NASA Contractor Report 181759

Study of A Unified Hardware and Software Fault Tolerant Architecture

(NASA-CR-181759) STUDY OF A UNIFIED HARDWARE AND SOFTWARE FAULT-TOLERANT ARCHITECTURE Final Report (Draper (Charles Stark) Lab.) 67 p CSCL 09B

N89-14003

Unclas
G3/62 0183410

Jaynarayan Lala
Linda Alger
Steven Friend
Gregory Greeley
Stephen Sacco
Stuart Adams

THE CHARLES STARK DRAPER LABORATORY, INC.
CAMBRIDGE, MA 02139

Contract NAS1-18061
January 1989

NASA
National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

TABLE OF CONTENTS

Section | Page

LIST OF ILLUSTRATIONS v

1.0 INTRODUCTION 1

2.0 FAULT TOLERANT PROCESSOR-ATTACHED PROCESSOR ARCHITECTURE 5
- 2.1 Design Diversity Based Architectures 5
- 2.2 CSDL FTP/AP Architecture 7
- 2.3 Architecture Solutions to N-Version Programming 11

3.0 MARKOV MODEL FOR FTP/AP HARDWARE AND SOFTWARE FAILURES . 17
- 3.1 Markov Model for a Quad Fault Tolerant Processor Random Hardware Failures 17
- 3.2 Markov Model for Software Failures in a Multi-Version System 21
  - 3.2.1 A Theory on Software Failures 21
  - 3.2.2 Modeling Multiple Programs 22
  - 3.2.3 A Markov Model for FTP/AP Reliability 24

4.0 RESULT RESOLUTION 29
- 4.1 FTP/AP Precision Voter 29
  - 4.1.1 Difference Function 30
  - 4.1.2 Ordering and Placement 32
  - 4.1.3 Hardware Implementation 34
- 4.2 Hardware/Software Isolation 34
  - 4.2.1 Isolation Algorithm 35
  - 4.2.2 Software Faults 36

5.0 THE DECISION ALGORITHM 37
- 5.1 The Confidence Voter 39
- 5.2 The N-Version Simulator 39
- 5.3 The Simulation Results 40

6.0 EXPERIMENTS 43
- 6.1 System Operation 43
- 6.2 Verification of FTP/AP FTSW Fault Isolation Algorithm 45
  - 6.2.1 Objective 45
  - 6.2.2 Experiment Operation 45
  - 6.2.3 Summary of Results 48
  - 6.2.4 Conclusions 54

6.3 Fault Injection Experiment 54
6.3.1 Objective 54
6.3.2 Description 54
6.3.3 Results 55
6.4 Performance Measurements 59
6.4.1 Objective 59
6.4.2 Description 59
6.4.3 Summary of Results 61
6.4.4 Conclusions 62

7.0 CONCLUSIONS 63

8.0 REFERENCES 65

1.0 INTRODUCTION

During the last two decades, a great deal of progress has been made in the development of fault tolerance techniques that improve the ability of computer systems to cope with hardware component failures. Hardware fault tolerance is now a relatively mature technology with the development of such computers as the FTMP, SIFT, FTP [1, 2, 3, 4] and several other fault tolerant flight control computers. By comparison, designers of mission and life critical systems, with few exceptions, still strive to develop perfect or error-free software, with fault avoidance rather than fault tolerance the prevalent approach. Error-free software, if it is possible to produce it at all, is a necessary but not a sufficient condition for many critical applications. It is also necessary to formally certify the correctness and error-free nature of the system. The state-of-the-art in validating and verifying software and hardware cannot guarantee that the operation of a system of any practical size will be error-free, or even that a probabilistic bound will be attained.

Historically, 50% of the software failures that occur in avionics systems take place during the operational phase of the system's life cycle [5]. One has to assume that, given the increasing complexity of future aerospace applications, these systems will, in fact, contain latent software faults and that software fault tolerance will be necessary in order to prevent system failure for these missions. Software entities are even more complex than the computer hardware and possess orders-of-magnitude more states. Additionally, it is our contention that for real-time flight crucial applications in aerospace vehicles there are several unique requirements and conditions that contribute to increased software complexity. These include management of hardware redundancy, real-time response constraints and almost always a new and unique computer hardware architecture and a new and unique operating system software for each new application. All of these factors combine to dilute and weaken the traditional software validation and verification techniques. As a result, the probability of achieving error-free software for flight crucial applications is highly diminished. The only solution, for flight crucial applications and other systems where human safety is at stake, is to detect software errors during system operation and take corrective measures in real time so that execution of critical functions is not interrupted.

Avizienis proposed N-version software [6] as a technique to detect and recover from software errors as a more specific form of redundant programming put forward earlier by Elmendorf in 1972 [7]. However, several practical problems have been encountered in the implementation of the N-version software. One problem, observed fairly widely, is the prevalence of correlated errors among independently coded versions [8, 9]. A second major problem, in our opinion, is the manner in which N-version software has been implemented in the hardware architecture. It is our contention that these implementations have actually reduced the overall system reliability by weakening the architecture's ability to withstand hardware malfunctions. One of the goals of this project was to produce a unified architectural approach that extends a well known hardware fault tolerant concept to provide for software fault tolerance without violating the fundamental hardware fault tolerance design principles and provides a possible solution to the problem of correlated software errors.

