue is filled at time t4, the NE is busy processing a packet from PE_2 and thus can not immediately empty PE_0's transmit queue. It is not until time t6 that the NE finishes processing the SERP that indicates PE_0's queue is full. Since the NE services the PEs in round-robin fashion, PE_0 will have its queue emptied at time t7. Figure 13 shows that the t4-t7 time interval is greater than the t1-t3 time interval. The amount of time spent by PE_0 waiting for NE access increased because it had to contend with other PEs for NE service. Figure 13 also indicates the phasing in the system for this example. The phasing between two PEs is the difference in time between the start of each of their minor frames.

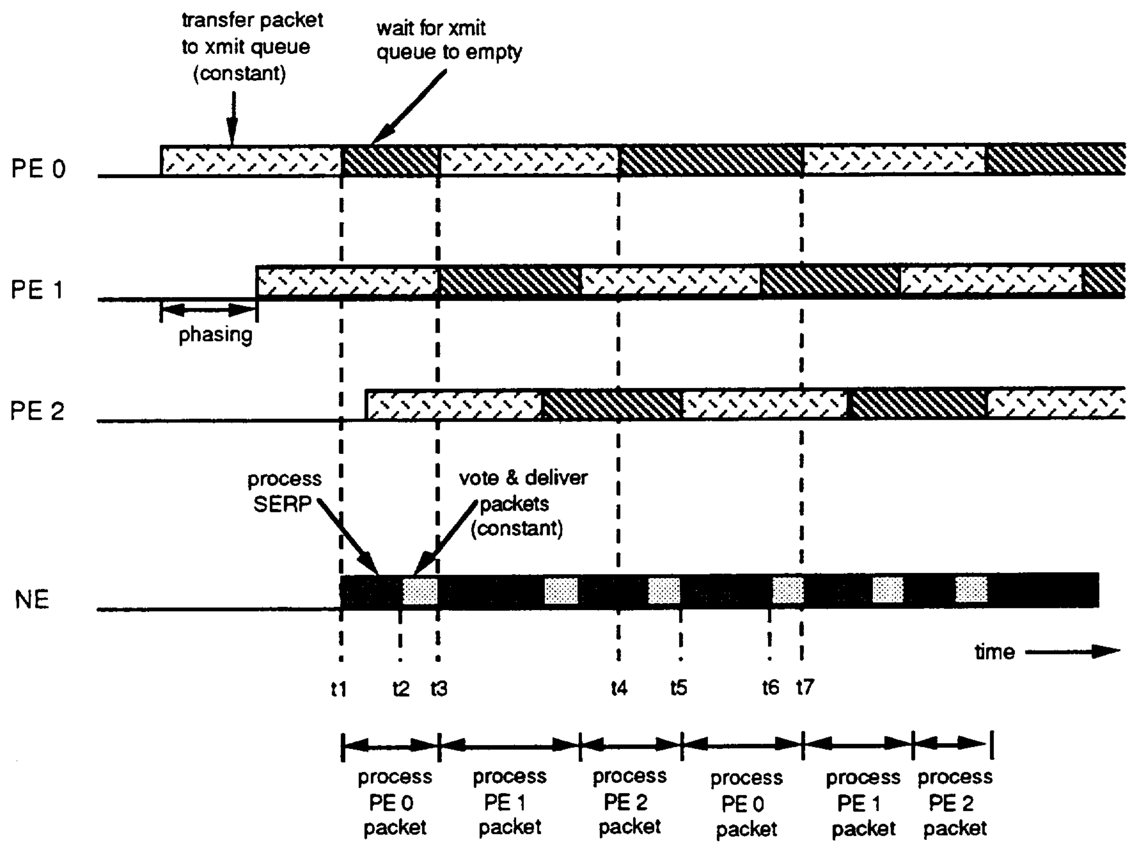

Figure 13. Contention Timeline

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

4.2.2.1.3. Simplifying Assumptions

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

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

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

Operating system overheads are identical for each PE for each frame. The operating system overhead generally varies with the minor frame number. For example, minor frame 0 usually has the largest OS overhead since all tasks, regardless of their rate group, have suspended themselves and are ready to send queued messages. Minor frame 1 usually has a minimal overhead since only RG4 tasks can send their queues. We assume the OS overhead variance is negligible, and that it is identical for each minor frame for each PE. Therefore, each minor frame on each PE appears like every other minor frame. Without this assumption, the time within the minor frame when RGD_2 was executed (and thus when the queued packets can be sent) would vary from frame to frame and would be a function of the number of tasks, the distribution of tasks among the four rate groups, the number of packets enqueued by each task per minor frame, and the amount and type of IO performed by each task per frame.

The phasing among the eight PEs is constant. The phasing between two PEs is the difference in time between the start of each of their minor frames. In Figure 14, an example of phasing among eight PEs is given. The phasing between PE_0 and PE_1 is indicated in the figure. We assume that the phasing from one PE to its neighbor is the same. The amount of phasing is important because it determines how much overlap there is when PEs are performing a `send_queue` call. The worst case, in terms of contention, would be zero phasing; then, all PEs would be sending their queues at the same time, and contention would be maximized. During simulation, the phasing is varied to note its effect on contention.

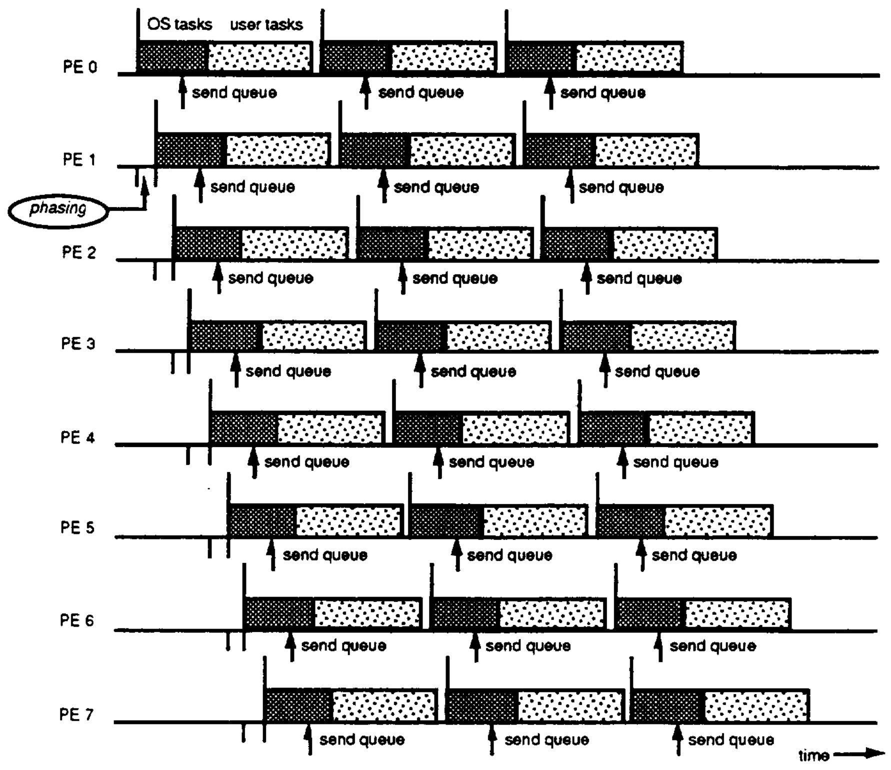

Figure 14. Phasing Among PEs

There is no contention for the VMEbus which connects the PEs to the NE. One detail that has been ignored so far is the bus connecting the PEs to their NE. The prototype AFTA uses VMEbus to connect the PEs and their NE, and it is possible that the PEs may have to contend for the VMEbus while transferring their packets from local memory to the transmit queue. We assume that there is no contention among PEs for use of the VMEbus. Consider the worst case scenario for data traffic over the VMEbus (zero phasing among the eight PEs). In this case, all eight PEs attempt to send a 64-byte packet over the VMEbus at the same time. Empirical performance data show it takes approximately 60 μsec for a single PE to transfer a packet from local memory to the transmit queue. Therefore, at worst 512 bytes are being sent over the VMEbus in a 60-μsec period, which corresponds to a data rate of 8.5 Mbytes/sec. The VMEbus has been rated at 40 Mbytes/sec, so the worst case amount of VMEbus traffic only uses about 21% of the available bandwidth. As a result, we consider the assumption of no VMEbus contention valid.

4.2.2.2. Contention Simulation

An example timeline for the contention model is given in Figure 15. In this example, three PEs each send two packets to the NE for processing.

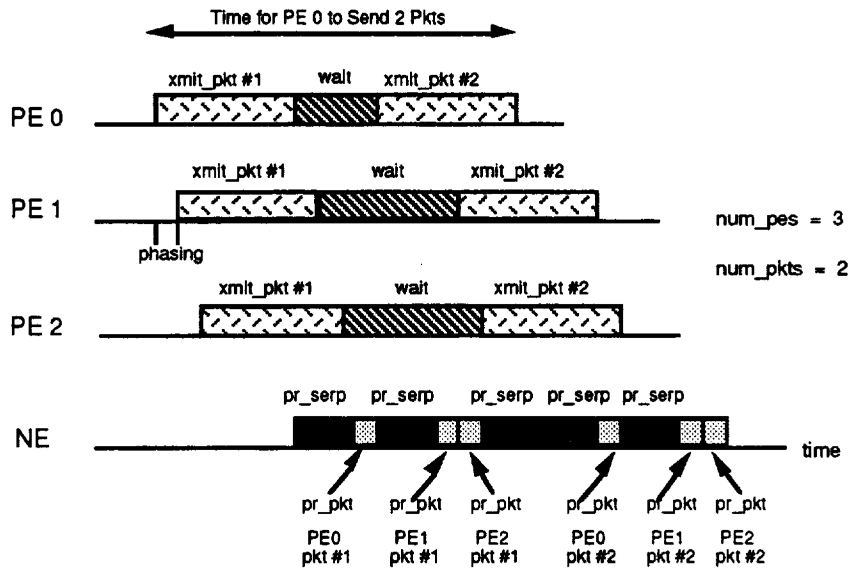

Figure 15. Contention Model Timeline

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

xfer_pkt	This is the time it takes a PE to transfer a packet from its local memory space to the transmit queue located in the dual-port RAM shared by the PE and NE. The default value is 60 µsec.
pr_serp	This is the time it takes a PE to process a SERP. This value is assumed to be constant, and the default value is 25 µsec.
pr_pkt	This is the time needed by the PE to process a packet. Processing a packet includes voting redundant copies of the packet and delivering the voted packet to its destination. The default value is 10 µsec.
num_pkts	This is the number of packets each PE sends during each frame. For simulation purposes, all PEs send the same number of packets. The default value is 10 packets per PE per frame.
num_pes	This is the number of PEs connected to the NE. The default value is 8 PEs (the maximum possible).

phasing The phasing between two different PEs is the difference in time between the start of each of their minor frames. It is assumed that the phasing among PEs is constant, as shown in Figure 14. The default value for phasing is 0 μsec; this represents worst case contention.

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

4.2.2.3. Results of the Simulation

Of the parameters listed in the previous section, some are of more interest to application engineers than system designers. Application engineers are concerned with the number of PEs connected to a NE, the number of packets send by each PE, and the phasing among the PEs; these are the parameters they control. Their goal is to minimize the time needed for a PE to send its packets within the time constraints of the application task. The effect of varying the number of packets sent by each PE is shown in Figure 16. By reducing the number of packets per PE the delay in sending the packets is reduced, and this is shown in the graph. For a given number of packets, different amounts of phasing can result in slight improvements in performance. However, the performance improvement is not very significant.

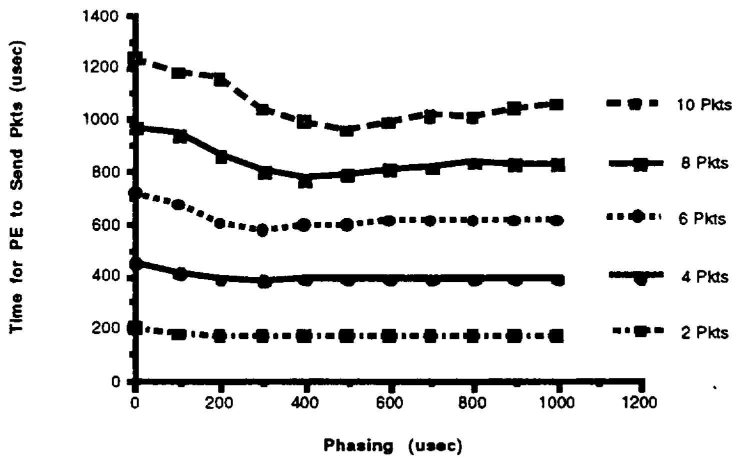

Figure 16. Effect of Varying Number of Packets and Phasing on Time to Send Message Packets

Another parameter of interest to the applications engineer is the effect of reducing the number of PEs connected to a NE. Simulation results showing the impact of varying the number of PEs on the time needed by each PE to send its packets are given in Figure 17. With small amounts of phasing, the number of PEs has some effect on the time needed to send packets. However, the improvement is not large. Consider the case of no phasing. With eight PEs, the time delay is 1230 µsec, but reducing the number of PEs to four only decreases the time delay to 1100 µsec. Reducing the number of PEs by 50% results in an improvement of only 10.6% in performance. It is also interesting to note that as the phasing increases, the effect of reducing the number of PEs becomes negligible.

Though not shown in Figure 17, the time to send packets for one PE is of interest because it indicates the extent to which contention can increase system overhead. With only one PE connected to a Network Element, no contention can occur; the simulation predicts a 1035 µsec time delay for one PE to send its queued packets. The worst-case contention occurs when eight PEs are connected to one NE, and the amount of time needed for a PE to send 10 packets in this configuration is 1230 µsec. Therefore, contention can increase the amount of overhead in sending packets by 18.8% compared with the case when no contention occurs.

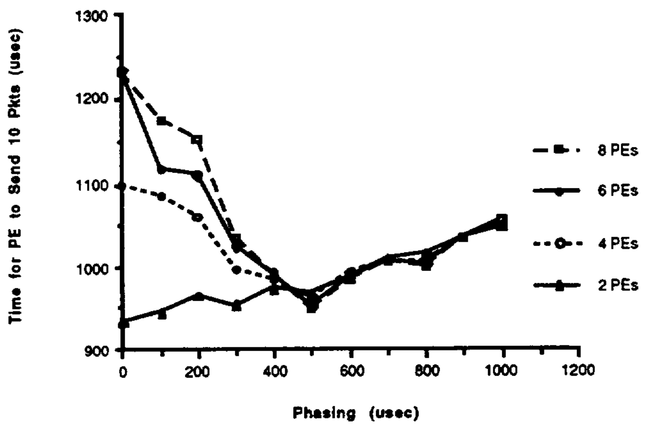

Figure 17. Effect of Varying Number of PEs and Phasing on Time to Send Message Packets

System designers are also interested in ways to reduce the amount of time spent sending packets. The parameters controlled by system designers include the time it takes the NE to process a packet, the time it takes for the NE to process a SERP, and the time needed by a PE to transfer a packet from its local memory space to the dual-port RAM shared by it and the NE. Figure 18 shows the effect of varying the process packet time and varying the transfer packet time. For a transfer packet time of 60 µsec, an 80% reduction in process packet time (from pr_pkt = 10 µsec to pr_pkt = 2 µsec) results in a 26.5% performance improvement.

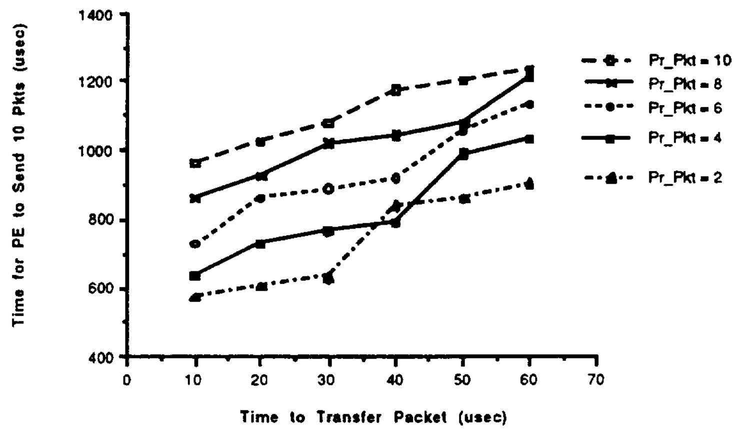

Figure 18. Effect of Varying Process Packet Time and Transfer Packet Time on Time to Send Message Packets

The system designer also determines the time needed by the NE to process a SERP. Figure 19 presents the simulation data for different values of the process SERP time. As expected, reducing the process SERP time reduces the delay needed by a PE to send its packets. With a transfer packet time of 60 μsec, a reduction in the process SERP time of 80% (from pr_serp = 25 μsec to pr_serp = 5 μsec) results in a performance improvement of 24.5%. This is approximately the same effect as varying the process packet time from 10 μsec to 2 μsec.

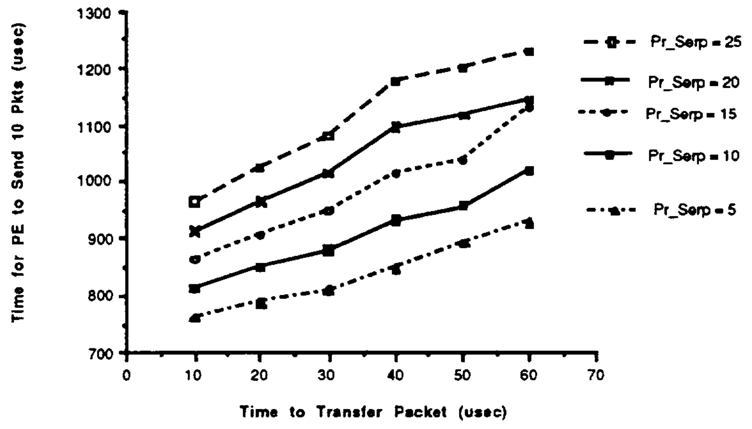

Figure 19. Effect of Varying Process SERP Time and Transfer Packet Time on Time to Send Message Packets

The goal of both the applications engineer and the system designer is to reduce the amount of time needed for a PE to send its message packets, even when contending with other PEs for NE service. To find out which parameter has the greatest single impact, each parameter was decreased by 50% of its default value. The results are given in Figure 20. This graph shows that the largest improvement in performance for a given number of packets is obtained by reducing the transfer packet time by 50%. This implies that if effort can only be spent reducing one parameter, it should be spent reducing the transfer packet time. The transfer packet time can be reduced by using direct memory access or by using a faster bus.

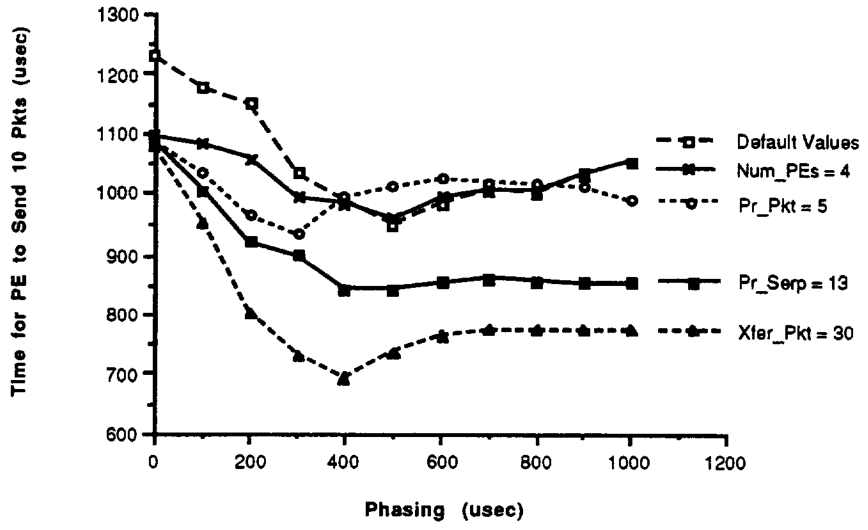

Figure 20. Effect of Reducing Each Default Parameter by 50% on Time to Send Message Packets

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

4.3. Performance Measurement Methodology

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

Raw performance data is collected through the use of software probes. The probes are a software routine which records relevant system information, including the value of the system clock. These probes are placed around or directly inside the code of the operating system procedures of interest. During execution, the probes are activated along with the OS procedure of interest. The probes record execution times and other parameters of interest in the processor's local memory. The real-time AFTA system is not suited to perform the analysis of this raw data, so the data are transferred to a host VAX computer for reduction and analysis. The AFTA IO System Services are used to move the data, via an Ethernet link, from the PE to the host. Figure 21 shows the path the performance data take from initial storage in the debug log to final analysis on the VAX.

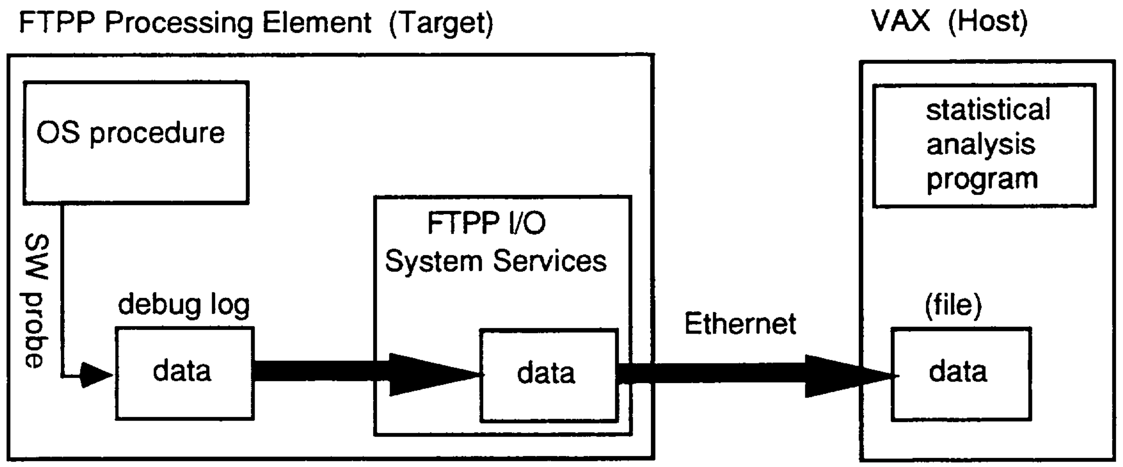

Figure 21. Performance Measurement Overview

4.3.1. Software Probes

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

4.3.1.1. Description of Data Recorded by Software Probes

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

- label field
- parameter field
- timestamp field

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

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

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

4.3.1.2. Example of Software Probe Use

As an example of how the debug log entry fields are used to measure system performance, consider a method to determine the length of time for the operating system to queue a message for delivery. Software probes are inserted in an application task just prior to, and immediately after, calling the queueing procedure. This is illustrated in Figure 22.

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

Figure 22. Placement of Software Probes

The software probe is activated by the call to the Ada procedure `debug_log`. Two parameters are passed during the call to `debug_log`. The first parameter is a number that will be stored in the label field of the debug log entry. For the first software probe shown in Figure 22, the hexadecimal number 1111 is used as the label (or tag). The label field for the second software probe (just after the queue_message procedure) contains 1112 hex.

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

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

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

Table 5. Representative Use of Debug Log Data Fields

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

4.3.2. Transfer of Data from the AFTA to the Host VAX

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

4.3.3. Data Analysis

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

4.3.3.1. Determination of Time Interval

In determining the time interval between two debug log entries, the analysis program uses the label field of the debug log entries to identify the data associated with each software probe. The user supplies the analysis program with the labels for each pair of appropriate software probes. For instance, in the queue message example, the pertinent labels are 1111 hex and 1112 hex. The analysis program searches through all the debug log entries stored in the VAX file, and saves entries that have the given labels. These saved entries are then paired, and their timestamp values are subtracted. This value gives the number of clock ticks that occurred between the activation of the pair of software probes. To convert this number to a time value, it is multiplied by the clock resolution, which is 1.28 μsec per clock tick. One final bit of processing is needed before determining the length of the time interval. The overhead of activating the software probe (the time it takes to make the debug_log procedure call) needs to be subtracted from the time interval value.

To determine the overhead for software probe activation, a number of debug_log procedure calls were sequentially executed. As shown in Table 6, there was a 22 µsec time delay between the activation of two software probes. This implies that the overhead that should be subtracted between a pair of consecutive probes is 22 µsec. This is the overhead value that would be subtracted for the queue message example because the probes are in consecutive debug log entries. However, sometimes other debug entries are located in between the two entries that are of interest. For example, suppose we want to measure the length of time it takes for a task to execute, and within that task is a queue message call that we also want to measure. The task measurement probes would not be consecutive entries in the debug log because the queue message measurement probes would be located between them. Since there are two nested probes between the task probes, the overhead associated with the queue message probes also needs to be subtracted from the time for the task. Therefore, the number of intervening probes must be counted, so the overhead for all these probes can be taken into account. Hence, 22 µsec should be subtracted as additional overhead for each intervening software probe activation.

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

Table 6. Overheads Associated with debug_log Procedure Calls

4.3.3.2. Statistical Analysis of Time Data

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

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

4.4. Performance Measurement Results

Using the methodology described above, empirical performance data for AFTA operating system overheads were collected. This section summarizes the measurements. Performance data for each of the operating system tasks are presented in the order in which they occur during each minor frame (Figure 11).

4.4.1. System Configuration

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

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

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

RG3 tasks (one)
- io_processing_task_rg3

RG2 tasks (one)
- io_processing_task_rg2

RG1 tasks (one)
- io_processing_task_rg1

4.4.2. Interrupt Handler Overhead

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

4.4.2.1. Scoop Message

The scoop procedure transfers message packets from a PE's receive queue in the Network Element's dual-port RAM to the PE's local memory space where they are reassembled into complete messages. The time to scoop messages is dependent on the number of packets to be scooped, as shown in Table 7.

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

Table 7. Scoop Message Execution Time as a Function of Number of Packets

4.4.3. Rate Group Dispatcher (Part One) Overhead

The primary functions of the first part of the rate group dispatcher (RGD_1) are to check for task overruns and to schedule the IO dispatcher for execution. Overall, the execution time for RGD_1 varies as a function of the minor frame number, as shown in Table 8. The reason for this variance is that different minor frames have a different number of rate groups that have reached their RG boundaries. When a rate group reaches its boundary, all tasks within that rate group should have completed their iterative cycle. RGD_1 ensures that all tasks that should have completed actually did, and the number of tasks to check depends on the number of rate groups that have reached RG boundaries.

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

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

Notice that RGD_1 executes longest during minor frame 0. This is because all rate group tasks have completed their iterative cycle at the completion of minor frame 7. Therefore, RGD_1 has to check for overruns of tasks in every rate group. RGD_1 has a minimal execution time during minor frames 1, 3, 5, and 7 because during those frames it only needs to check RG4 tasks for overruns.

The overall execution time for RGD_1 can be broken down into three main segments: (1) the time needed to record the congruent time value and to check for RGD (part two) overrun, (2) the time to check for rate group task overruns, and (3) the time needed to set up the next rate group interval and schedule the IO dispatcher.

4.4.3.1. Record Congruent Time Value, Check for RGD_2 Overrun

At the beginning of execution, RGD_1 records the congruent time value and then verifies that the second part of the rate group dispatcher (RGD_2) did not overrun during the previous minor frame. The time to accomplish these duties, as seen in Table 9, is constant and thus does not vary with the frame number or task distribution.

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

Table 9. RGD_1 Update Congruent Time Value and Check for RGD_2 Overrun Execution Time

4.4.3.2. Check for RG Task Overruns

RGD_1 determines whether any of the tasks that were to complete their iterative cycle and suspend themselves during the previous minor frame overran the frame boundary. The time needed to accomplish this is a function of the number of RG tasks that were scheduled to suspend themselves during the previous minor frame. This is shown in Table 10. This segment of RGD_1 is the only one that does not have a constant execution time.

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

Table 10. RGD_1 Check for Rate Group Task Overruns Execution Time as a Function of Number of Rate Group Tasks

4.4.3.3. Set Up Next RG Interval, Schedule IO Dispatcher

Before finishing execution, RGD_1 sets up the next rate group interval; this entails determining when the next interrupt should occur. RGD_1 then schedules the IO dispatcher to execute next. These duties are done every minor frame, and the amount of time needed to do them is constant for all minor frames. The execution times are summarized in Table 11.

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

Table 11. RGD_1 Set Up RG Interval and Schedule IO Dispatcher Execution Time

4.4.4. IO Dispatcher (IOD) Overhead

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

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

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

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

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

4.4.5. Rate Group Dispatcher (Part Two) Overhead

The primary functions of the second part of the rate group dispatcher (RGD_2) are to send queued message packets and to schedule rate group tasks for execution. A summary of the overall RGD_2 execution times, sorted by minor frame number, is given in Table 12.

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

Table 12. Overall Rate Group Dispatcher (Part Two) Execution Time as a Function of Minor Frame Number

As is evident from the large standard deviations in Table 12, RGD_2 execution times do not vary directly with the minor frame number. Unlike RGD_1, which only varied as a function of the number of tasks that suspended themselves during the previous minor frame, the dependencies of RGD_2 are more complicated. In particular, RGD_2 performance is related to the number of message packets that were enqueued during the previous minor frame. Since the number of enqueued packets can differ for a given application from one iteration to the next, it is not meaningful to examine RGD_2 execution times as a function of only the minor frame number.

It is more useful to break RGD_2 into several segments and then examine each segment separately. The following five sections describe the five major segments of RGD_2: update congruent time value and check for RGD_1 and IOD overruns; send queued message packets; update message packet queues; schedule rate group tasks; and increment minor frame number and set up IO interval for the next frame.

4.4.5.1. Update Congruent Time Value, Check for RGD_1 and IOD Overrun

At the beginning of each iteration cycle, RGD_2 updates the congruent time value used by each rate group and checks to see if either the rate group dispatcher (part one) task or the IO dispatcher task exceeded its execution time bound. The time to accomplish these duties is the same during each iteration of RGD_2. A summary of the execution time data is given in Table 13.

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

Table 13. RGD_2 Update Congruent Time Values and Check for RGD_1 and IOD Overrun

4.4.5.2. Send Queue

RGD_2 calls the `send_queue` procedure once for each task that suspended itself during the previous minor frame. `send_queue` transfers enqueued message packets from each PE's local memory space to the Network Element where they are processed and delivered. The execution time of each `send_queue` call is a function of the number of packets that were queued by that task, as shown in Table 14. Therefore, the total amount of time RGD_2 spends sending queued packets depends on the number of tasks that suspended themselves during the previous minor frame and on the number of packets enqueued by each task.

It is important to note that the data in Table 14 was collected using only one Virtual Group. Since only one PE was connected to the NE, no contention for NE service occurred. Therefore, these numbers represent best case performance; if there were contention, the `send_queue` execution times would increase.

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

Table 14. RGD_2 Send Queue (Per Task) Execution Time as a Function of Number of Packets

4.4.5.3. Update Queue

RGD_2 calls the `update_queue` procedure once for each task that suspended itself during the previous minor frame. This procedure updates pointers used in each PE's receive queue, located in the dual-port RAM. The execution time of each update_queue procedure call varies as a function of the number of receive_queue pointers which need to be updated and is equal to the number of packets enqueued during the previous frame. This is shown in Table 15. The total amount of time spent by RGD_2 updating queues is a function of the number of tasks that suspended themselves during the previous minor frame and the number of packets enqueued by each task.

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

Table 15. RGD_2 Update Queue (Per Task) Execution Time as a Function of Number of Packets

The data for send_queue and update_queue are linear, as shown by the graphical representation of the performance data, which is given in Figure 23.

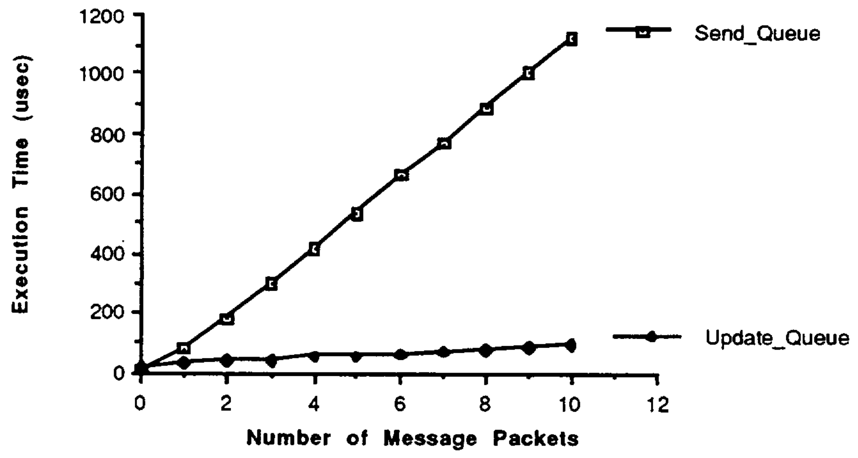

Figure 23. Graphical Representation of Send Queue and Update Queue Execution Time as a Function of Number of Packets

4.4.5.4. Schedule Rate Group Tasks

RGD_2 schedules rate group tasks to run in the time remaining in the current frame. It does this by calling a scheduling procedure for each rate group that reached its RG boundary during the previous minor frame. Therefore, this scheduling procedure is called a maximum of four times by RGD_2 (in minor frame 0). It is always called at least once during a minor frame. The execution time of the scheduler is a function of the number of tasks that need to be scheduled for a particular rate group. Table 16 summarizes the scheduler performance data. To collect more data points for the time needed to schedule RG tasks, the system configuration described at the beginning of this section was altered by adding more application tasks.

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

Table 16. RGD_2 Schedule Rate Group Tasks Execution Time as a Function of Number of Tasks Per Rate Group

4.4.5.5. Increment Frame Number, Set Up IO Interval for Next Frame

At the end of each RGD_2 execution cycle, the minor frame number is incremented and the IO interval is set up for the next minor frame. These activities take place just one time per RGD_2 execution. As seen in Table 17, the execution time to perform these duties is constant and is negligible compared to the total RGD_2 execution time.

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

Table 17. RGD_2 Increment Frame Number and Set IO Interval Execution Time

4.4.5.6. RGD_2 Summary

The overall RGD_2 execution time has five constituent parts. Two of these are constant, and they account for 49 µsec of RGD_2 overhead. Of the other three constituents, two (`send_queue` and `update_queue`) have execution times which are a function of the number of enqueued message packets. The final constituent of RGD_2 overhead is the time needed to schedule rate group tasks; this is a function of the number of tasks to schedule.

4.4.6. Fault Detection, Identification, and Recovery (FDIR) Overhead

The FDIR overhead for all Virtual Groups (VGs) within AFTA is the time to execute the Local FDIR task, except for the System VG, which executes the System FDIR task in addition to Local FDIR. Performance data for the System FDIR task are not presented because the task has not yet been fully implemented. Data for the execution times of the Local FDIR task are given in Table 18. Local FDIR simply enqueues a one-packet message which is delivered to the System FDIR task. Its execution time is constant, even with faults present in the system.

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

Table 18. Local FDIR Execution Time

4.4.7. IO Source Congruency Manager (IOSC) Overhead

The IO Source Congruency Manager ensures that all members of a redundant VG receive a copy of any input read by another member. The system configuration used to collect performance data used a simplex VG for IO, so the IOSC execution time reported in Table 19 should be regarded as a "best case" execution time.

avg time (µsec)	**std dev** (µsec)	**max time** (µsec)	**min time** (µsec)	**# samples**
52	1	59	52	142

Table 19. Minimal IO Source Congruency Manager Execution Time

4.4.8. IO Processing Task (IOP) Overhead

The IO Processing task is responsible for ensuring that all members of a redundant VG end up with a single input value. This involves some data smoothing or averaging. The performance measurements summarized in Table 20 indicate a relatively large standard deviation. This might be because there are four instantiations of this task, one for each rate group. The IOP code is not fully implemented, and the implementation will be strongly dependent on the application.

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

Table 20. Minimal IO Processing Task Execution Time

4.4.9. Other Overheads

There are several system overheads that are not explicitly shown in the minor frame overview given in Figure 11. These include the `queue_message` overhead, the `retrieve_message` overhead, and the time needed to context switch between tasks. Performance data for these three overheads are given in the following sections.

4.4.9.1. Queue Message

The `queue_message` procedure call is used by a task when sending a message. This procedure decomposes the message into packets and then enqueues these packets in the PE's local memory space for later transfer to the NE. As indicated in Table 21, the amount of time needed to enqueue a message is a function of the length of the message.

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

Table 21. Queue Message Execution Time as a Function of Message Size

4.4.9.2. Retrieve Message

The `retrieve_message` procedure is used by tasks to reassemble delivered packets into complete messages. As with `queue_message`, the time to retrieve a message is dependent upon the size of the message. This is shown in Table 22. Notice that it takes longer to retrieve a message of a given length than to enqueue it. When packets are delivered by the NE, syndrome information indicating whether any redundant copies of the packet differed from the majority vote is attached to each packet. While retrieving a message, some of this syndrome information is processed, and that accounts for the increased execution time.