The real time applications of digital computers where human safety is at stake require ultrahigh system reliability. This includes an ability to tolerate hardware failures as well as software errors. The research and development of the past two decades has established a firm theoretical foundation for hardware fault tolerance. The problem is how to extend these proven hardware fault tolerant architectures to tolerate software errors and to continue to perform the intended application function correctly in the presence of hardware and/or software malfunctions. The extensions to the architecture should be such that the proven aspects of the architecture that are crucial for high coverage of hardware faults are not compromised. The fundamental theoretical requirements for hardware fault tolerance should not be violated by any architectural features that are added for software fault tolerance. The added overheads of software fault tolerance should not be so high as to make the architecture unsuitable for practical applications. The architecture should not compromise the performance to the point of making the system unfit for real time applications with time critical response requirements. After the overheads of hardware and software fault tolerance are taken into account enough throughput should be available to perform the application computation in real time with adequate response time to external events and sufficiently small transport lag between inputs and outputs.

Finally, for most advanced applications, it will not be sufficient simply to detect software errors and initiate a fail-safe shutdown of the computer system. For a digital fly-by-wire flight control application, for example, shutting down the computer is tantamount to a loss of the vehicle if there is no backup. Similarly, for totally autonomous vehicles, highly automated manned vehicles, or remotely operated sites, it may not be possible to provide a suitable backup for the computer system. In these applications it is necessary to identify the source of the failure (hardware module or software entity), isolate it from the rest of the system, and provide a recovery mechanism so that the application function may continue to be executed correctly and with no interruption. Advanced applications, therefore, will require the computer systems to be fail-operational rather than just fail-safe.

The proposed Fault Tolerant Processor/Attached Processor architecture has been developed by CSDL in order to unify the treatment of hardware failures and software errors and at the same time meet the NASA requirements for real time aerospace applications. This architecture concept differs fundamentally from most other approaches for unifying hardware and software fault tolerance. There were four major goals of this project: (1) to develop this architecture to solve the problems of implementing N-version software, such that hardware fault tolerance or performance is not compromised, (2) to solve the problems of correlated errors among independently coded versions, (3) to isolate the source of the failure between hardware and software, and (4) to provide a recovery mechanism. Other goals of the study included the investigation of the suitability of the FTP/AP architecture for

N-version software using an actual application, providing a testbed for future fault tolerant software experiments and relating the results to the Advanced Information Processing System (AIPS) and extending the results to other architectures.

A quadruply redundant core FTP together with the interfaces to four Attached Processors was designed and fabricated by Draper Laboratory and delivered to the NASA Langley Research Center AIRLAB as a proof-of-concept of the FTP-AP architecture and to be used for this study. It has been configured in the AIRLAB as shown in Figure 2. Four VAX-11/750 computers in the AIRLAB are being utilized to emulate the Attached Processors. The Attached Processors execute four versions of the yaw damper, a critical part of a commercial transport aircraft's flight control system. Two of the versions are coded in Fortran, one in C, and one has been coded in Ada by an Computer Aided Software Engineering (CASE) tool. A fifth VAX, called the Host VAX, simulates the aircraft dynamics, provides simulated sensor inputs to the FTP, and accepts actuator commands from the FTP. The FTP passes the sensor information to the attached VAX computers, schedules the yaw damper control law, votes on their actuator commands and passes the voted actuator command back to the aircraft simulation in the Host VAX.

Section 2 is a discussion of the Fault Tolerant Processor/Attached Processor Architecture. Section 3 discusses the reliability, maintainability and availability (RMA) analysis of the quad CSDL FTP/AP hardware and a new method of modeling multiple software failures that is the result of investigating the underlying failure mechanism, not just the symptoms of the failures. Section 4 is a discussion of the hardware/software isolation algorithms and Section 5 explains the decision algorithm used in the system to solve the problem of correlated software errors and the results of simulations run with the algorithm. Several experiments were conducted to validate the isolation algorithm and its implementation and to measure the real time performance of the system. Section 6 is a discussion of those experiments. Finally, Section 7 concludes with a summary of results and suggestions for future work with the testbed.

2.0 FAULT TOLERANT PROCESSOR - ATTACHED PROCESSOR ARCHITECTURE

This section presents a detailed description of the CSDL FTP/AP architecture which is a unified approach to hardware and software fault tolerance. Subsection 2.1 is a description of other architectures that have been designed in the past in order to unify hardware and software fault tolerance. The approach of these architectures is fundamentally different from the CSDL approach. Subsection 2.2 is a detailed description of the CSDL FTP/AP architecture and Subsection 2.3 is a discussion of the FTP/AP solutions to many of the problems with the N-version programming technique.

2.1 Present Design Diversity Based Architectures

The architectures that have have been proposed or employed in actual applications to date do not meet all of the conditions discussed in Section 1 that are necessary for advanced applications requiring ultrahigh system reliability. These include operational systems such as the SP-300 digital autopilot/flight director (category IIIa) for the Boeing 737-300 commercial transport aircraft [10], the slat and flap control system for the Airbus A310 aircraft [11, 12] and the newer systems about to enter service such as the digital fly-by-wire flight control system for the Airbus A320.

Generally, these designs are all based on the principles of design diversity. Typically, the hardware as well as the software in redundant channels is dissimilar in specification, design, and execution. The microprocessors in redundant channels have different Instruction Set Architectures (ISAs), are manufactured by different vendors, and execute software that has been designed, developed, and tested by independent teams.

The rationale behind these architectures is the principle of design diversity. By using independent designs, it is argued, the redundant channels would not exhibit common mode failures. A microcode error in an instruction of a microprocessor, for example, would result in the failure of only one channel while the other channel(s) would continue to operate correctly since they do not use the same microprocessor design. Similarly, an error in software in one channel, whether due to incorrect coding or a misinterpretation of specifications, would result in an incorrect output only from that channel while other redundant channel(s) continue to produce correct output. These arguments have been successfully used in certifying the aforementioned systems and others for safety critical applications where required probability of certain system failure modes such as an uncommanded surface movement is 10^{-9} per flight hour or less [10, 11].

However, these architectures do not address all the issues and requirements outlined earlier in Section 1. In particular, they have three significant shortcomings. They are designed to be fail-safe and can not be extended to fail-operational applications. (One exception is the A320 system which is designed to be fail-op, fail-op.) They provide protection against common mode failures at the expense of weakening the protection against random hardware component failures. Finally, even the protection against common mode failures is of a questionable degree since design diversity can not guarantee a total elimination of *coincident errors* [8]. (Coincident errors are defined as errors manifested by redundant channels in the form of incorrect outputs when excited by the same input whether or not the erroneous outputs are exactly the same.)

The primary reason for the shortcomings of these architectures is the approach to redundancy management. The redundant channel outputs are compared to detect presence of a hardware fault or a software error. If the outputs disagree by more than a certain threshold, a shutdown of the computer, which is typically dual redundant, is initiated. If another pair of channels is available, such as in dual-dual systems, control is transferred to the operational pair. Sometimes the output disagreement between two computers must exceed the threshold for a certain *time interval* before a failure is declared. Additionally, the disagreement threshold may depend on flight conditions and on the specific output variables being compared, among other things. When such a failure is indicated, the cause or the location of the failure is quite uncertain. It could be hardware or software and it could be in either channel. In fact, under some conditions the threshold may be exceeded quite legitimately because of sensor skew in redundant channels, thus causing a false alarm [13].

Without further isolation procedures, the only option in many systems is to permanently disable a pair of computers and their associated software which typically means losing two dissimilar hardware channels and two versions of applications code because one of these four entities failed. Furthermore, such a significant loss of resources may actually be caused by a false alarm or a temporary event such as a single event upset, a hardware transient or the passage of the applications code through an error sensitive input space.

In order to manage the redundancy intelligently, then, it is absolutely crucial that the cause of the disagreement be isolated to not only one of the four entities (two hardware channels and two software versions) but also whether or not the failure is transient, intermittent, or permanent. The fault isolation procedure typically invoked in the architectures under consideration is the execution of predetermined self tests. The coverage of self tests in uncovering hardware faults is notoriously low. In any case, they are quite useless in resolving the hardware-software isolation problem unless the hardware happens to fail permanently. A majority of the hardware failures, from 60 to 80 percent, are known to be not permanent [29]. Given the low self test coverage and the propensity of hardware for transients, the likelihood of isolating the cause of the failure correctly in real time in such systems is orders of magnitude lower than would be required to meet the 10^{-9} failure per hour criterion after system reconfiguration. The only certain means of providing the fail-operational capability with the requisite reliability, then, is to discard the redundant hardware pair, and its associated two software versions, at the slightest hint of trouble. Unfortunately, such a strategy could result in a quick loss of resources, depending on the transient failure rates. For example, in a dual-dual system two transients, one in each pair, could cause a total system shutdown.

The same hardware and software resources as used in these architectures can be configured and managed in a different way to yield ultrahigh reliable systems that meet the real time application requirements outlined earlier.

2.2 CSDL FTP/AP Architecture

The proposed architecture consists of a Byzantine resilient hard core Fault Tolerant Processor (FTP) which meets the fundamental theoretical requirements for fault tolerance [3,14]. To each redundant channel of the FTP is attached an application processor (AP) which is nonredundant. The core FTP may be duplex, triplex, or quadruply redundant depending on the reliability requirements and fail-safe, fail-operational or 2-fail-operational considerations of the application. A minimum of at least a triplex configuration is recommended for reasons to be discussed later. The quadruplex configuration of the FTP-AP architecture that has been used in this study is shown in Figures 1 and 2. The core FTP is responsible for I/O management, assignment of application functions to attached processors, and hardware and software redundancy management. The attached processors are responsible for executing the applications software. This architecture allows for design diversity in hardware and software and at the same time meets the real time applications requirements discussed in Section 1. This is achieved by the unique approach to redundancy management.