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

Table 22. Retrieve Message Execution Time as a Function of Message Size

Figure 24 depicts a graphical representation of the data contained in Table 21 (Queue Message) and Table 22 (Retrieve Message).

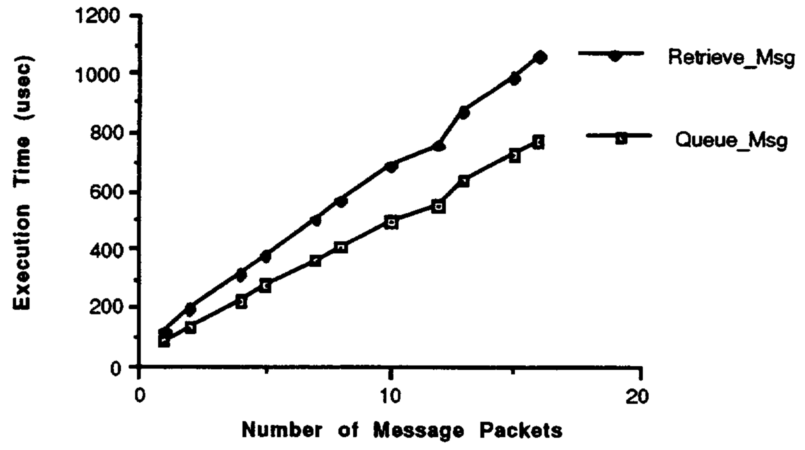

Figure 24. Graphical Representation of Queue Message and Retrieve Message Execution Time as a Function of Number of Packets

4.4.9.3. Context Switch Overhead

The amount of time needed to context switch between two tasks was measured, and the results are summarized in Table 23. These measurements were collected by creating a system configuration where two tasks in the same rate group were given consecutive priorities. This ensured that one task would execute immediately prior to the second one. Software probes were placed just before the iterative completion point of the first tasks and just after the iterative completion point of the second task. The context switch time was determined by subtracting the two timestamp values,.

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

Table 23. Context Switch Execution Time

4.4.10. Performance Data Summary

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

Communication Overheads

IH (scoop message)

RGD_2 (send_queue and update_queue)

Queue Message (called by application task)

Retrieve Message (called by application task)

Scheduling Overheads

RGD_1

RGD_2 (excluding send_queue and update_queue)

Context Switching

Input/Output

IOD

IOSC

IOP

Fault Detection

FDIR

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

The overheads associated with scheduling and fault detection are rather low compared with those associated with communication. Scheduling activities take, on average, 553 μsec per minor frame, which is 25.1% of the total OS overhead. The local FDIR task takes just 84 μsec per minor frame, or 3.8% of the total. Table 24 summarizes the percentage of overhead (excluding IO) due to communication, scheduling and fault detection. for an average minor frame. Note that the data in Table 24 represent values averaged over eight minor frames; the overhead can vary widely from minor frame to minor frame.

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

Table 24. OS Overhead Due to Communication, Scheduling and Fault Detection (Average Values for a Minor Frame)

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

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

4.5. Detailed OS Overhead Model

One important use of the performance data presented in the previous section is its incorporation into a model which can estimate the operating system (OS) overhead under various configurations and workloads. Using the empirical performance data summarized above, this section presents a detailed description of the OS overhead model, illustrates the use of the model with a given system configuration and workload, and compares predicted overheads to measured overheads.

4.5.1. OS Overhead Model with Empirical Data

This section gives a detailed description of the AFTA operating system overhead model, based on the empirical performance data presented above. The overhead model will be described according to the occurrence of each OS task in the minor frame (Figure 11).

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

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

A detailed description of each of these eight overheads follows.

4.5.1.1. Interrupt Handler (IH_1) Overhead

The overhead associated with the first interrupt handler (IH_1) is given by the following general equation:

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

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

The time to scoop message packets is a function of the number of packets that arrived in the processor's receive queue since the last time a scoop was executed. The data in Table 7 indicate that the relationship between the scoop time and the number of packets is linear. As a result, the overhead associated with the interrupt handler can be given as below:

$$IH_1 = 110 * \textit{number\_of\_packets} + 103 \ (\mu sec)$$

4.5.1.2. Rate Group Dispatcher - Part One (RGD_1) Overhead

The amount of time needed to execute the first part of the rate group dispatcher (RGD_1) can be summarized with the following general equation:

$$RGD_1 = (\textit{time to update congruent time}) + (\textit{time to check for } RGD_2 \textit{ overrun}) + (\textit{time to check for task overruns}) + (\textit{time to set up next RG interval}) + (\textit{time to schedule IOD})$$

With the exception of checking for task overruns, all the components of the rate group dispatcher (part one) are constant. Table 9 and Table 11 quantify this total constant overhead as 69 μsec. The time needed to check for task overruns varies with the number of tasks that completed their iterative cycle during the previous minor frame. Table 10 shows that this overhead is approximately 10 μsec per task. Therefore, the total overhead associated with RGD_1 can be described by the following:

$$RGD_1 = 10 * number\_of\_suspended\_tasks + 69\ (\mu sec)$$

4.5.1.3. IO Dispatcher (IOD) Overhead

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

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

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

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

4.5.1.4. Interrupt Handler (IH_2) Overhead

The overhead equation associated with the second interrupt handler (IH_2) is the same as that given for IH_1 and is repeated below:

$$IH_2 = 110 * number\_of\_packets + 103\ \ (\mu sec)$$

Even though both instances of the interrupt handler are modeled by the same equation, in general the overheads associated with IH_1 and IH_2 will be different. This is because the time to scoop messages will vary with the number of packets present in the receive queue for the processor. Typically, the time interval between the occurrence of IH_1 and IH_2 is less than the time duration from IH_2 to the next occurrence of IH_1. This implies that more packets have had an opportunity to arrive in the receive queue during the interval from IH_2 to IH_1, and therefore the time to scoop messages should generally be longer for IH_1 than IH_2.

4.5.1.5. Rate Group Dispatcher - Part Two (RGD_2) Overhead

The execution time for the second part of the rate group dispatcher (RGD_2) can be generally described as follows:

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

All but three of the RGD_2 constituents listed above have constant execution times. The time to update the congruent time value, check for RGD_1 and IOD overrun, increment frame count, and set up IO interval is constant and equals 49 μsec. The three variable constituents of RGD_2 are the time to send queued messages, the time to update queues, and the time to schedule RG tasks.

The time to send queued messages is a function of the number of tasks that suspended themselves during the previous minor frame and the number of message packets that each task had enqueued since the last time its queue was sent. For each task, the time to send the queued packets (Table 14) is given by:

$$\textit{Send\_Queue (per task)} = 115 * \textit{number\_of\_packets} - 31\ (\mu sec)$$

The time to update a task's queue is a function of the number of packets received and the number of packets read since the last time the queue was updated. Table 15 yields the following equation:

$$\textit{Update\_Queue (per task)} = 8 * \textit{number\_of\_packets} + 19\ (\mu sec)$$

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

$$\textit{Send\_and\_Update\_Queue (per task)} = 123 * \textit{number\_of\_packets} - 12\ (\mu sec)$$

The time to schedule the rate group (RG) tasks is a function of the number of RG tasks that are to be scheduled this minor frame. The data in Table 16 results in the following equation:

$$\textit{Schedule\_Tasks (per rate group)} = 26 * \textit{number\_of\_rg\_tasks} + 15\ (\mu sec)$$

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

RGD_2 = (time to send and update queues) + (time to schedule tasks) + (a constant)

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

$$RGD_2 = \sum_{i=1}^{num\_tsk} [(123 * num\_pkt_i) - 12] + \sum_{j=1}^{num\_rg\_tsk} [(26 * num\_rg\_tsk_j) + 15] + 49\ (\mu sec)$$

where,

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

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

4.5.1.6. Fault Detection Identification and Recovery (FDIR) Overhead

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

FDIR = (time to enqueue message to System FDIR task)

The Local FDIR task has a constant execution time, as shown in Table 18. Therefore, the overhead for FDIR can be expressed as:

$$FDIR = 84\ (\mu sec)$$

The overhead for different types of fault recovery strategies (e.g., degrade the system or virtual group, bring up a hot spare) was not measured and is not included in the overhead summary. These times must be eventually included into the analysis to allow the user to estimate the performance overhead for handling faults.

4.5.1.7. IO Source Congruency Manager (IOSC) Overhead

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

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

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

$$IOSC = 52 \ (\mu sec)$$

4.5.1.8. IO Processing Task (IOP) Overhead

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

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

The IOP task is not fully implemented, and the implementation will be strongly dependent on the application. Therefore, the data for IOP execution time given in Table 20 represent minimum execution times for IOP. Using these data, the minimal IOP overhead is:

$$IOP = 15 \ (\mu sec)$$

4.5.1.9. Total OS Overhead

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

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

where,

$$IH_1 = 110 * \text{number\_of\_packets} + 103$$

$$RGD_1 = 10 * \text{number\_of\_suspended\_tasks} + 69$$

$$IH_2 = 110 * \text{number\_of\_packets} + 103$$

$$RGD_2 = \sum_{i=1}^{\text{num\_tsk}} [(123 * \text{num\_pkt}_i) - 12] + \sum_{j=1}^{\text{num\_rg\_tsk}} [(26 * \text{num\_rg\_tsk}_j) + 15] + 49$$

FDIR = 84

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

$$OH = (110 * num\_pkt\_scooped) + (10 * num\_tsk) +$$

$$\sum_{i=1}^{num\_tsk} [(123 * num\_pkt_i) - 12] + \sum_{j=1}^{num\_rg\_tsk} [(26 * num\_rg\_tsk_j) + 15] + 305 \ (\mu sec)$$

where,

num_pkt_scooped	is the total number of packets scooped during the minor frame.
num_tsk	is the number of tasks with messages to send that completed their iterative cycle during the previous minor frame.
num_pkt	is the number of packets a task has enqueued since its last `send_queue` call.
num_rg	is the number of rate groups that begin a new frame boundary in the current minor frame.
num_rg_tsk	is the number of tasks in a given rate group.

4.5.2. Example of Overhead Model Use

To illustrate the use of the detailed OS overhead model presented above, an example system configuration is created , the system parameters are used as input to the overhead model in order to predict the OS overheads, and the predicted overheads are compared with empirically measured overheads. This section also illustrates several other ways to use the OS overhead model.

4.5.2.1. Description of Example System Configuration

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

RG4 tasks (six)
- fdir (local)
- system_fdir
- io_source_congruency_mgr
- io_processing_task_rg4
- io_application_task (user task)
- application_task _1 (user task)

RG3 tasks (two)
- io_processing_task_rg3
- application_task _2 (user task)

RG2 tasks (one)
- io_processing_task_rg2

RG1 tasks (two)
- io_processing_task_rg1
- application_task _3 (user task)

4.5.2.2. Predicted Overheads

The OS overheads vary as function of several parameters. These parameters include the total number of message packets scooped, the number of tasks that completed their iterative cycle during the previous minor frame, the number of packets sent by each task during the previous frame, the number of rate groups that reached a frame boundary during the previous minor frame, and the number of schedulable tasks for each of the rate groups which are at a frame boundary. Based on the system configuration given above, the values for each of these parameters during each minor frame are given in Table 25.

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

Table 25. System Parameters for Each Minor Frame

Using the parameter values given in Table 25 and the OS overhead equations, the OS overhead for each minor frame can be predicted. These predictions are presented below:

Frame 0

$IH_1 = (110 * 12) + 103 =$ 1423 μsec

$RGD_1 = (10 * 11) + 69 =$ 179 μsec

$$RGD2 = [(123 * 3) - 12] + [(123 * 6) - 12] + [(123 * 2) - 12] + [(123 * 1) - 12] + [(26 * 6) + 15] + [(26 * 2) + 15] + [(26 * 1) + 15] + [(26 * 2) + 15] + 49 = 1823 \mu sec$$

FDIR = 84 μsec

TOTAL = 3509 μsec (35.1% of minor frame)

Frames 1, 3, 5, and 7

$IH_1 = (110 * 4) + 103 =$ 543 μsec

$RGD_1 = (10 * 6) + 69 =$ 129 μsec

RGD2 = [(123 * 3) - 12] + [(123 * 1) - 12] + [(26 * 6) + 15] + 49 = 688 μsec

FDIR = 84 μsec

TOTAL = 1444 μsec (14.4% of each minor frame)

<u>Frames 2 and 6</u>

IH_1 = (110 * 10) + 103 = 1203 μsec

RGD_1 = (10 * 8) + 69 = 149 μsec

RGD2 = [(123 * 3) - 12] + [(123 * 6) - 12] + [(123 * 1) - 12] +
[(26 * 6) + 15] + [(26 * 2) + 15] + 49 = 1481 μsec

FDIR = 84 μsec

TOTAL = 2917 μsec (29.2% of each minor frame)

<u>Frame 4</u>

IH_1 = (110 * 10) + 103 = 1203 μsec

RGD_1 = (10 * 9) + 69 = 159 μsec

RGD2 = [(123 * 3) - 12] + [(123 * 6) - 12] + [(123 * 1) - 12] +
[(26 * 6) + 15] + [(26 * 2) + 15] + [(26 * 1) + 15] + 49 = 1522 μsec

FDIR = 84 μsec

TOTAL = 2995 μsec (30.0% of minor frame)

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

4.5.2.3. Comparison of Predicted and Actual Overheads

To determine the accuracy of the OS overhead model, empirical performance data were collected using the system configuration described above. A comparison of the overheads predicted by the model and the observed overheads is given in Table 26. Note that the overheads are the average values for a minor frame; individual overheads varied from minor frame to minor frame.

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

Table 26. Comparison of Predicted and Measured Overheads

(Average Values for a Minor Frame)

Table 26 shows that the overhead model is accurate for the IH_1, RGD_1, and FDIR overheads. However, the predicted RGD_2 overhead is 17.0% less than the observed overhead. The RGD_2 error caused the total predicted overhead to be 8% less than the total measured overhead (excluding IO).

There are several causes for the inaccuracy of the RGD_2 model. The primary cause is the model does not account for the time consumed by `send_queue` and `update_queue` when a task has no message packets to send. The overhead of making the `send_queue` call for tasks with no message packets is 5 μsec per task (Table 14). The corresponding overhead for `update_queue` is 16 μsec per task (Table 15). Therefore, 21 μsec is spent for each task that doesn't have any message packets to send. If a 0 is inserted into Equation 1 for the number of packets, the equation results in a -12 μsec overhead to send_and_update for each task, instead of the correct 21 μsec value. Equation 1 is a least squares line approximation to the data contained in Table 14 and Table 15. The approximation is very accurate except for the case when the number of packets equals zero. The model currently only considers send_and_update overheads for tasks that have message packets to send. To be more accurate, it should account for the overhead for tasks that have no packets.

To see the effect of this on the RGD_2 overhead, consider minor frame 1. The model predicts an RGD_2 overhead of 640 μsec, versus the observed overhead of 804 μsec (25.6% error). If the send_and_update overhead for the four tasks in that minor frame which had no messages is included, the predicted RGD_2 overhead becomes 724 μsec, and the RGD_2 error is reduced to 10.0%. This reduction in the RGD_2 error can be achieved by using the following modified send_and_update queue equation:

Send_and_Update_Queue (per task)

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

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

Another cause for the RGD_2 overhead error is the inaccuracy of the least square line used to predict the time to schedule RG tasks. The time predicted by this equation can be as much as 22% in error. For better accuracy, the time to schedule RG tasks should be determined by a second- or third-order polynomial, instead of a linear approximation. Figure 25 is a graphical comparison of the measured overhead associated with scheduling tasks with the least square line approximation of that data.

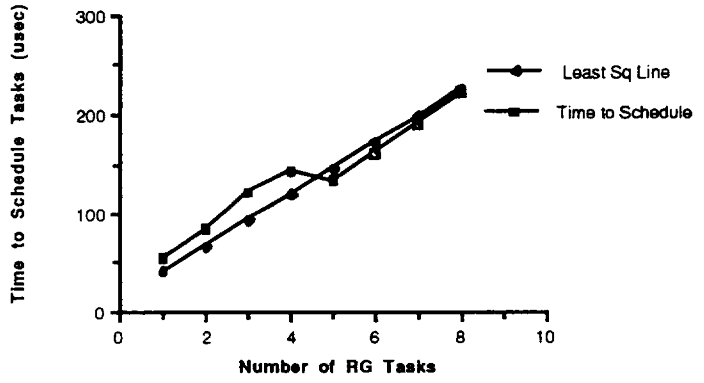

Figure 25. Comparison of Time to Schedule Tasks (Measured) with Least Square Line Approximation

The RGD_2 overhead is much more susceptible to inaccuracies in the model than the other overheads because the RGD_2 code contains several nested loops that can cause small errors to quickly multiply into significant ones. For example, `send_queue` and `update_queue` are called once each for every task that completed its iterative cycle during the previous minor frame. For the configuration given in this section, `send_queue` and `update_queue` are called 11 times each during minor frame 0. Any error in the predicted overheads for `send_queue` and `update_queue` will be multiplied by 11; thus, a small error may quickly become a significant one.

4.5.2.4. Other Uses of OS Overhead Model

In addition to its use in predicting overhead for a given system configuration, the OS overhead model can also be used to predict bounds on OS performance. For example, the model can be used to determine the minimum amount of OS overhead. A minimal configuration would consist of the following system tasks: Local FDIR (RG4), IOSC (RG4), IOP

(RG4), IOP (RG3), IOP (RG2), IOP (RG1). One RG1 application task which did not send any messages would also be present. The OS overhead model predicts the average total OS overhead per minor frame (excluding IO) for this minimum configuration to be 698 µsec (7% of minor frame).

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

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

5. POSIX Study

5.1. Objective and Approach

The overall objective of this task was to extend AFTA's open system characteristics to include its operating system and software. To achieve this objective, the AFTA Network Element was interfaced to a standard operating system, which was then hosted on a quadruply redundant AFTA. A POSIX-compliant operating system was selected for this demonstration. The utility of the resulting system was demonstrated by rehosting and executing an Army flight-critical application (Dynapath Terrain-Following / Terrain Avoidance) on the AFTA.