As described in greater detail in [3,14], the core Fault Tolerant Processor architecture is designed such that the outputs of all redundant channels are always in bit-for-bit exact agreement when there are no faults and the outputs of all correctly operating channels are in exact agreement in the presence of a predefined number of arbitrarily malicious or Byzantine faults. The theoretical requirements necessary to guarantee this behavior can be summarized as follows. In order to tolerate f simultaneous Byzantine faults, it is necessary to have 3f+1 independent fault containment regions which are connected by 2f+1 disjoint paths and which go through f+1 rounds of information exchange to arrive at exact consensus [15, 16, 17]. Furthermore, if the skew between redundant channels is bounded, then a bit-for-bit comparison and voting of channel outputs, after waiting for maximum time skew, indicates unambiguously if there is a failure as well as the source of the failure. (This is the reason for recommending at least a triplex level of redundancy for the FTP which can tolerate one Byzantine fault and still continue to operate correctly). The core FTP redundant channels execute identical software on identical hardware. The core FTP software consists of a hardened operating system kernel, redundancy management, and input/output management software [18]. Thus, it is very small in size and application independent and need not change with every new application All the traditional validation and verification techniques can be applied to this kernel. The kernel software may also be small enough to be formally verified when mature formal software verification techniques became available. The same is true of the core FTP hardware. Since the requirements of the functions being executed by the core FTP are not complex, a simple RISC microprocessor that has been formally verified [19] can be utilized to minimize the risk of a common mode processor failure. In the unlikely event of a common mode failure in the core FTP, either hardware or software kernel, several detection and recovery mechanisms are provided [18]. These include hardware watchdog timers, software timeouts, hardware and software exception handlers, total system restart, etc.

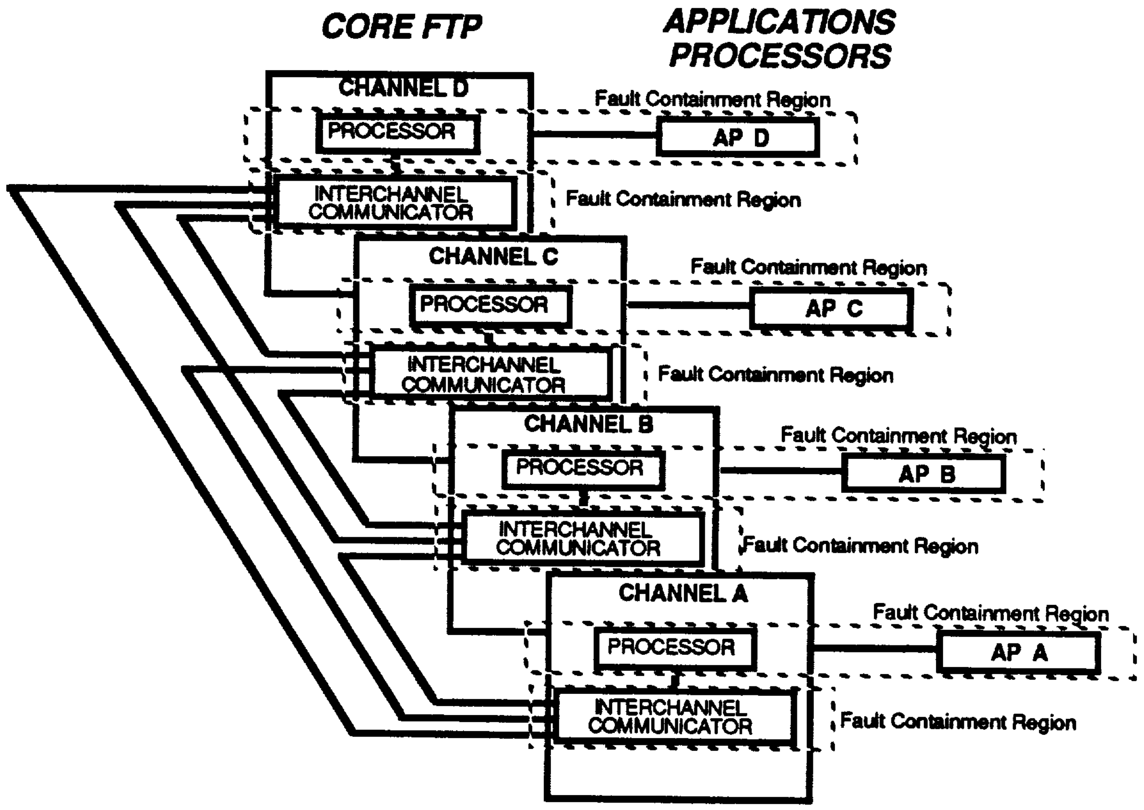


Figure 1. FTP/AP Hardware Architecture

Since the redundant channels of the core FTP execute identical software, on identical hardware, with identical inputs, any disagreement of a single channel with the majority is an indication of a random hardware failure. This failure, of course, needs to be further categorized as permanent, intermittent, or transient such as one that might be caused by a single event upset (SEU). Restarting the failed channel after restoring its internal state from other operating channels indicates whether or not the fault, or at least its manifestation in the form of an erroneous output, is permanent. If the channel can be restarted successfully, it is brought back in operation. However, it is assigned a demerit in its dynamic health variable. This variable is used to differentiate between transient and intermittent failures. There is also a background memory scrub program that routinely compares memory contents of redundant channels, thereby exposing any bit flips caused by single event upsets. If an SEU causes a more insidious problem than a bit flip such as a change in processor control flow, then that failure would show up as a transient since the affected processor would disagree with others until it is reinitialized and restarted.

The core FTP architecture has served well in numerous applications [4] by demonstrating an extremely high coverage of random hardware faults. By attaching applications processors, it has been extended to provide a unified hardware-software fault tolerant architecture. Each application processor communicates with only one channel of the core FTP. However, all the redundant channels of the core FTP congruently decide the workload for each of the applications processors. This extension of the architecture is used to provide design diversity and protection against common mode failures. It is the applications area which is new and different for each new application and therefore most likely to contain design errors. The attached processor hardware and software redundancy is managed by the core FTP as follows.

The configuration used for the study and experiments was one in which the core FTP is quadruply redundant and each FTP channel has an attached processor, although such a rich hardware configuration is not necessary for all applications. In this configuration, each AP is assigned to execute a different version of the application code such as the flight control program for a commercial transport. At the beginning of each iteration of the control law, the FTP reads all the sensors and distributes an identical copy to each AP. At the end of each iteration, the results from each AP are obtained by the FTP and internally distributed to all redundant channels so that each FTP channel has all four results. The four results are then compared to determine if there is a disagreement. Because of dissimilar software, these results are not expected to be bit-for-bit identical. However, it should be emphasized here that the approximate equality in this case is solely due to differences in the mechanization of the control law specification and not due to sensor skew. (Furthermore, the AP's have virtually no effect on the detection and isolation of internal FTP failures because the outputs of APs are not used to detect core FTP failures.) If one of the AP results disagrees with the majority by more than the accuracy described in their common specification, an isolation procedure is invoked to determine if the error was due to hardware or software. The details of the hardware/software isolation algorithm are described in Section 4. An outline of the isolation algorithm is as follows.

The iteration of the software version in question is rerun on all APs using inputs for the previous iteration and after restoring the internal status of the software version to its previous state. A bit-for-bit majority vote of the three APs, which are not suspects, provides the suspect software version's output for that iteration with hardware faults masked. Any difference in this voted response and that of the suspect AP is attributed to a hardware fault in the suspect AP. The algorithm compares the original result from the suspect AP with the voted result of the non suspect APs and with the result of the isolation iteration of the suspect AP. The algorithm detects both transient and hard faults in the suspect AP. It also determines if the fault was caused by a software error (or a common mode hardware fault which is treated as a software error). The algorithm identifies a hard hardware fault if the results of both iterations in the suspect AP disagree with the voted result from the non suspect APs. The algorithm identifies the fault as a transient hardware fault, if the original iteration of the suspect AP disagrees with the voted result from the non suspect APs, but the result from the isolation iteration of the suspect AP is the same as that of the voted result from the non suspect APs. The fault is identified as due to the software version, if the original result from the suspect AP agrees with the voted result from the non suspect APs, but does not agree (within the specified accuracy) with the original results from the other software versions. The algorithm also identifies that there is both a hardware fault in the AP and a software error in the version, if the result of the suspect AP disagrees with the voted result of the non suspect APs and the voted result of the APs, running the suspect version, disagrees (within the specified accuracy) with the results of the original iteration for the other software versions.

If the malfunction was caused by the software version executing in the AP, the *confidence voter* is invoked. The confidence voter keeps a history of error occurrences in all software versions. It uses this information to resolve those cases where multiple software versions malfunction simultaneously, causing coincidence errors. As mentioned earlier, coincidence errors have been found to be one of the stumbling blocks in improving software reliability through N-version coding. The confidence voter principles of operation and some simulation results indicating its effectiveness in tolerating coincidence errors are described in greater detail in Section 5.

Three important comparisons of the FTP-AP architecture with other design diversity based architectures are in order. The hardware content of the quad redundant FTP-AP is of the order of eight processors with some interprocessor communication hardware which is just a few custom designed chips. This architecture provides a full fail-op, fail-safe capability. The FTP-AP hardware is thus comparable in size and complexity to other architectures that provide similar capability [10, 30].