5.2. Overview of Progress

Several POSIX-compliant kernels were evaluated via vendor presentations and literature surveys. These kernels included LynxOS, Quantum QNX, RTMX Uniflex, and HP-RT. Because it is currently a market leader and compatible with the 68030 processors currently in use in the AFTA, LynxOS was selected for detailed evaluation and demonstration. LynxOS was purchased and installed on a nonredundant simplex MVME147 68030-based workstation. (Because of the extremely rapid pace at which new processors and kernels are being introduced, this decision should be re-evaluated at appropriate intervals.)

The AFTA NE was installed into the simplex LynxOS environment and tested via the self-test code developed under the earlier AFTA detailed design phase. LynxOS / UNIX-compatible device drivers were written to allow application programs to access the NE, and a simple Application Programmer Interface was implemented to allow application programs to perform interchannel exchanges and synchronization. "Dynapath," a terrain-following / terrain-avoidance helicopter trajectory generation application developed by NASA Ames, was acquired from the Army and demonstrated in real-time on this nonredundant workstation environment, with interchannel exchanges being performed by the AFTA NE. The NE was operated in "fiber optic loopback mode," in which the single NE's optical output was connected to its inputs to simulate being connected to four other FCRs.

Subsequently, LynxOS was installed in the quadruply redundant target environment of the AFTA. This environment consists of four FCRs, each containing one NE and one or more 68030 PEs. The NE interface device drivers and Dynapath were ported to this environment and demonstrated. Limited fault injections (e.g., channel resets) were performed to demonstrate fault tolerant behavior.

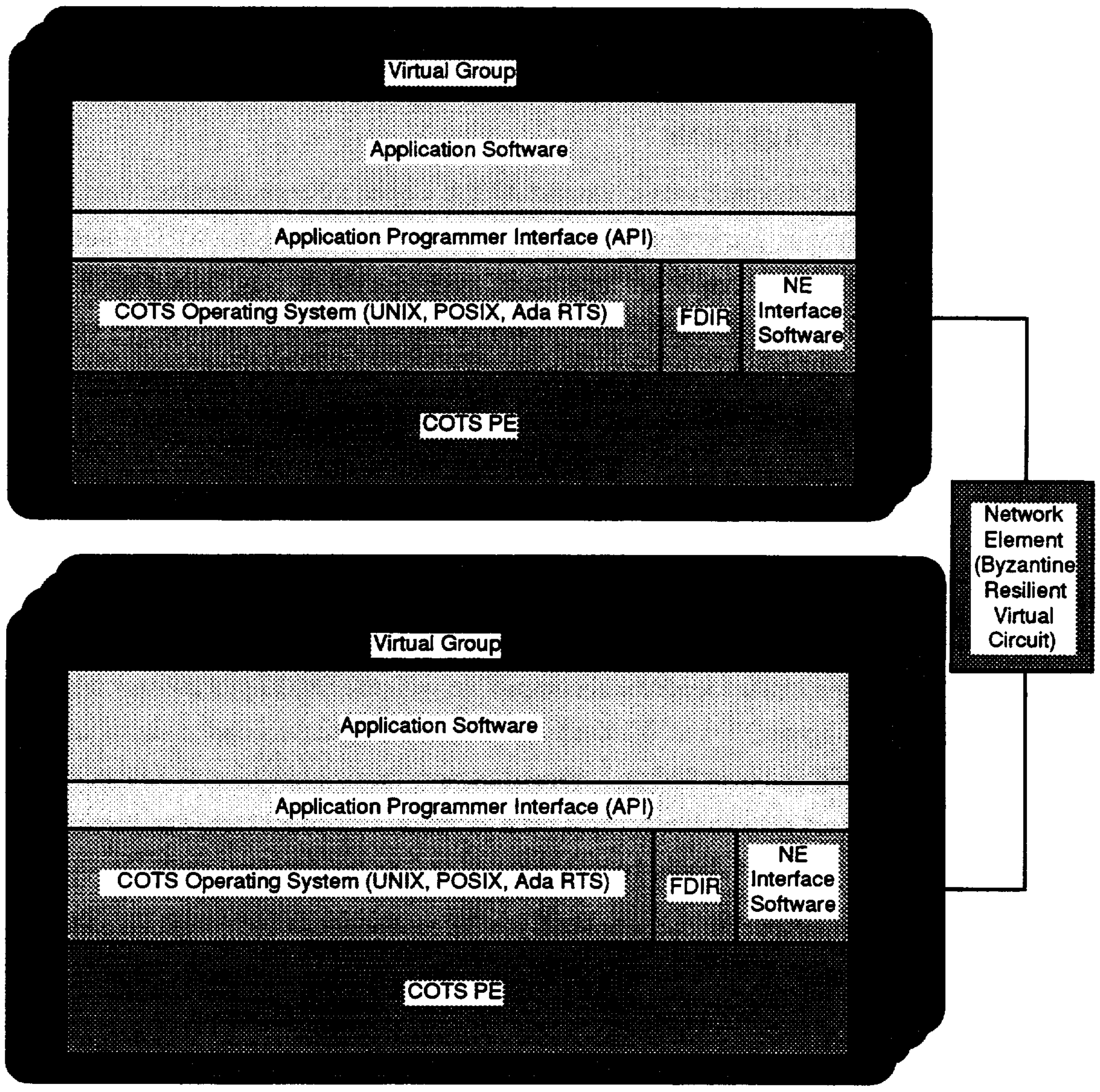
Virtual Group
Application Software
Application Programmer Interface (API)
COTS Operating System (UNIX, POSIX, Ada RTS)
FDIR
NE Interface Software
COTS PE
Network Element (Byzantine Resilient Virtual Circuit)
Virtual Group
Application Software
Application Programmer Interface (API)
COTS Operating System (UNIX, POSIX, Ada RTS)
FDIR
NE Interface Software
COTS PE

Figure 26. Generic Layered View of AFTA

5.3. Generic AFTA Virtual Architecture

A generic layered view of the AFTA is shown in Figure 26. Multiple COTS Processing Elements (PEs) are formed into synchronous Virtual Groups having redundancy of one, three, or four. Figure 26 shows a quadruplex and triplex VG. Each VG executes different application software and accesses the fault tolerant-related services, if desired, via an Application Programmer Interface (API). The API and application tasks reside within the context of a COTS operating system. Fault Detection, Identification, and Recovery task also resides in the OS as a separate task, and detects and identifies faulty components and performs recovery actions appropriate to the application and mission phase. The API interacts with the application software on the one hand and the NE interface software on the other hand, to provide the Byzantine Resilient Virtual Circuit Abstraction for the application software.

5.4. POSIX Study Virtual Architecture

The generic abstract architecture above was instantiated as shown in Figure 27 for the POSIX study. The Dynapath application program invokes the "exch()" primitive to perform source congruency and voting on input and output data, respectively. The exch() primitive also synchronizes the multiple redundant copies of Dynapath. The exch() primitive accesses the NE via a UNIX device driver. FDIR was not implemented for this demonstration.

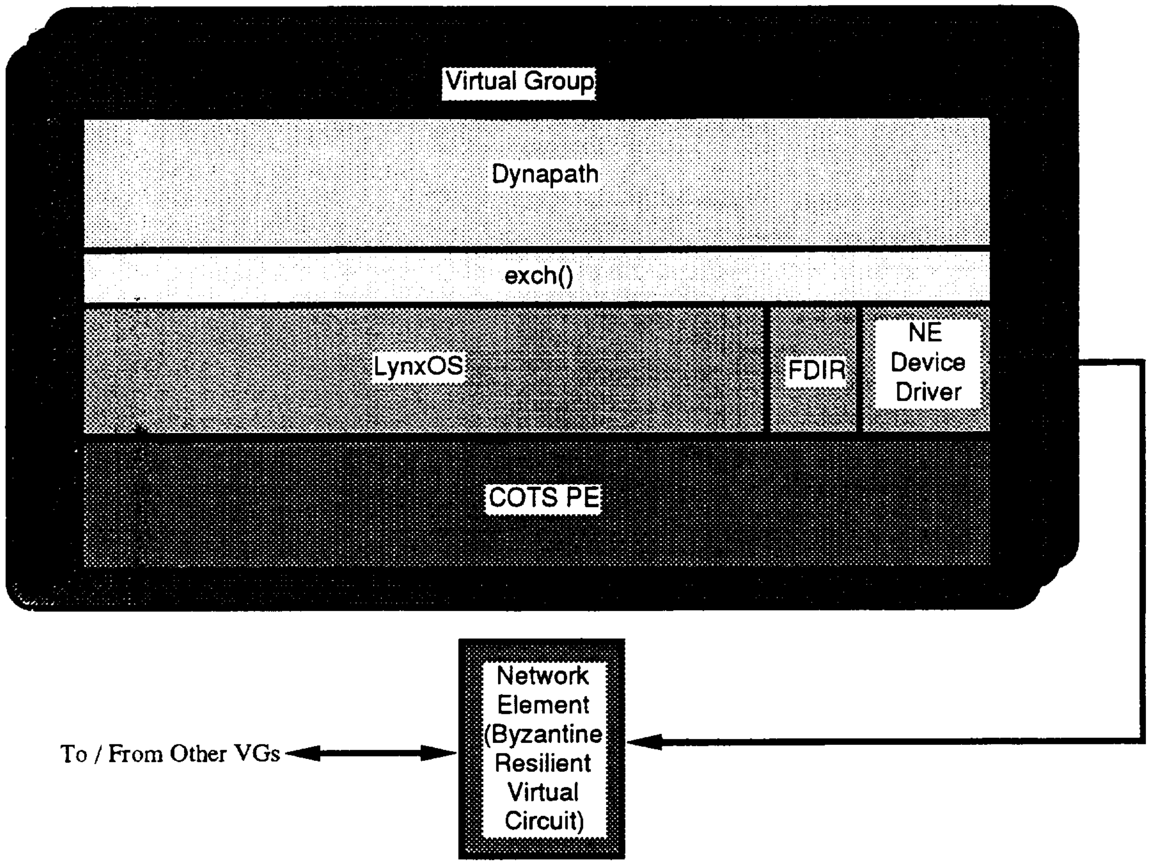

Figure 27. Layered View of AFTA for POSIX Study

5.5. POSIX Study Physical Architecture

In the physical architecture, the host workstation executes the LynxOS "self-hosted" operating system and development environment. This environment contains the editors, compilers, linkers, file systems, and download facilities required for code development.

The Network Element was installed into this environment for preliminary development and testing (Figure 28), but was subsequently removed to the target environment.

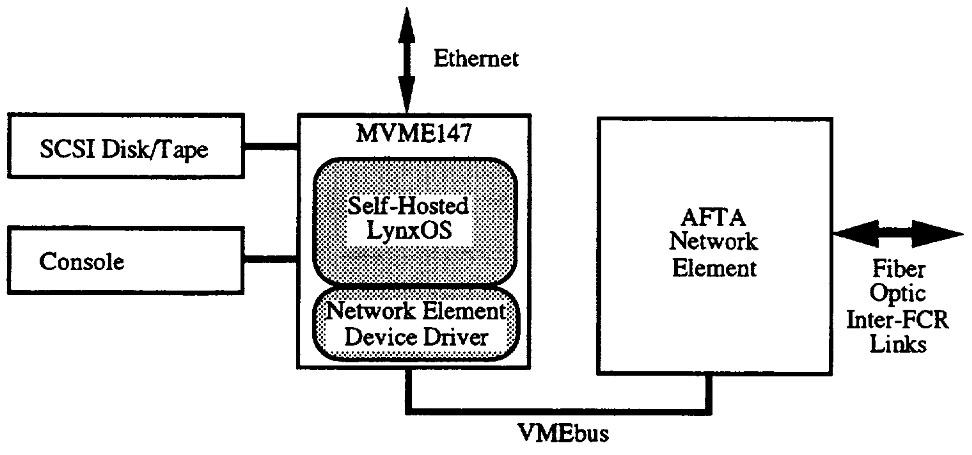

Figure 28. Simplex Self-Hosted Development Environment

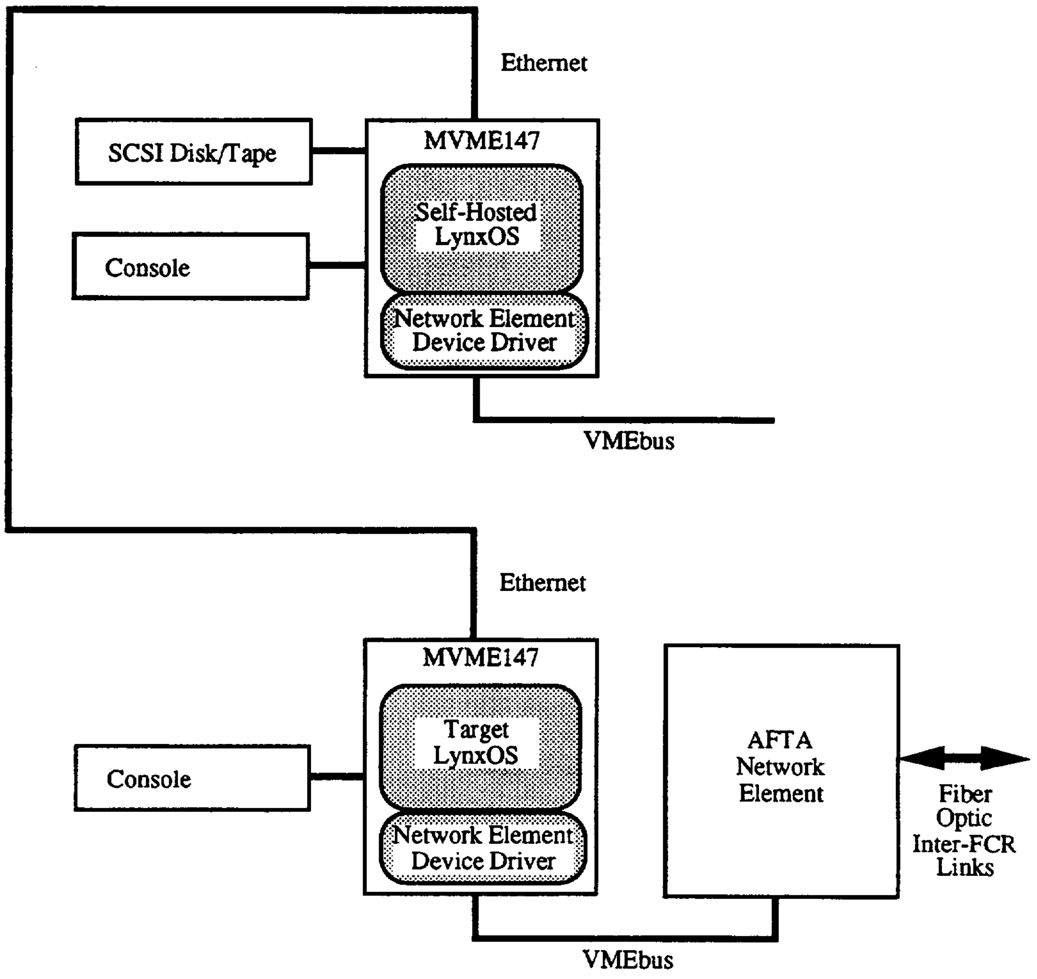

Figure 29. Simplex Target Environment and Development Environment

In the next step of the development, a target LynxOS was obtained and ported to a target environment which did not contain a disk or tape. The Network Element was moved to this environment to comprise a complete single FCR of an AFTA (Figure 29). All NE interface code was then ported to this environment. All communication (downloading, file services, etc.) among the development environment, target environment, and other computers in the laboratory occurs over Ethernet.

At this stage of the development it was noticed that there was very little difference in operation between the self-hosted and target LynxOS environments. Application and system code which was developed on the self-hosted workstation environment in general executes without modification on the target environment.

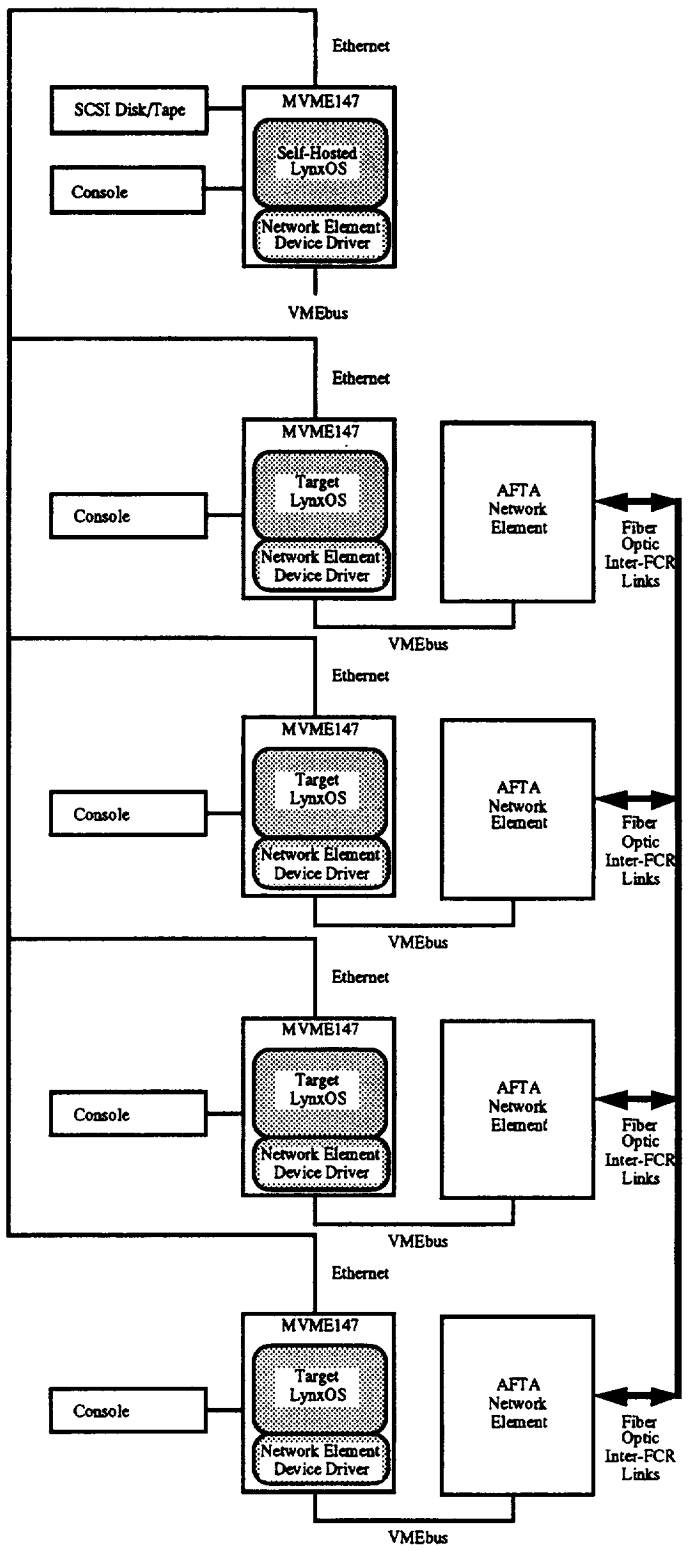

Figure 30. Redundant Target Environment and Development Environment

Subsequently, the LynxOS target operating system, the NE interface code, and the application code were ported to all four processors in the quadruply redundant AFTA (Figure 30).