A major difference between the FTP-AP and other design-diversity-based architectures is the degree of consensus between outputs of redundant channels. Many design-diversity-based architectures are designed as asynchronous systems. The asynchronous approach involves execution of the versions of the application without imposing any synchronization constraints on the redundant channels. The fast lag-time requirements typical of control applications necessitate that the asynchronous versions must use the latest sensor values available. Thus, the different versions are often executing with significant differences in their input values used. Occasionally this can even extend to versions executing under widely differing assumptions of mission phases or under differing application dynamics. To counteract these effects, the thresholds used to detect and isolate faults must be set very wide.

By contrast, the dissimilar versions on the FTP-AP architecture execute in parallel using bit-for-bit identical inputs guaranteed by source congruency. The outputs of the versions executing on the APs are not expected to be bit-for-bit identical, and a fault is signalled only when outputs disagree by more than the accuracy required by their common specification. However, since the disagreement threshold does not have to allow for sensor skews or differing mission phases or application dynamics between versions, the FTP-AP provides more sensitive fault detection than would be possible in an asynchronous system. Furthermore, once a fault has been detected, the isolation algorithm does require a bit-for-bit consensus between AP outputs as explained in the Fault Isolation section of Chapter 4. The core FTP and possibly some applications on the APs are not executing dissimilar versions, thus exact consensus is required between outputs of redundant channels, providing very sensitive fault detection and identification.

2.3 Architecture Solutions To N-Version Programming

Past work in fault tolerant software has uncovered several issues and problems with the N-version programming technique. Among the issues noted by Avizienis [21] were the assurance of input consistency, interversion communication, protection of the support environment, version synchronization, meeting real time constraints, recovery of failed versions and the decision algorithm.

The FTP/AP architecture guarantees input consistency with the core FTP data exchange hardware. Inputs such as sensor values that need to be distributed to all versions are obtained by the core FTP. Assume, for example, that Channel A of the FTP is connected to a sensor. After reading the sensor, Channel A broadcasts the value to the other three channels of the FTP using the Byzantine-resilient, two-phase exchange illustrated in Figures 3a and 3b. At the end of this exchange, each operating channel in the FTP has an identical value of the sensor even in the presence of a Byzantine Fault [14]. The four channels then distribute this value to their respective attached processors.

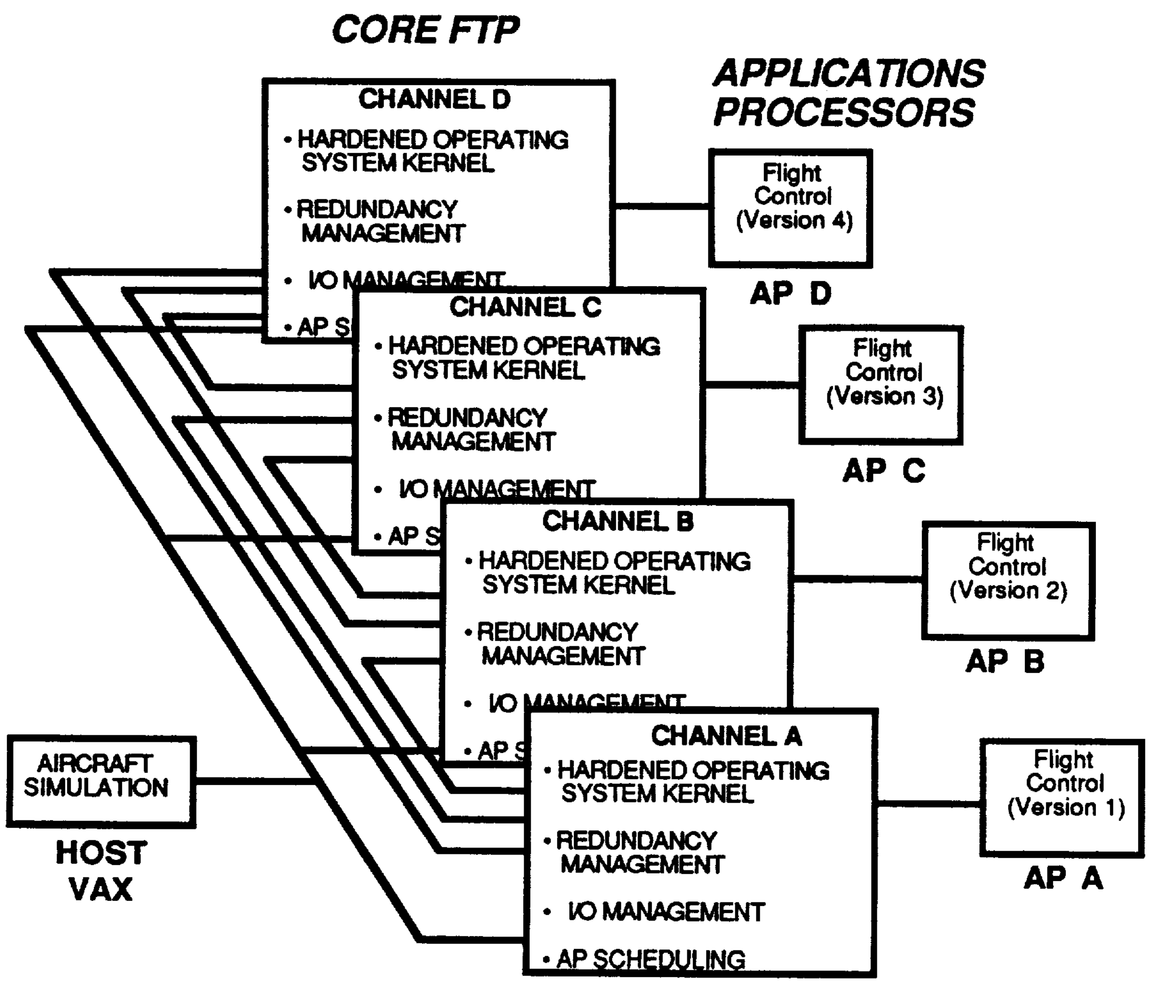


Figure 2. FTP/AP Software Architecture

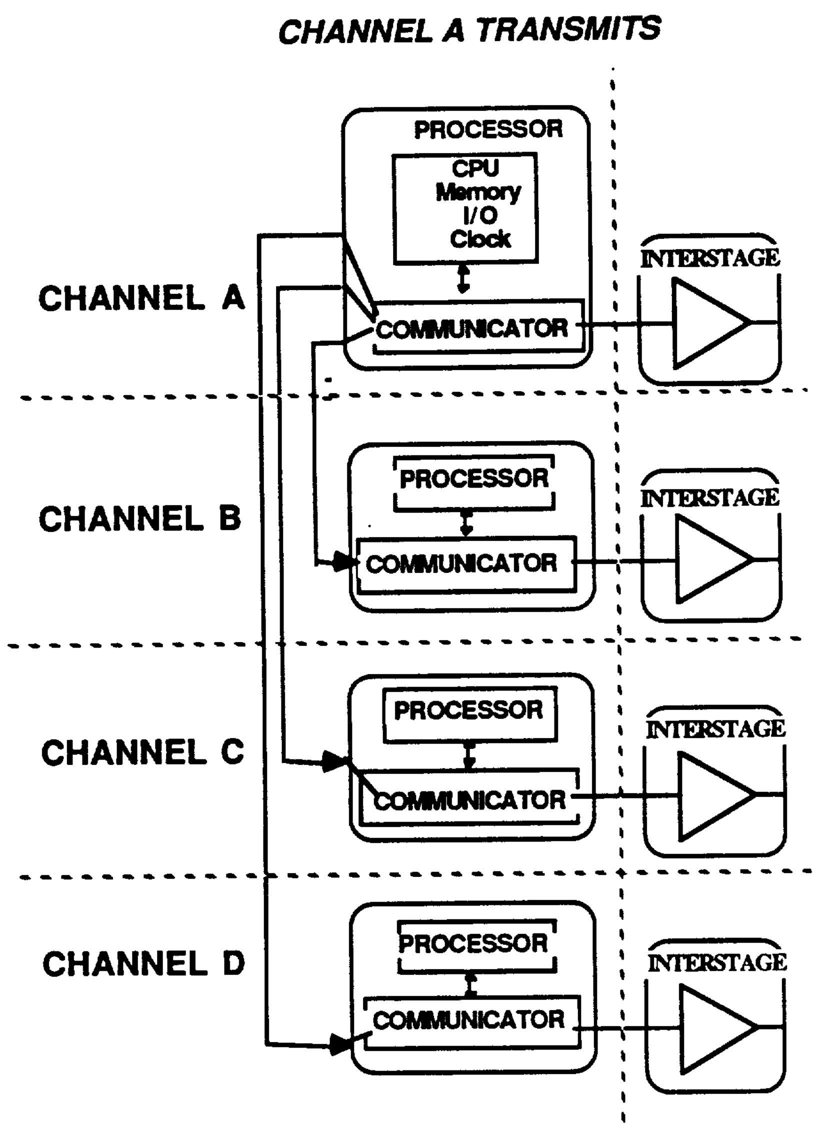


Figure 3a. Byzantine Resilient Data Distribution From Channel A: Phase I-Channel A to Interstages