5.6. POSIX Study Scheduling

Under the POSIX study the rate group task scheduling model developed under previous AFTA tasks was extended to remove certain limitations. The task taxonomy used in the context of this study partitions tasks according to their trigger mechanism and execution time. Tasks may be either "time triggered" or "event triggered." Time triggered tasks may be either periodically triggered (e.g., every hour) or sporadically triggered (e.g., at five o'-clock today). Event triggered tasks may be triggered by externally-occurring events (e.g., the operator pushes a button), or internally triggered (e.g., a buffer condition reaches a given state, a prior iteration of the task has completed). Moreover, regardless of how they are triggered, tasks may have constant (or bounded) or variable (or unbounded) execution times. The matrix of possibilities is shown in Figure 31.

			Execution Time	
			Constant (bounded)	Variable (unbounded)
Trigger Mechanism	Time	Periodic	H, S	S
		Sporadic	H, S	S
	Event	External	H, S	S
		Internal	H, S	S

H = Hard Real Time
S = Soft or Non-Real Time

Figure 31. Task Taxonomy

5.6.1. Periodic Hard Real-Time Tasks

Time-triggered periodic hard real-time tasks are specified by the ordered pair <period, offset>. Task period can be an arbitrary number of minor frames. Obviously, the execution time of the task must be less than its period. A task can be scheduled to start at an arbitrary start frame offset, as measured from a given baseline frame. Periodic hard real-time tasks are scheduled according to rate monotonic theory. This model should be compared to AFTA Ada RTS rate group scheduling model described in Section 4 of this document

Figure 32 shows three such tasks. Task 1 starts on frame 0 and has a period of 1 frame, Task 2 starts on frame 1 with period 2, and Task 3 starts on frame 0 with period 5.

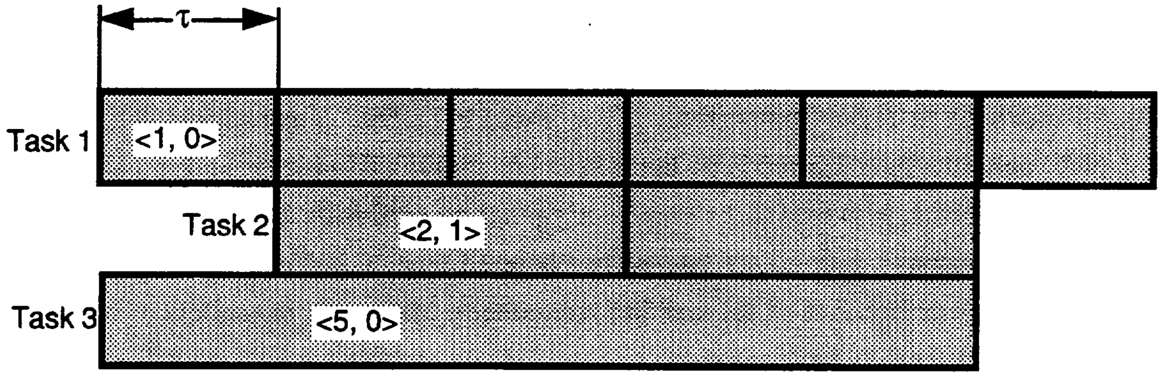

Figure 32. Scheduling of Hard Real-Time Periodic Tasks

5.6.2. Event-Triggered Hard Real-Time Tasks

An event-triggered hard real-time task is specified by the same parameters as periodic tasks, except that the start frame offset now refers to how many frames after an event's occurrence the task must be started. Note that an explicit decision is required at system programming time as to which task(s) event-triggered task(s) may preempt. Once enabled, scheduling of an event-triggered hard real-time task is mechanized via its priority within rate monotonic priority class corresponding to the period of the task.

Figure 33 shows event-triggered Task 4 which has a maximum execution time of 1 period, and an offset of 0 after the frame in which the event occurs.

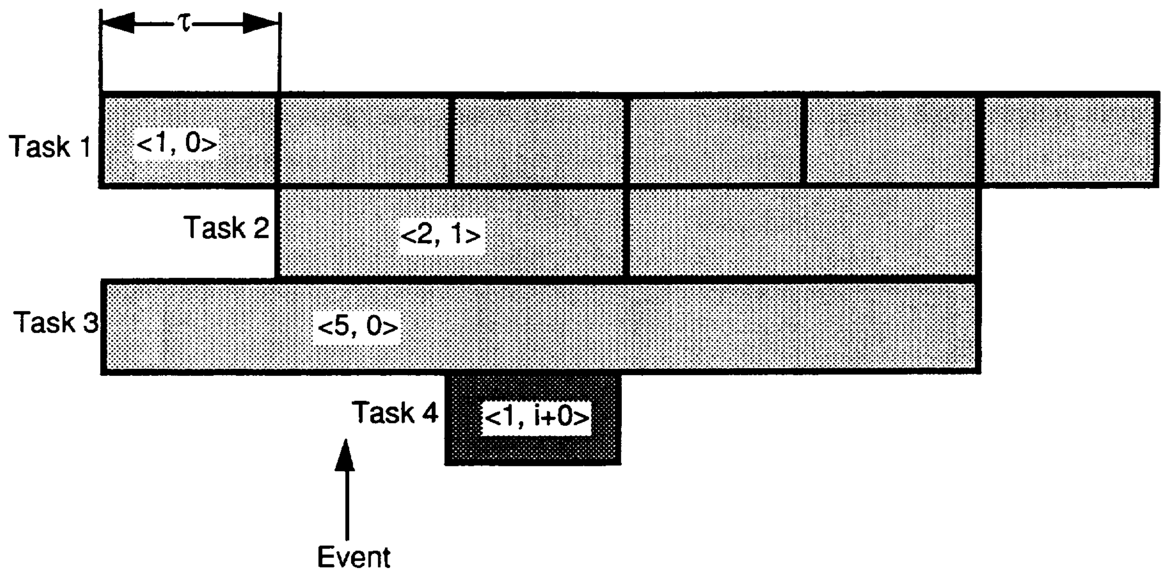

Figure 33. Scheduling of Event-Triggered Hard Real-Time Tasks

5.6.3. Time- or Event-Triggered Soft Real-Time Tasks

Time- or event-triggered soft real-time tasks are specified by the maximum skew which can build up between nonfaulty redundant copies of the task between "synchronization points." Soft real-time tasks may be arbitrarily scheduled by the underlying operating system according to a totally arbitrary (e.g., round-robin or self-suspension) policy, so long as they may be preempted by hard real-time tasks. A synchronization point may be a request to perform source congruency on data, vote data, wait for a specified time interval, wait until a given time, or other any other "synchronizing act" as needed by the application.

When the redundant copies of a soft real-time task arrive at a synchronization point, they invoke the AFTA NE device driver which registers their request to perform the synchronization act. This invocation blocks the caller until the synchronization act request has been approved and executed by the AFTA NE device driver. At timer interrupts, the AFTA NE device driver interrupt service routine (ISR) exchanges the synchronization act request patterns of all soft real-time tasks and determines which may be approved and executed. A redundant task's synchronization act is approved and executed by the ISR if all copies of the task have requested the exchange, or a majority of tasks have requested the act and the maximum task skew has expired. After the synchronization act has been executed by the device driver, the caller is unblocked and may continue execution. Note that when the caller is unblocked, the redundant copies of the caller are synchronized. Each task may have a different skew, which it may change at any time using calls to the AFTA NE device driver.

Figure 34 shows soft real-time Task 5, which occasionally requests synchronizing acts which are approved on scheduling frame boundaries by the AFTA NE device driver. Note the variation in task period and skew which this approach accommodates.

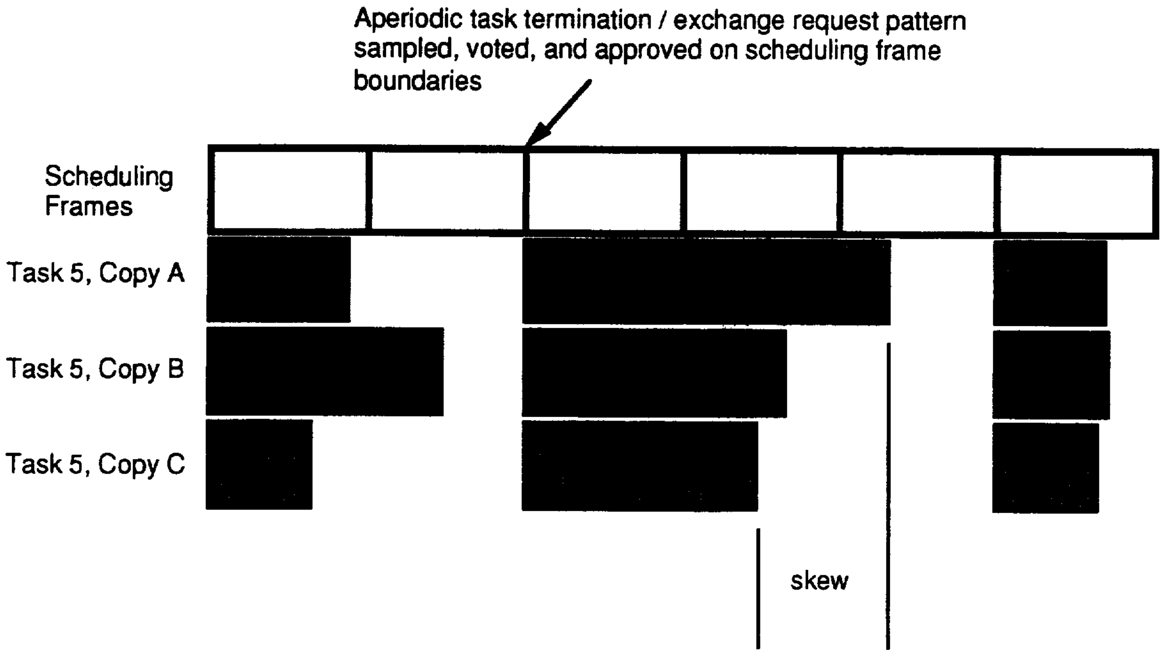
Aperiodic task termination / exchange request pattern sampled, voted, and approved on scheduling frame boundaries
Scheduling Frames
Task 5, Copy A
Task 5, Copy B
Task 5, Copy C
skew

Figure 34. Scheduling of Aperiodic Tasks

5.7. Network Element Device Driver

A UNIX-compatible device driver interface was implemented under LynxOS to allow applications to interface with the Network Element. This device driver supports the UNIX install(), open(), write(), read(), ioctl(), close(), and uninstall() calls.

5.7.1. Device Driver Operations

The NE is opened and closed using standard open() and close() calls. Once the device has been opened using the open() call, interactions with the NE are accomplished using the ioctl() call. NE-related functions which can be performed using this call are:

NE_ISYNC. This operation performs initial synchronization of the Network Elements. It is only necessary to perform this operation once after bootstrapping the AFTA. Repeated invocations of the NE_ISYNC function have no effect.

TIMER_PERIOD. This operation sets the period (in microseconds) of the timer-driven interrupt service routine (ISR) which exchanges the synchronization act request pattern of all tasks, performs unanimity and majority plus timeout calculation on these patterns, performs all enabled exchanges, and unblocks tasks which are awaiting completion of a synchronization act.

TIMER_SET. This operation enables the timer interrupt, which, every TIMER_PERIOD microseconds thereafter, causes the NE device driver ISR to be executed.

TIMER_RESET. This operation disables the timer interrupt.

NE_EXCH_TIMELIMIT. This operation allows the application programmer to set the amount of time that the NE_EXCHANGE operation will wait before declaring a timeout on a tardy exchange request and, consequently, enabling the exchange.

NE_EXCHANGE. This operation requests a one-round (vote) or two-round (source congruency) exchange of data. The exchange is performed by the NE device driver ISR only if all of the copies of the caller have requested the exchange, or a majority of the callers have requested the exchange and at least NE_EXCH_TIMELIMIT microseconds have expired. The caller is blocked until the exchange has been completed.

Additional NE-related functions, such as changing the Configuration Table and performing inline exchanges without waiting for the ISR, will be added to this driver as needed by upcoming applications.

5.7.2. Device Driver Installation

The Network Element memory map is defined in Volume 4 of the AFTA Conceptual Study. For the LynxOS integration, the NE was located in standard VMEbus address space at location 10000000 (hex). The device driver was installed using the following script prior to execution of Dynapath.

```
drinstall  -c  NE_driver
mknod  /dev/ne  c  8  0
devinstall  -c  -d  9  NEinfo
```

Figure 35. AFTA NE Device Driver Installation Script

5.8. Dynapath Demonstration Architecture

This section describes the architecture and operation of the Dynapath demonstration which was hosted on the quadruply redundant environment described above.

Dynapath is an algorithm for generating a low-altitude helicopter trajectory through rugged terrain. It uses digital map data, the current vehicle state (e.g., position, velocity), vehicle dynamical constraints (e.g., maximum rate-of-bank), a set of waypoints over which the vehicle must fly, desired trajectory constraints (e.g., setpoint altitude), and other information to construct a trajectory which meets all these constraints and requirements. The generated trajectory is then presented to the pilot on a head-up-display (HUD) in a simple-to-use "highway-in-the-sky" format, which the pilot may follow. The Dynapath functionality is likely to be safety-critical, especially in low-visibility conditions.

Figure 36 shows the major components of the demonstration. At the left of the figure, Dynapath resides on the quadruply redundant AFTA, along with LynxOS and the Network interface software described elsewhere in this report. Dynapath communicates with vehicle dynamical simulation software (Helsim) and the HUD symbology generation software running on a Silicon Graphics (SG) workstation using Ethernet-based TCP/IP. The out-of-window view of the terrain, the Dynapath-generated highway-in-the-sky symbology, and other HUD symbology are presented on a high-resolution graphics monitor connected to the SG.

Periodically, Dynapath transmits a request for vehicle state from the helicopter simulation. When it receives a state update from the simulation, Dynapath calculates a new commanded trajectory segment and transmits the new trajectory segment description to the symbology generation software. The "pilot" views the terrain and Dynapath symbology and provides cyclic and collective commands to the helicopter simulation via a mouse and joystick as she attempts to follow the commanded trajectory.

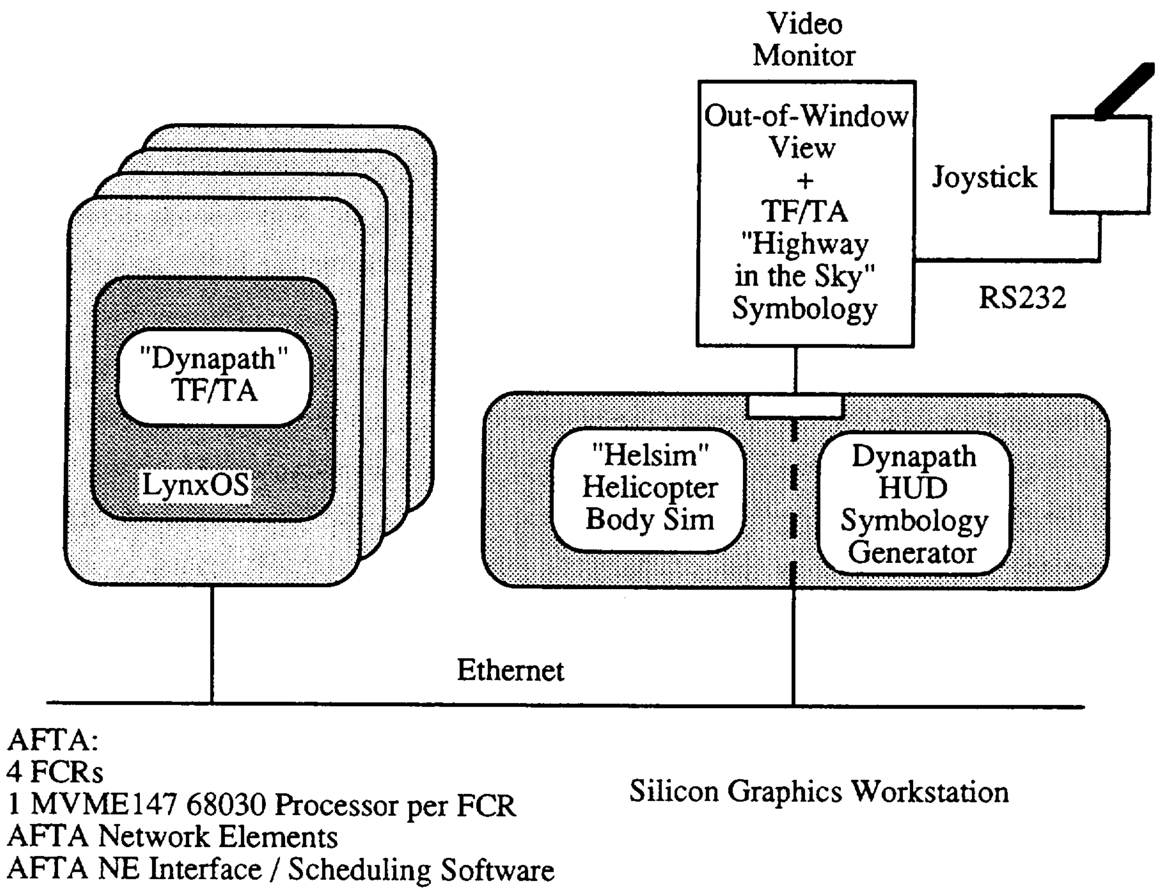
Video Monitor
Out-of-Window View + TF/TA "Highway in the Sky" Symbology
Joystick
RS232
"Dynapath" TF/TA
LynxOS
"Helsim" Helicopter Body Sim
Dynapath HUD Symbology Generator
Ethernet
AFTA:
4 FCRs
1 MVME147 68030 Processor per FCR
AFTA Network Elements
AFTA NE Interface / Scheduling Software
Silicon Graphics Workstation

Figure 36. Architecture of Dynapath Demonstration

5.8.1. Dynapath Application Programmer Interface to NE

Dynapath is scheduled on the AFTA as an aperiodic soft real-time application task. The details of the scheduling are of no consequence. While four copies of Dynapath run on the redundant AFTA, only one processor (say, processor A) is connected over Ethernet to the SG. Therefore Dynapath must invoke NE exchange primitives on the following occasions:

1. Processor A polls its Ethernet input buffers and provides all channels with consistent copies of the polling result.

2. Processor A reads its Ethernet input buffers and provides all channels with consistent copies of the input data.

3. All processors have completed computation of a trajectory segment and must vote the result before Processor A transmits it to the SG over Ethernet.

Dynapath uses the blocking "exch()" call to perform these operations. The exch() call is built on the AFTA NE device driver NE_EXCHANGE ioctl, which blocks Dynapath until all copies of Dynapath have requested the exchange, or a majority of the copies have requested the exchange and a user-defined timeout has expired. At this point, the exchange is said to be "enabled." The exchange can be either a request for a vote or a request to distribute single-source data, such as that obtained from the helicopter simulation, to all members of the Virtual Group. When the exchange is enabled, the AFTA NE device driver ISR performs the requested exchange, delivers the exchanged data into the buffer by the caller of the ioctl, and unblocks the caller.

The contents of the exch() call are shown in Figure 37. The call is invoked by the application programmer with four arguments. The pointer "raw" points to the source of the data to be exchanged, "voted" points to the destination of the data to be exchanged, "class" indicates the class of the exchange (i.e., single-source or voted), and "size" indicates the size (in bytes) of the data to be exchanged. The current implementation of the NE_EXCHANGE ioctl clobbers the data pointed to by raw.

The exch() call stores this data into a structure which is shared between the application program and the Network Element Device Driver, sets the exchange request flag (dyna_exch.sync_flag), and waits on the successful completion of the ioctl(), indicating that the exchange has been completed. At this point, control is returned to the application task, which can access the exchanged data.

```
void exch(char *raw, char *voted, int class, int size)
{
      dyna_exch.raw_data = raw;
      dyna_exch.voted_data = voted;
      dyna_exch.class = class;
      dyna_exch.size = size;
      dyna_exch.sync_flag = TRUE;
      /* block here until exchange complete */
      if( ioctl(NE_fd, NE_EXCHANGE, &dyna_exch) < 0)
            {
             perror("NE_EXCHANGE");
            exit(-2);
            }
      /*    exchange complete */
}
```

Figure 37. Listing of exch() Procedure

5.8.2. Dynapath Code for Interfacing with NE Device Driver

This section contains the file "dutils.c" which contains definitions of Dynapath's interface to the Network Element. The file also demonstrates how to open the NE device driver, vary the exchange timeout for the Dynapath task, initially synchronize the NEs (which also sets up the timer-based interrupt service routine which synchronously services the NE exchange requests)†, and close the NE. The exch() call code is repeated here, which in this case includes code which reduces the exchange timeout for the Dynapath task. The NE-specific calls are printed in boldface and enclosed in boxes.

Note that Dynapath uses two task skew timeouts. Initially the timeout is large since the multiple copies of Dynapath build up a huge skew as they contend for and read the waypoint file from the single copy of the waypoint file, which is NFS-mounted on the LynxOS server. However, after the file is read and iterative execution of the task begins, no further file accesses are needed so Dynapath can tighten up the skew. Also note that an application task can modulate its skew as it executes and enters and leaves skew-inducing phases such as file reads and writes. For example, if Dynapath were to enter a phase in which it reads or writes a large shared file or performs a lengthy Ethernet transmission, it could temporarily increase its maximum skew parameter.

† These functions would normally not be done by an application task. In this demonstration they were implemented in Dynapath for simplicity.

```
#include  "NE_driver.h"
#include <stdio.h>
#include <smem.h>
#include <sem.h>
#include <pthread.h>
#include <lock.h>
#include <file.h>
#include "playback.h"
/*                          NOTICE                                     */
/*                                                                     */
/*    This NASA computer program has been released soley for one of   */
/*    the purposes set forth in NASA Management Intruction 2210.2B    */
/*    and further dissemination of the program is prohibited.         */
/*                                                                     */
char dmain[] = "@(#)dutils.c  1.4   12/31/91";
void *attach_map();
extern void dynapath();
int  NE_fd;
int  waypointread,timeoutlowered;
main(int argc, char *argv[])
{
  int *ptr;
  int loadid;
  char set[5];
  int dyna_status;
   /*  presence  vector  and  generic  "integer  value"  */
   int  pvect,ival;
  int sbsock;
   waypointread  =  FALSE;
   timeoutlowered  =  FALSE;

   /*  This  code  opens  the  NE  device  driver  */
   if((NE_fd  =  open("/dev/NE",  O_RDWR))  <  0)
  {
      perror("open  NE");
      pthread_exit(-1);
  }
   printf("opened  NE  \n");

   /*  Set  NE  device  driver  timer  interrupt  period  */
   ival  =  40000;  /*  40ms  period  */
   if  (ioctl(NE_fd,  TIMER_PERIOD,  &ival)  <  0)
  {
      perror("setting  timer  period");
     exit(-1);
  }
   /*  Reset  the  timer  and  disable  timer  interrupt  */
   if  (ioctl(NE_fd,  TIMER_RESET)  <  0)
  {
      perror("resetting  timer");
     exit(-1);
  }
 /*  Set  up  a  30  sec  task  timeout  for  high-skew  execution  */
   ival  =  30000000;  /*  30  sec  task  timeout  */
   if  (ioctl(NE_fd,  NE_EXCH_TIMELIMIT,  &ival)  <  0)
  {
      perror("setting  exchange  time  limit");
     exit(-1);
  }
   /*  This  code  causes  all  channels  to  wait  for  a
       synchronous  "go"  command  from  the  user  interface.  */
   socksync();/**/

   /*  This  code  causes  the  NEs  to  become  synchronized  */
   /*  It  also  starts  the  timer  and  enables  timer  rupt  */
   if  (ioctl(NE_fd,  NE_ISYNC,  &pvect)  <  0)
  {
     perror("isync");
     exit(-1);
  }

   printf("ISYNC  complete\n");
  ETHinit();
  /* Setup for Avrada */
  ptr = (int *) init(argc, argv);

  switch(argc) {

  case 0:
    pb.save_flag     = 0;
    pb.playback_flag = 0;
    break;

  case 1:
    pb.save_flag     = atoi(argv[0]);
    pb.playback_flag = 0;
    break;

  case 2:
    pb.save_flag     = atoi(argv[0]);
    pb.playback_flag = atoi(argv[1]);
    break;
  }

  strcpy(pb.save_file_name,"pbd");
  sprintf(set,"%d", pb.save_flag );
  strcat(pb.save_file_name,set);
  strcat(pb.save_file_name,".dat");

  strcpy(pb.playback_file_name,"pbd");
  sprintf(set,"%d", pb.playback_flag );
  strcat(pb.playback_file_name,set);
  strcat(pb.playback_file_name,".dat");

  dynapath();                                /* Execute Dynapath */

  printf("dynapath task terminated with status %d\n",dyna_status);

   /*  All  done.  Close  the  NE  device  driver.  */
   close(NE_fd);
}
/*  This  is  the  body  of  the  exch()  routine  as  used  in
     Dynapath  demonstration.  */
void  exch(char  *raw,  char  *voted,  int  class,  int  size)
{
   struct  exch_struct  dyna_exch;
   int ival;

   dyna_exch.raw_data = raw;
   dyna_exch.voted_data = voted;
   dyna_exch.class = class;
   dyna_exch.size = size;
   dyna_exch.sync_flag = TRUE;
   /* Perform the exchange */
   if( ioctl(NE_fd, NE_EXCHANGE, &dyna_exch) < 0)
   {
       perror("NE_EXCHANGE");
      exit(-2);
   }
   /* Exchange done...redundant copies synchronized. */
   /*
      The current ioctl exchanges data in place,
     from raw to raw.
      The application expects the voted data to be in voted.
      Therefore must bcopy from raw to voted.
      Note that raw is !clobbered! by the ioctl.
     */
   bcopy (raw,voted,size);
/* This code reduces the timeout if the skew-inducing way-
point file read has been completed. */
   if (waypointread && !timeoutlowered)
   {
      ival = 100000; /* 100 milli-seconds task timeout */
      if (ioctl(NE_fd, NE_EXCH_TIMELIMIT, &ival) < 0)
     {
         perror("setting exchange time limit");
        exit(-1);
     }
      timeoutlowered = TRUE;
   }
}
```

Figure 38. Dynapath's dutils.c NE Interface Code

5.8.3. Use of NE Interface by Dynapath's Ethernet Communications Procedures

This section illustrates the use of the NE exch() primitive by Dynapath's Ethernet Communications Procedures resident in the file "comm.c." Two uses of the exch() primitive are demonstrated. The first is in the routine "get_helsim_data," in which a single-source exchange (CLASS2A) is used first to exchange the status of the incoming Ethernet buffer (i.e., whether data are present), and second, to exchange the actual data if present. The second use of exch() is in the "send_hud" routine, where the Dynapath output data emanating from the redundant Dynapath executions are voted prior to being sent to the SG workstation.

The NE-specific calls are printed in boldface and enclosed in boxes.

```
/*********************************************************************/
/* Code to be linked with Andre's code on the VME in order to        */
/* communicate with the Silicon Graphics machine displaying the HUD   */
/* and the PC displaying the map.                                     */
/* This module contains integer functions ETHinit, send_hud,          */
/* send_map, get_data_rec, and ETHclose.                              */
/*********************************************************************/
#include "ethernet.h"   /* file containing AVRADA's ethernet addresses
*/
#include "common.h"
#include "heldata.h"
#include  "ne_drvr.h"
#include <stdio.h>      /* Standard C I/O include file            */
#include <fcntl.h>      /* File constants for opening TCP ports */
#include <errno.h>      /* System Error numbers and constants */
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
char comm_c[] = "@(#)M% 1.1 10/28/91";

#define     FALSE 0
#define     ETH_PORT    1510
#define HEL_ETH         1500
/*********************************************************************/
/* L O C A L   D A T A                                               */
/*********************************************************************/
int   send_sock = -1;         /* The Send socket       */
int   recv_sock = -1;         /* The Receive socket    */
int   hel_sock = -1;          /* receive socket for ethernet */
int   size;               /* Size of ethernet address structure */
static      struct      sockaddr_in src,  /* The source address       */
                        dst;  /* The destination address  */
static struct sockaddr_in     hel;  /* helsim address       */
static      unsigned long host_address;   /* Network address of this ma-
chine */
static      unsigned long broadcast_address; /* Network broadcast ad-
dress  */
/*********************************************************************/
/* ETHinit  Initializes ethernet send end recieve ports.             */
/* Returned Value: 0 if successful, -1 if not                        */
/*********************************************************************/
int ETHinit()
{
int   tcp_error,  /* Error number for opening TCP port */
      fd;         /* Temporary file descriptor */
fd = socket(AF_INET,SOCK_DGRAM,0);
tcp_error = errno;
close(fd);
if(!tcp_error||tcp_error==EACCES) {
      return(tcp_init());
      }
else {
      return(-1);
      }
}
/*********************************************************************
*/
```

```
/* Function to get data from helsim                                    */
/************************************************************************
*/
int   get_helsim_data(struct heldata *msg)
{
  int rval=0;
  static int   r_msize = -1;
  static int      msize = -1;
  static struct heldata r_msg;
  int dcount = 0;
  do {
    rval = recvfrom(hel_sock,(char *)(&r_msg),sizeof(struct heldata),
                0,(struct sockaddr *)&hel,&size);
    if(rval > 0)
    {
      dcount++;
      r_msize = rval;
    }
  } while((rval>0)&&(ntohl(hel.sin_addr.s_addr)==host_address));


    /*  Exchange  status  and,  conditionally,  data  for  incoming
        Ethernet  socket  used  by  Channel  A  */
    exch(&r_msize,  &msize,  CLASS2A,  sizeof(r_msize));
   if(msize  >  0)
   {
       exch(&r_msg,  msg,  CLASS2A,  sizeof(struct  heldata));
   }
  if (dcount > 0)
    printf("got %d pkts\n",dcount);

  return(msize);
}
int send_hud(struct dynapath_output *msg)
{
/*  dst.sin_addr.s_addr = APOLLO;    /*  */
/*  dst.sin_addr.s_addr = MAXWELL; /*  */
  dst.sin_addr.s_addr = ONYX; /*  */
    /*  Vote  outgoing  data  before  sending  to  helsim.  */
     exch(msg,msg,CLASS1,sizeof(struct  dynapath_output));
  return(sendto(send_sock,msg,sizeof(struct dynapath_output),0,(struct
sockaddr *)&dst,sizeof(dst)) );
}
/*******************************************************************/
/* ETHclose  Close the socket descriptors.                          */
/* Returned Value:   0                                              */
/*******************************************************************/
ETHclose()
{
close(send_sock);
close(recv_sock);
send_sock = -1;
recv_sock = -1;
return(0);
}
/***************************************************************/
/*< tcp_get_addrs   Get the host and broadcast addresses      >*/
/*                                                             */
/* host_addr    O   The address of this machine.               */
/*                                                              */
/* Returned Value: 0 if successful; (-1) otherwise.            */
/***************************************************************/
int   tcp_get_addrs( host_addr )
   unsigned long *host_addr;
{
   int s;                           /* Socket descriptor */
   struct ifreq *ifp;               /* Pointer to interface info */
   /***********************/
   /* Get the host address */
   /***********************/
   {
      char host_name[32];
      struct hostent *h;
      if (gethostname( host_name, sizeof( host_name ) ) < 0) {
       perror( "Getting host name" );
       return( -1 );
      }
      if (!(h = gethostbyname( host_name ))) {
       perror( "Getting host by name" );
       return( -1 );
      }
      *host_addr = ntohl( *(u_long *) h->h_addr );
   }
return( 0 );
}
/*********************************************************************/
/* tcp_init   Initialize a communication sockets.                   */
/*                                                                   */
/* Returned Value:   0 if the socket is created without error;      */
/*                  -1 otherwise.                                    */
/*********************************************************************/
int   tcp_init()
{
int   bsize;
int     opt;              /* Value of socket options          */
   /*****************************************************/
   /* Check to see if initialization has already been done */
   /*****************************************************/
   if ((send_sock != -1) || (recv_sock != -1))
      return( 0 );
   /*****************************************************************/
   /* Determine host and broadcast addresses                        */
   /*****************************************************************/
   if (tcp_get_addrs( &host_address ) < 0)
      return( -1 );
   /*****************************************************************/
   /* Open Send socket                                              */
   /*****************************************************************/
   if ((send_sock = socket( AF_INET, SOCK_DGRAM, 0)) < 0) {
      perror( "Opening Send socket" );
      return( -1 );
   }
   /*****************************************************************/
   /* Enable broadcasting on Send socket                            */
   /*****************************************************************/
```

```
#ifdef SO_BROADCAST
   printf("SO_BROADCAST DEFINED\n");
   opt = 1;
   if ( setsockopt( send_sock, SOL_SOCKET, SO_BROADCAST,
                    &opt, sizeof(opt) ) < 0 ) {
      perror( "Setting Send socket options" );
      return( -1 );
   }
#endif
   /*********************************************************************/
   /* Set up destination address to send to                             */
   /*********************************************************************/

   dst.sin_family = AF_INET;
   dst.sin_addr.s_addr = htonl( broadcast_address );
   dst.sin_port = HEL_ETH; /*ETH_PORT;      */
   /*********************************************************************/
   /* Receive socket                                                    */
   /*********************************************************************/

   if ((recv_sock = socket( AF_INET, SOCK_DGRAM, 0 )) < 0) {
      perror( "Opening Receive socket" );
      return( -1 );
   }
   if((hel_sock = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
   {
      perror("Opening helsim socket");
      return(-1);
   }
   /*********************************************************************/
   /* Identify Receive socket as non-blocking                           */
   /*********************************************************************/

   opt = 1;
   if (fcntl(recv_sock,F_SETFL,O_NDELAY)<0) {
      perror( "Setting non-blocking Receive socket" );
      exit( -1 );
   }
  if(fcntl(hel_sock,F_SETFL,O_NDELAY) <0)
  {
      perror("Setting non-blocking on helsim socket");
      exit(-1);
  }
   /*********************************************************************/
   /* Set up source address to receive from                             */
   /*********************************************************************/

   src.sin_family = AF_INET;
   src.sin_addr.s_addr = htonl( INADDR_ANY );
   src.sin_port = ETH_PORT;
   /*********************************************************************/
   /* Bind Receive socket                                               */
   /*********************************************************************/

   bind( recv_sock, (struct sockaddr *) &src, sizeof( src ) );
   hel.sin_family = AF_INET;
   hel.sin_addr.s_addr = htonl(INADDR_ANY);
   hel.sin_port = HEL_ETH;
```

```
    bind(hel_sock, (struct sockaddr *) &hel, sizeof(hel));
    size = sizeof(struct sockaddr);
    return (0);
}
```

Figure 39. Use of NE Interface Calls by Dynapath Ethernet I/O Procedures

6. References

[Abl88]	Abler, T., A Network Element Based Fault Tolerant Processor, MS Thesis, Massachusetts Institute of Technology, Cambridge, MA, May 1988.
[AMD89a]	The SUPERNET Family for FDDI, Advanced Micro Devices Data Book, Publication # 09734 Rev. C, February 1989.
[AMD89b]	Am7968/Am7969-175 TAXIchipTM Integrated Circuits, Advanced Micro Devices Data Sheet, Publication # 12834 Rev. A, November 1989.
[ANSI139]	"Fiber Distributed Data Interface (FDDI) - Token Ring Media Access Control (MAC)" American National Standard, ANSI X3.139-1987, November 5, 1986.
[ANSI148]	"Fiber Distributed Data Interface (FDDI) - Token Ring Physical Layer Protocol (PHY)," American National Standard, ANSI X3.148-1988, June 30, 1988.
[ANSI166]	"Fibre Data Distributed Interface (FDDI) - Token Ring Physical Layer Medium Dependent (PMD)," American National Standard, ANSI X3.166-1990, September 28, 1989.
[APS90]	Acarlar, M. S., Plourde, J. K., Snodgrass, M. L., "A High Speed Surface-Mount Optical Data Link for Military Applications," IEEE/AIAA/NASA 9th Digital Avionics Systems Conference Proceedings, October 15-18, 1990, p. 297-302.
[Bab90a]	Babikyan, C., "The Fault Tolerant Parallel Processor Operating System Concepts and Performance Measurement Overview," Proceedings of the 9th Digital Avionics Systems Conference, October 1990, pp. 366-371.
[Ber87]	Bertsekas, D., Gallager, R., Data Networks, Prentice-Hall, 1987.
[Ber90]	Berger, K. M., Abramson, M. R., Deutsch, O. L., "Far-Field Mission Planning for Helicopters," CSDL Technical Report CSDL-R-2234, March 1990.
[Bev90]	Bevier, W.R., and Young, W.D., "The Proof of Correctness of a Fault-Tolerant Circuit Design," 2nd International Working Conference on Dependable Computing for Critical Applications, Tucson, AZ, February 1991.

[Bic90] Bickford, M., and Srivas, M., "Verifying an Interactive Consistency Circuit: A Case Study in the Reuse of a Verification Technology," NASA Formal Methods Workshop 1990, NASA Conference Publication 10052, November 1990.

[Biv88] Bivens, G. A., "Reliability Assessment of Surface Mount Technology (SMT)," RADC report RADC-TR-88-72, March 1988.

[Bla91] Black, Uyless, OSI : A Model For Computer Communications Standards, Prentice-Hall, 1991.

[Boo88] Booth, F., "Advanced Apache Architecture," 8th Digital Avionics Systems Conference, October 1988.

[Bur89] Burkhardt, L., Advanced Information Processing System: Local System Services, NASA Contractor Report 181767, April 1989.

[But88] Butler, R. W., "A Survey of Provably Correct Fault Tolerant Clock Synchronization Techniques," NASA TM-100553, NASA Langley Research Center, February 1988.

[CAMP] CAMP-1 Final Technical Report AFATL-TR-85-93, 3 Volumes, Available as DTIC AD-B102 654, AD-B102 655, and AD-B102 656 from Defense Technical Information Center, Alexandria, VA 22304-6145.

[Car84] Carlow, G. D., "Architecture of the Space Shuttle Primary Avionics Software System", Communications of the ACM, 27(9):926-36, September 1984.

[Cha84] Chambers, F. B., ed., Distributed Computing, Academic Press, 1984.

[Che87] Cheng, S-C., Stankovic, J. A., Ramamritham, K., "Scheduling Algorithms for Hard Real-Time Systems - A Brief Survey," in Hard Real-Time Systems, IEEE Computer Society Press, 1988.

[Coh87] Cohn, Marc D., "The Conformance of the ANSI FDDI Standard to the SAE-9B HART High Speed Data Bus Requirements for Real-Time Local Area Networks," Society of Automotive Engineers Aerospace Systems Conference Proceedings, November 1987.

[Coh88] Cohn, Marc D., "The Fiber Optic Data Distribution Network: A Network for Next-Generation Avionics Systems," AIAA/IEEE 8th Digital Avionics Systems Conference Proceedings, October 17-20, 1988, p. 731-737.

[Coh90a] Cohen, G. C., et. al., Design of an Integrated Airframe/Propulsion Control System Architecture" NASA Contractor Report 182004, March 1990.

[Coh90b] Cohen, G. C., et. al., Final Report: Design of an Integrated Airframe/Propulsion Control System Architecture, NASA Contractor Report 182007, March 1990.

[Cohn88] Cohn, A., "Correctness Properties of the Viper Block Model: The Second Level," Tech. Report 134, Univ. of Cmabridge, Cambridge, England, May 1988.

[Com91] Comer, D. E., Internetworking with TCP/IP, Prentice-Hall, 1991.

[CSDL9214] Completion of the Advanced Information Processing System, response to NASA Langley Research Center, CBD Announcement REF SS017, issue PSA-9214, November 12, 1986.

[Cullyer 88] Cullyer, W. J., "Implementing Safety-Critical Systems: The VIPER Microprocessor," VLSI Specification, Verification and Synthesis, Kluwer Academic Publishers, 1988.

[CVC2] "System Specification for Combat Vehicle Command and Control (DRAFT)," CVC2 Systems Implementation Working Group, 31 October 1990.

[DACS] Defense & Analysis Center for Software, Kaman Sciences Corporation, P.O. Box 120, Utica, NY 13503.

[Dal73] Daly, W.M., Hopkins, A.L., and McKenna, J.F., "A Fault-Tolerant Digital Clocking System," 3rd International Symposium on Fault Tolerant Computing, Palo Alto, CA, June 1973.

[Deu88] Deutsch, O. L., Desai, M., "Development and Demonstration of an On-Board Mission Planner for Helicopters," CSDL Technical Report CSDL-R-2056, April 1988.

[DID80811] "VHSIC Hardware Description Language (VHDL) Documentation," Data Item Description, DD Form 1664, DI-EGDS-80811, May 11, 1989.

[DiV90] Di Vito, B. L., Butler, R. W., Caldwell, J. L., Formal Design and Verification of a Reliable Computing Platform for Real-Time Control, NASA Technical Memorandum 102716, October 1990.

[DiV91] Di Vito, B., Butler, R., and Caldwell, J., "High Level Design Proof of a Reliable Computing Platform," 2nd International Working Conference on

Dependable Computing for Critical Applications, Tucson, AZ, February 1991.

[Dol82] Dolev, D., "The Byzantine Generals Strike Again," Journal of Algorithms, Vol. 3, 1982, pp. 14-30.

[Dol84] Dolev, D., Dwork, C., Stockmeyer, L., "On the Minimal Synchronism Needed for Distributed Consensus," IBM Research Report RJ 4292 (46990), 5/8/84.

[Fel90] Felter, S. C., Douglas, P. H., Smith, C. A., "Avionics System Integration for the MH-53J Helicopter," 9th Digital Avionics Systems Conference, October 1990.

[Fis82] Fischer, M. J., Lynch, N. A., "A Lower Bound for the Time to Assure Interactive Consistency," Information Processing Letters, Vol. 14, No. 4, 13 June 1982, pp. 183-186.

[Foh89] Fohler, G., Koza, C., "Heuristic Scheduling for Distributed Real-Time Systems," Research Report No. 6/89, Institut fur Technische Informatik, Technische Universitat Wien, Vienna, Austria, April 1989.

[Gal90] Galetti, R. R., Real-Time Digital Signatures and Authentication Protocols, Master of Science thesis, Massachusetts Institute of Technology, May 1990.

[Goe91] Goel, A.L., and Sahoo, S.N., "Formal Specifications and Reliability: An Experimental Study," 1991 International Symposium on Software Reliability Engineering, Austin, Texas, May 1991.

[Gua90] Guaspari, D., Marceau, C., and Polak, W., "Formal Verification of Ada Programs," IEEE Transactions on Software Engineering, Special Issue on Formal Methods in Software Engineering, Vol. 16, No. 9, September 1990.

[Han89] Hanaway, J. F., Morrehead, R. W., Space Shuttle Avionics System, NASA SP-504, 1989.

[Har87] Harper, R., Critical Issues in Ultra-Reliable Parallel Processing, PhD Thesis, Massachusetts Institute of Technology, Cambridge, MA, June 1987.

[Har88a] Harper, R., Lala, J., Deyst, J., "Fault Tolerant Parallel Processor Overview," 18th International Symposium on Fault Tolerant Computing, June 1988, pp. 252-257.

[Har88b] Harper, R., "Reliability Analysis of Parallel Processing Systems," Proceedings of the 8th Digital Avionics Systems Conference., October 1988, pp. 213-219.

[Har91a] Harper, R., Lala, J., Fault Tolerant Parallel Processor, J. Guidance, Control, and Dynamics, V. 14, N. 3, May-June 1991, pp. 554-563.

[Har91b] Harper, R., Alger, L., Lala, J., "Advanced Information Processing System: Design and Validation Knowledgebase," NASA Contractor Report 187544, September 1991.

[Hir90] Hird, G.R., "Formal Methods in Software Engineering," 9th AIAA/IEEE Digital Avionics Systems Conference, Virginia Beach, VA, October 1990, pp. 230-234.

[Hun86] Hunt, W.A., "FM8501: A Verified Microprocessor," Proceedings of IFIP Working Group 10.2 Workshop, North Holland, Amsterdam, 1986.

[Hwa84] Hwang, K., Briggs, F., Computer Architecture and Parallel Processing, McGraw-Hill, 1984.

[IEEE1076] "VHDL Language Reference Manual," IEEE Standard, IEEE Std 1076-1987, March 31, 1988.

[IEEE8021] "Local and Metropolitan Area Networks: Overview and Architecture," IEEE Standard, IEEE Std 802-1990, May 31, 1990.

[IEEE8022] "Logical Link Control," IEEE Standard, IEEE Std 802.2-1989, August 17, 1989.

[IEEE8023] "Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications," IEEE Standard, IEEE 802.3-1988, June 9, 1988.

[IEEE8024] "Token-Passing Bus Access Method and Physical Layer Specifications," IEEE Standard, IEEE 802.4-1990.

[J88N2] "Linear Token Passing Multiplex Data Bus Protocol," Joint Integrated Avionics Working Group Standard, Document J88-N2.

[J8701] "Advanced Avionics Architecture (A3) Standard," Joint Integrated Avionics Working Group Standard, Document J87-01.

[Klj89] Kljaich, J., Jr., Smith, B.T., and Wojcik, A.S., "Formal Verification of Fault Tolerance Using Theorem-Proving Techniques," IEEE Transactions on Computers, Vol. 38, No. 3, March 1989.

[Kop89] Kopetz, H., et. al., "Distributed Fault-Tolerant Real-Time Systems: The MARS Approach," IEEE Micro, 9(1):25-40, February 1991.

[Kop91] Kopetz, H., et. al., "The Rolling Ball on MARS," Institut fur Technische Informatik Research Report No. 13/91, Technische Universitat Wien, Vienna, Austria, November 1991.

[Kri85] Krishna, C. M., Shin, K. G., Butler, R. W., "Ensuring Fault Tolerance of Phase Locked Clocks," IEEE Trans. Computers, Vol. C-34, No. 8, August, 1985.

[Lal84] Lala, J. H., "An Advanced Information Processing System," 6th AIAA-IEEE Digital Avionics Systems Conference, Baltimore, MD, Dec. 1984.

[Lal84] Lala, J. H., "An Advanced Information Processing System," 6th AIAA-IEEE Digital Avionics Systems Conference, Baltimore, MD, December 1984.

[Lal85] Lala, J. H., "Advanced Information Processing System: Fault Detection and Error Handling," AIAA Guidance, Navigation and Control Conf., Snowmass, CO, Aug. 1985.

[Lal86a] Lala, J.H., "Fault Detection, Isolation, and Reconfiguration in the Fault Tolerant Multiprocessor," Journal of Guidance, Control, and Dynamics, Sept-Oct. 1986.

[Lal86b] Lala, J. H., "A Byzantine Resilient Fault Tolerant Computer for Nuclear Power Plant Applications," 16th Annual International Symposium on Fault Tolerant Computing Systems, Vienna, Austria, 1-4 July 1986.

[Lal89] Lala, J.H., et. al., "Study of a Unified Hardware and Software Fault Tolerant Architecture," NASA Contractor Report 181759, January 1989.

[Lal91] Lala, J.H., R. Harper, K. Jaskowiak, G. Rosch, L. Alger, and A. Schor "AIPS for Advanced Launch System: Architecture Synthesis Report", NASA Contractor Report 187544, September 1991.

[Lam85] Lamport, L., Melliar-Smith, P. M., "Synchronizing Clocks in the Presence of Faults," Journal of the ACM, 32(1):52-78, January 1985.

[Lap90] "Dependability: Basic Concepts and Terminology," J.C. Laprie - Editor, Published by International Federation for Information Processing (IFIP) Working Group 10.4 on Dependable Computing and Fault Tolerance, December 1990.

[Leh87] Lehoczky, Sha, Ding, The Rate Monotonic Scheduling Algorithm - Exact Characterization and Average Case Behavior, Technical Report, Department of Statistics, Carnegie-Mellon University, 1987.

[Leh89] T. Lehr, et. al., "Visualizing Performance Debugging", *Computer*, October 1989, pp. 38-51.

[Liu73] Liu, C. L., Layland, J. W., "Scheduling Algorithms for Multiprograming in a hard Real-time Environment," J. ACM, 20(1):46-61, 1973.

[LSP82] Lamport, L., Shostak, R., Pease, M., "The Byzantine Generals Problem," ACM Transactions on Programming Languages and Systems, Vol. 4, No. 3, July 1982, p. 382-401.

[MA-HDBK] Modular Avionics Handbook, Document No. 21530(0-6), FSCM 51993, Draft C, U. S. Air Force ASD-ALD/AX, 19 April 1990.

[Ma78] Martin, D. L., Gangsaas, D., "Testing of the YC-14 Flight Control System Software," AIAA Journal of Guidance, Control, and Dynamics, Vol. 1, No. 4, July-August 1978.

[McE88] McElvany, M. C., "Guaranteeing Deadlines in MAFT," IEEE Real-Time Systems Symposium, Huntsville, AL, December 1988.

[MIL-HDBK-0036] "Survivable Adaptable Fiber Optic Embedded Network II - SAFENET II," Military Handbook, MIL-HDBK-0036, 1 March, 1990.

[MIL-HDBK-59] MIL-HDBK-59, "Computer-Aided Acquisition and Logistic Support (CALS) Program Implementation Guide," 20 December 1988.

[MIL-HDBK-217E] MIL-HDBK-217E, "Reliability Prediction of Electronic Equipment," 2 January 1990.

[MIL-STD-344] MIL-STD-344 (draft), "Standard Army Vetronics Architecture," 14 September, 1990.

[MIL-STD-785B] MIL-STD-785B, "Reliability Program for Systems and Equipment Development and Production," 15 September 1980.

[MIL-STD-1553] "Aircraft Internal Time Division Command/Response Multiplex Data Bus," Military Standard, MIL-STD-1553B, 12 February, 1980.

[MIL-STD-1815A] MIL-STD-1815A, "Reference Manual for the Ada Programming Language," 17 February 1983.

[Osd88] Osder, S. S., "Digital Fly-by-Wire System for Advanced AH-64 Helicopters," 8th Digital Avionics Systems Conference, October 1988.

[Pal85] D. Palumbo and R. Butler, "Measurement of SIFT Overhead", NASA Technical Memorandum 86722, Langley Research Center, Hampton, VA, April 1985.

[Pe80] Pease, M., Shostak, R., Lamport, L., "Reaching Agreement in the Presence of Faults," Journal of the ACM, Vol. 27, No. 2, April 1980, pp. 228-234.

[PEI90120] XTP® Protocol Definition, Revision 3.5, Published by Protocol Engines Inc., September 1990.

[Pek88] Pekelsma, N. J., "Optimal Guidance with Obstacle Avoidance for Nap-of-the Earth Flight," NASA Contractor Report 177515, December 1988.

[Pus89] Puschner, P., Koza, C., "Calculating the Maximum Execution Time of Real-Time Programs," Real-Time Systems, 1(2):159-176, September 1989.

[Rad90] PMV 68 CPU-3A Specification, Issue 3, Publication No. 681/SA/04085, Radstone Technology plc, 1990.

[Rus89] Rushby, J., von Henke, F., "Formal Verification of a Fault Tolerant Clock Synchronization Algorithm," NASA Contractor Report 4239, June 1989.

[SAE91] SAE/AS-2A Subcommittee RTMT Statement on Requirements for Real-Time Communication Protocols (RTCP), Issue #1, SAE ARD50007, August 2 1991.

[San90] STAR MVP Technical Description, Document No. 4069718, Lockheed Sanders, 25 June 1990.

[Sch91] Schutz, W., "On the Testability of Distributed Real-Time Systems," Proc. Tenth Symposium on Reliable Distributed Systems, Pisa, Italy, September, 1991.

[Spi89] Spivey, J.M., The Z Notation, A Reference Manual, Prentice Hall International (UK) Ltd, 1989.

[Spi90] Spivey, J.M., "Specifying a Real-Time Kernel," IEEE Software, Special Issue on Formal Methods, Vol. 7, No. 5, Sep 1990.

[Sri90] Srivas, M. and Bickford, M., "Formal Verification of a Pipelined Microprocessor," IEEE Software, Special Issue on Formal Methods, Vol. 7, No. 5, September 1990.

[Sta87] Stankovic, J. A., Ramamritham, K., "The Design of the Spring Kernel," Proc. of the Real Time Systems Symposium, December 1987.

[Sun74] Sundstrom, R. J., "On-Line Diagnosis of Sequential Systems," PhD Thesis, University of Michigan, 1974.

[Tan88] Tanenbaum, A. S., Computer Networks, second edition, Prentice-Hall, 1988.

[X3T95] "FDDI Station Management (SMT)," Preliminary Draft Proposed American National Standard, X3T9.5/84-49, Rev. 6.2, May 18, 1990.

7. Glossary of Terms and Acronyms

AFTA-Army Fault-Tolerant Architecture-A computer designed for both high reliability and high throughput. The AFTA is based on the FTPP architecture.

aperiodic tasks-A set of tasks whose iteration rates are unknown or undefined.

ASIC-Application Specific Integrated Circuit-A type of integrated circuit that can be custom designed by the hardware engineer so that it will perform a particular logic or processing function and at the same time save circuit board space and power consumption. The advent of VLSI design techniques has made ASICs a more flexible and practical option for hardware designers.

ATP-Authentication Protocol-A protocol utilized by the BRNP to sign outgoing packets and to test the authenticity of incoming packets.

ATPG-Automatic Test Pattern Generation-The generation of test vectors directly from a netlist for verification of device functionality. Test vectors from an ATPG program do not test the correct functionality of the device; they only test that the device is a correct implementation of the design as specified by the netlist.

behavioral VHDL is defined to be a VHDL architecture which uses any of the legal VHDL constructs, including those which do not correspond to possible hardware realizations of the description (i.e., pure behavioral may not be synthesizeable). A level of description that specifies a device functionally in terms of output reactions to input stimulus. A behavioral description can also specify the timing relationships of inputs to outputs.

BIT-Built In Test-This is an internal diagnostic testing system that is included as part of the AFTA design. There are three forms of the BIT-- I-BIT is the initial power-on test system, M-BIT is for maintenance testing, C-BIT is the continuous in-flight test system.

BRNP-Byzantine Resilient Network Protocol-A network layer protocol which implements the Byzantine Resilient Virtual Circuit in order to guarantee that all messages are delivered accurately.

broadcast addressing-A method of station addressing using an identifier that causes all stations to respond to the specified address.

bypass-The ability to effectively isolate a node from the network without disrupting the continuity of the network.

Byzantine Resilient-Capable of tolerating Byzantine faults. A Byzantine Resilient system is capable of handling arbitrarily malfunctioning components that may supply faulty information to other parts of the system thereby causing a spread of faulty information within the system.

C3-Cluster 3-An FTPP model number. Composed of either 4 or 5 FCRs, 3-40 processors, 1-40 VIDs, simplex, triplex, and quadruplex processor redundancy levels. Previous FTPP models were C1 (4 FCRs, 16 processors, 4-16 VIDs, simplex, duplex, triplex, and quadruplex processor redundancy levels) and C2 (4 FCRs, 4 processors, one fixed quad VID).

cache-A form of memory that is typically much faster and much smaller than main memory. Through utilization of cache memory, a processor's throughput will be increased. Typically cache memory acts as a staging area for data; information will be pulled from main memory and temporarily stored in cache while it undergoes processing.

CDU-Cockpit Display Unit-A cathode ray tube display located in the vehicle cockpit for display of system status. The CDU may display overall AFTA system status, LRU level status, or LRM level status.

CID-Communication Identification-A designation assigned to each task which is used for intertask communication.

class test-A test of the Network Element voting mechanism that requests a non-congruent message exchange selectively on each channel of a fault masking group.

cluster-An FTPP consisting of 4 or 5 FCRs containing at least one virtual processing site. Multiple clusters could be connected by a network device (such as a fault-tolerant data bus) to provide even greater throughput than a single cluster. Most references to an FTPP refer to a single cluster design.

CMF-Common Mode Fault-A type of malfunction which will cause multiple faults or complete execution failure within a redundant processing group. Common mode faults may result from software flaws, hardware bugs, design flaws, massive electrical upsets etc.

concurrent I/O-Input/Output processes that allow the associated virtual group to perform other tasks while I/O is collecting data. This allows for greater processor throughput.

CRC-Cyclic Redundancy Check-An error detecting code used in data communications that allows the unit receiving a message to ensure through binary mathematics that it is the same message sent by the transmitting unit.

CSMA/CD-Carrier Sense Multiple Access with Collision Detection-A form of media access control whereby a potential transmitting station will monitor the bus to ensure that it is clear before transmission begins. During transmission, the station also monitors the bus to check for message collisions. If a collision occurs, the message must be re-transmitted.

CT-Configuration Table-A table stored on the Network Element that contains the current configuration of the system, i.e. which processors are members of which virtual groups.

DAIS-Digital Avionics Instruction Set-A benchmark for measuring processor throughput.

depot test-A set of diagnostic level tests executed outside of the constraints of a real-time environment with emphasis on the isolation of chip level faults in these components. These tests would occur at a maintenance repair facility in contrast to the various forms of built-in testing.

DPRAM-Dual-Port Random Access Memory-The type of memory that occupies the data segment. It provides a buffer between the NE and the PE; both the NE and the PE may access the data segment asynchronously, provided that they do not attempt to access the same location.

DR-Discrepancy Report-A report that is filed whenever unexpected behavior of the hardware, software, or system is encountered. By recording observable symptoms of the system throughout testing, integration, verification and validation, one may better trace and identify system flaws.

entity-A specific instance of a protocol element in an Open Systems Interconnection layer or sublayer.

FCR-Fault Containment Region-Usually comprised of a number of line replaceable modules such as Processing Elements, Network Elements, input/output controller, and power conditioners. The AFTA is made up of four or five FCR's, and each FCR usually resides on a single circuit board (with the exception of the power conditioner). An interchangeable term for the FCR is Line Replaceable Unit or LRU.

FDDI-Fiber Distributed Data Interface-A networking standard developed by the American National Standards Institute to provide high bandwidth for Local Area Networks.

FDIR-Fault Detection, Identification and Recovery-FDIR software designed for the AFTA allows it to sustain multiple successive faults by identifying a faulty component and reconfiguring the AFTA system operation to compensate for the fault.

FIFO-First In First Out-A type of information buffer in which the data that is stored first chronologically will be the first to be extracted.

FMEA-Failure Modes and Effects Analysis

FMG-Fault Masking Group-A logical grouping of three or four processors to enhance the reliability of critical tasks. The members of an FMG execute the same code with the same data and periodically exchange messages to ensure that they produce the same outputs.

FTC-Fault Tolerant Clock-A distributed digital phase-locked loop used for synchronization of AFTA fault containment regions.

FTDB-Fault Tolerant Data Bus-A local area network designed around principles of Byzantine resilience. Its primary objective is to provide an optimal internetworking system between simplex and redundant processing sites.

FTNP-Fault Tolerant Navigation Processor-The initial ground vehicle application for the AFTA is for the navigations system in Armored Systems Modernization vehicles.

FTPP-Fault-Tolerant Parallel Processor-A computer designed for both high reliability and high throughput. The core of the FTPP is the Network Element.

functional reliability-The probability that a given function can be executed because its resources are operational.

functional synchronization-In maintaining synchronous operation, the members of a VID perform a synchronizing act after some sequence of functions has been completed. The sequence of functions between the synchronization points is referred to as a frame.

GC-Global Controller-A microcoded finite-state machine used to coordinate the functions throughout the Network Element.

graceful degradation-Through self-testing, a virtual group may identify a faulty member and gracefully degrade its redundancy level using a configuration table update message to eliminate the faulty channel.

IOC-Input/Output Controller-These devices connect the AFTA to the outside world, and they must be compatible with the bus connecting elements of the FCR. They may have a programmable processor on board to drive the I/O, or they may require off-board processors for operation.

IPS-Instructions Per Second-The number of machine language instructions that a processor will execute every second. This measurement is used to reference the speed of the processor.

ISO/OSI-International Standards Organization/Open Systems Interconnection-A specification and model for computer communication networks.

LAN-Local Area Network-A network topology that interconnects computer systems separated by relatively short distances (2-2000 meters). LAN technology is usually based on a shared medium with no intermediate switching nodes required.

leaf-level-(VHDL) The models at the bottom of the model tree. Leaf-level models in VHDL are always pure behavioral models.

LERP-Local Exchange Request Pattern-A string of bytes describing the current state of the input and output buffers for each processor in an FCR. The LERP is used to generate the SERP. Each FCR has a different configuration, therefore the LERPs for each FCR will be different. For this reason, LERPs must be treated as single-source data.

link-An element in a physical network that provides interconnection between nodes.

LOC-Loss of Control-This will occur as a result of a failure in any flight critical portion of the Flight Control System. For analysis purposes, LOC will be considered as a total loss of the vehicle.

Local FDI-Each virtual group will exercise its own fault detection and identification processes to monitor failures among its processors. Also, each virtual group may initiate its own recovery options.

logical addressing-A method of station addressing using an identifier that may select a group of stations to respond to the specified address.

LRM-Line Replaceable Module-The physical unit for field diagnosis and repair. Typically it consists of one circuit card assembly with one or more Processing Elements.

LTPB-Linear Token Passing Bus-A media access control method whereby stations pass a token along a virtual ring from one to another. A station may only transmit when it possesses the token.

MDC-Minimum Dispatch Complement-This specifies the absolute minimum level of operability for the AFTA system to be cleared for a sortie.

media access control-The method by which access to the physical network media is limited to a single node so that communications over the media are undisturbed.

media layer-One or more physical layer media. Multiple media layers are physically and electrically isolated from each other to the same degree as a fault-containment region in a fault-tolerant computer. Most traditional LANs use only a single network layer. A Byzantine resilient network usually employs multiple media layers for redundancy.

memory alignment-A process whereby the RAM and registers in each processor of a virtual group are made congruent as part of the resynchronization of a virtual group.

mission reliability-Arithmetically speaking, mission reliability is one minus the probability that failure of the AFTA causes abortion of the mission.

MMC-Minimum Mission Complement-This specifies the minimum level of AFTA operability for the vehicle to continue its mission.

NDI-Non-Developmental Item

NE-Network Element-The hardware device which provides the connectivity between virtual groups. The primary function of the NE is to exchange and vote packets of data provided by the processors. The ensemble of Network Elements forms a virtual bus network to which all virtual groups are connected.

NEID-Network Element ID-The name by which a Network Element is known in the physical AFTA configuration. An NEID refers to a specific Network Element in the system, i.e. the same NEID on different FCRs refers to the same Network Element. The NEID is also used to refer to the FCR in which the referenced Network Element resides. By convention, letters are used to denote the NEID.

netlist-A list defining interconnections of components. Netlists are typically used for designing printed circuit boards or ASICs.

NIU-Network Interface Unit-The connection between a station and the FTDB

node-An element in a physical network that provides the necessary interface between a station and the network media.

nonpreemptible I/O dispatcher-A task on the virtual group that manages the execution of certain I/O instructions that cannot be interrupted.

packet-A block of data consisting of a header, data, and a trailer exchanged between peer protocol entities. The term packet is somewhat generic and is applied at all levels of the protocol hierarchy.

packet-A string of data of fixed or variable length for transmission from one processor to another through an inter-processor network. A message-passing network handles data in packets. The term packet is used here to refer to a fixed-size (64 bytes) block of data which is transmitted by the Network Elements.

PDU-Protocol Data Unit-A fancy name for a packet. PDU is the name used by OSI.

PE-Processing Element-A hardware device which provides a general or special purpose processing site. A minimal PE configuration contains a single processor and local memory (RAM and ROM). PEs may optionally have private I/O, making them a combination PE and IOC.

PEID-Processing Element ID-The name by which a Processing Element is known in the physical AFTA configuration. Each PE in an FCR has a unique PEID. However, the same PEID may be used by another processor in another FCR. A combination of NEID and PEID is used to uniquely identify a single Processing Element within a cluster.

physical addressing-A method of station addressing using a unique identifier such that at most one station responds to the specified address.

PIMA-Portable Intelligent Maintenance Aid-A system resembling a laptop computer which will initiate the maintenance built in testing (M-BIT), interrogate AFTA for fault information logged during a mission, and extract maintenance records for system components.

PMD-Physical layer Medium Dependent-The standard which defines the physical medium that is used for the data communications channel on a network.

presence test-The polling of various components to determine if each is active and synchronized. The testing may be performed on members of virtual groups or on the virtual groups themselves.

primitive-A function or procedure that one entity provides to another. The primitive definition specifies the inputs, outputs, and data formats for the primitive.

PROM-Programmable Read Only Memory-A form of computer memory that will store a permanent copy of one or more subroutines specifically intended for use by a particular microprocessor. PROM's allow for a certain level of hard-wired software control over the processor.

quadruplex-A virtual group consisting of four processing sites.

rate group dispatcher-An RG4 task that is responsible for controlling the execution of the rate group tasks and providing reliable communication between the rate group tasks throughout the system.

Register Transfer Level (RTL) VHDL-A behavioral format which specifies the functionality of a block from the standpoint of random combinational logic and/or synchronous registers. For the purpose of the AFTA NE development, RTL is defined to be synthesizeable behavioral VHDL, that is, a behavioral VHDL description that is suitable for input to a synthesis tool.

reprocurement-The act of obtaining new parts to replace parts in an existing system, or to build additional copies of an existing design.

RG-Rate Group-A set of tasks whose iteration rate is well-defined and whose execution times do not exceed the iteration frame (the inverse of the iteration rate).

RISC-Reduced Instruction Set Computer-A type of microprocessor which utilizes a limited set of machine language instructions to allow for more rapid execution of those instructions and thus greater throughput for the computer.

RTS-Run Time System

SAVA-Standard Army Vetronics Architecture

sequential I/O-Input/Output processes that require the managing virtual group to completely supervise the activity. In other words, the virtual group must block itself until the I/O is finished.

SERP-System Exchange Request Pattern-A string of bytes describing the current state of the input and output buffers for each processor in the system. The SERP is used to determine if packets can be sent from one virtual group to another. The LERP from each FCR is exchanged using a source congruency to generate the SERP. Because the SERP originates from a source congruency exchange, it can be considered congruent throughout all functioning FCRs.

SIFT-Software Implemented Fault Tolerance-System fault tolerance functions achieved primarily through operating system programming rather than primarily through dedicated hardware.

simplex-A virtual group consisting of only one processing site.

single-source data-An element of information which originates from a single point. Examples of single-source data include sensor readings, input values, and syndromes. Single-source data must be distributed to fault-masking groups using a source congruency exchange to maintain Byzantine resilience.

sortie availability-One minus the probability that the vehicle is prevented by the AFTA from beginning a mission at the desired time.

source congruency-A type of exchange used to distribute data from a single source, such as an input device, to members of a fault-masking group. The source congruency, which is also known as a class 2, 2-round exchange, or interactive consistency, is a primary requirement for a Byzantine resilient system.

station-A device connected to a network that can transmit or receive data over the network. Often a station is a processing site. In the FTDB, a station can be a redundant processing site.

structural VHDL-A level of description that specifies a VHDL architecture by defining interconnections of instantiations of VHDL entities . A structural description resembles a conventional netlist.

syndrome-A bit field indicating the observance of unusual behavior somewhere in the system. Syndromes can be used in an attempt to diagnose and repair faults in the system.

System FDI- A process that will coordinate system status and fault information as well as testing and analyzing shared components.

task migration-The movement of a necessary task from a failed processor to another processor within the same fault containment region.

test bench-A model of a test fixture that is used to test a device being designed with VHDL. The test bench is written in VHDL and provides a non-proprietary way of stimulating and monitoring a design in a simulator.

testability-The ability to unambiguously ascertain the functionality of each Line Replaceable Module of the AFTA.

TF/TA/NOE-Terrain Following/Terrain Avoidance/Nap of the Earth-A typical helicopter mission application for which the AFTA will be designed.

THT-Token Holding Timer-A method used with token passing media access protocols to limit the amount of time each station can transmit on the network.

timeout-A value of time used to monitor skew between processors of an FMG. All processors in an FMG should be synchronized to within one timeout value, so if a processor does not respond within the timeout period, that processor is considered faulty, and the other processors will continue uninhibited. Timeouts are necessary on the AFTA to prevent faulty processors from halting the system.

timestamp-A 32-bit quantity that indicates the relative time within the cluster. The Network Element places a timestamp in the input info block for each packet successfully delivered to a virtual group.

TNR-Transient NE Recovery-The procedure by which a Network Element which has suffered a transient fault is reintegrated into the cluster. The first part of TNR is similar to the ISYNC procedure. TNR also specifies the realignment of the Network Element state.

transient recovery policy-A recovery option whereby the faulty component is immediately disabled and an attempt is made to reintegrate the component into the system.

triplex-A virtual group consisting of three processing sites.

validation-The process of demonstrating that an implemented system correctly performs its intended functions under all reasonably anticipated operational scenarios.

validity-In a Byzantine resilient system, a condition in which all functioning members of a fault-masking group are guaranteed to possess correct data. The validity condition also implies the agreement condition.

vehicle reliability-One minus the probability that the vehicle is lost due to failure of the AFTA.

VG-virtual group-A grouping of one or more processors to form a virtual (possibly redundant) single processing site. All processors in a virtual group execute the same instruction stream. If a virtual group has more than one member, those members must reside in different FCRs. Virtual groups of 3 or more members are known as fault-masking groups.

VHDL-VHSIC Hardware Description Language-A language for specifying hardware design. VHDL designs can be expressed in a behavioral or a structural method. VHDL also defines a simulation environment and incorporates an intrinsic sense of time.

VHSIC-Very High Speed Integrated Circuit-A Government-funded project to develop technologies to be applied to new, high speed integrated circuits. The VHSIC Hardware Description Language (VHDL) was developed under the VHSIC program.

VID-Virtual Identifier-The name by which a virtual group is known to the system. Also, sometimes used as a synonym for virtual group.

voted message-A message sent by all members of a redundant processing group. This message type is only used when exact consensus among all redundant members is expected. This is also known as a Class 1 message.

voter test-A test of the Network Element voting mechanism that seeds non-congruent values selectively on each channel of a fault masking group.

WAN-Wide Area Network-A network topology that interconnects computer systems separated by long distances. WAN systems usually use packet switched technology.

watchdog timer-A simple timekeeper that will monitor operations in both the Processing Elements and the Network Elements to keep the hardware and software from wandering into undesirable states.

working group-The set of FCRs in a cluster which are synchronized and in the operational phase. An FCR which suffers a fault drops out of the working group. The working group may attempt to reintegrate the failed FCR into the working group.

WPV-Weight Power Volume-These are physical characteristics used to describe the AFTA.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE June 1994	3. REPORT TYPE AND DATES COVERED Contractor Report

4. TITLE AND SUBTITLE Advanced Information Processing System: The Army Fault-Tolerant Architecture Detailed Design Overview	5. FUNDING NUMBERS C NAS1-18565 WU 505-64-52-53
6. AUTHOR(S) Richard E. Harper, Carol A. Babikyan, Bryan P. Butler, Robert J. Clasen, Chris H. Harris, Jaynarayan H. Lala, Thomas K. Masotto, Gail A. Nagle, Mark J. Prizant, and Steven Treadwell	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) The Charles Stark Draper Laboratory, Inc. Cambridge, MA 02139	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration Langley Research Center Hampton, VA 23681-0001	10. SPONSORING / MONITORING AGENCY REPORT NUMBER NASA CR-194924
11. SUPPLEMENTARY NOTES Langley Technical Monitor: Carl R. Elks Final Report	
12a. DISTRIBUTION / AVAILABILITY STATEMENT Unclassified-Unlimited Subject Category 62	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The Army Avionics Research and Development Activity (AVRADA) is pursuing programs that would enable effective and efficient management of large amounts of situational data that occurs during tactical rotorcraft missions. The "Computer Aided Low Altitude Night Helicopter Flight Program" has identified automated Terrain Following/Terrain Avoidance, Nap of the Earth (TF/TA, NOE) operation as key enabling technology for advanced tactical rotor craft to enhance mission survivability and mission effectiveness. The processing of critical information at low altitudes with short reaction times is life-critical and mission critical necessitating an ultra-reliable/high throughput computing platform for dependable service for flight control, fusion of sensor data, route planning, near-field/far-field navigation, and obstacle avoidance operations.

To address these needs the Army Fault Tolerant Architecture (AFTA) is being designed and developed. This computer system is based upon the Fault Tolerant Parallel Processor (FTPP) developed by Charles Stark Draper Labs (CSDL). AFTA is hard real-time, Byzantine, fault-tolerant parallel processor which is programmed in the ADA language.

This document describes the results of the Detailed Design (Phase II and III of a 3-year project) of the AFTA development. This document contains detailed descriptions of the program objectives, the TF/TA NOE application requirements, Architecture, hardware design, operating systems design, systems performance measurements, and analytical models .

14. SUBJECT TERMS Fault-tolerant, real-time digital computer, Terrain Following/Terrain avoidance operation	15. NUMBER OF PAGES 126
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102