ALL CHANNELS RECEIVE

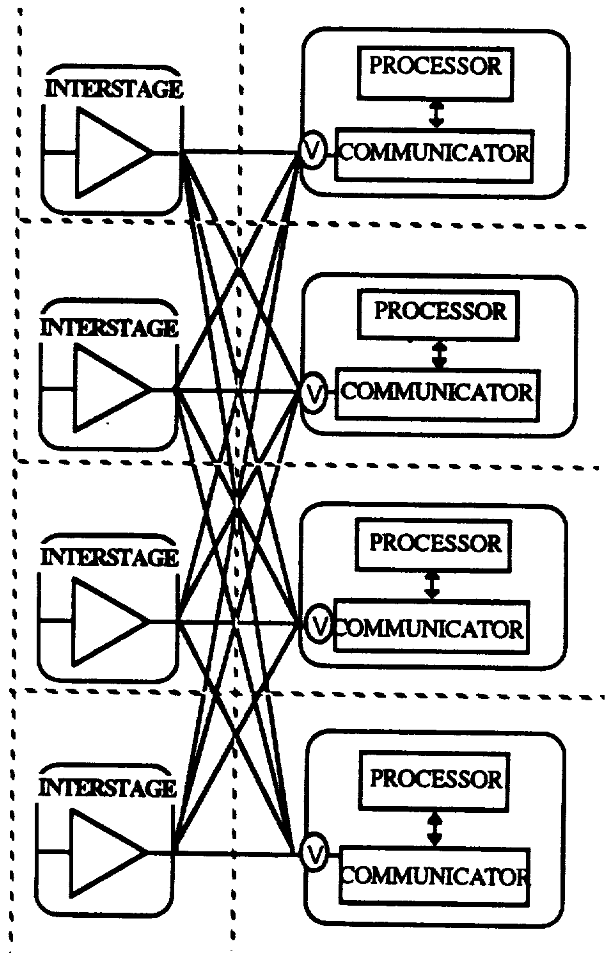


Figure 3b. Byzantine Resilient Data Distribution From Channel A: Phase II-Interstages to all Channels

Interversion communication in the traditional sense is not necessary in this system. Each version runs with the set of sensor inputs and its internal state variables which are saved by the FTP after each iteration. The code for all versions is resident on all APs but only one version is executing on each AP. The inputs and internal state variables are given to each version at the beginning of each iteration. This is necessary so that the FTP can distribute versions among the APs for hardware/software isolation or other reasons. Communication between the FTP and the versions is supported by the FTP/AP hardware by means of a shared memory between each core FTP channel and its attached processor and the FTP data exchange. All inputs to the APs are transmitted by the FTP via shared memory. Outputs from the APs are received by their respective FTP channels and distributed internally in the core FTP by means of the data exchange mechanism. The APs are the slaves of the FTP such that the FTP controls all interversion communication.

Protection of the support environment is also one of the functions of the FTP/AP architecture. Since each version is running on a physically separate processor that forms a distinct fault containment region, the rest of the system will continue to operate even if one of the APs experiences a hardware fault. Also, if a software error causes a version to abort and destroy the operating system on one AP, the error will not propagate beyond that AP and not affect the continued operation of the rest of the system. Several prior N-version software experiments that used a single processor to execute multiple versions sequentially found that failure of one version could cause failures of succeeding versions as well [9, 22].

The problem of version synchronization is solved by the FTP scheduling each iteration of the four software versions. The FTP then waits a predetermined maximum time and reads the outputs from the four versions which are running in parallel. If any of the versions has not completed at the end of this period, it is declared failed and its output masked out of the vote.

As a program travels through its input space, it will also travel through regions where the input will cause the program to fail (see Figure 5). During the period that the program is in this failure region, it will produce erroneous results. When it leaves this region, the hope is that it will produce correct results if reinitialized to a correct state. Therefore, when a version fails, it is possible to attempt recovery of the failed version. The recovery method in the FTP/AP involves restoring the failed version to its initial state followed by the continued execution of the failed version, with its output masked, but comparing its output to the voted output. Reinitialization of a failed software version to a state that is congruent with other versions is a difficult task [23]. Instead, the failed version is initialized to a cold start state and allowed to bring itself to a congruent state over time by open loop operation. The version is restored if its output agrees with the voted output for several iterations.

3.0 MARKOV MODEL for FTP/AP HARDWARE and SOFTWARE FAILURES

Markov models for both hardware and software failures of the quad Fault Tolerant Processor/Attached Processors running four versions have been developed. Developing the hardware model followed known techniques since hardware failure models assume that failures are random, statistically independent events with a constant hazard rate. Using similar models for software failures was not feasible since software failure mechanisms are very different from hardware failures. Software failures are deterministic functions of the program's input and state, and it has been shown that the failures of multiple versions of a program are not statistically independent [24]. In creating a Markov model for software failures of several versions running concurrently, a new technique was developed. Section 3.1 is the result of the hardware failure analysis and Section 3.2 discusses a new technique for modeling software failures.

Future work should include the development of a combined hardware/software model for a complete reliability analysis of the unified hardware and software FTP/AP architecture. Hardware failure rates for a given amount of functionality continue to decrease as the microelectronics technology advances due to increased integration levels and reduced number connectors [26, 31]. The technology needed for a specific application must be determined in order to derive the specific failure rates for the hardware model. For the software model no data on failure rates of software is presently available.

3.1 Markov Model for a Quad Fault Tolerant Processor Random Hardware Failures

A Markov model has been developed for the FTP both with and without repairs. The state transitions of the model are based upon four rates: The rate of processors channel failures, the rate of inter-stage failures, the repair rate for a processor channel or inter-stage and the FTP fault recovery rate. The Markov model for the FTP with repairs is depicted in Figure 4. The model for the FTP without repairs is identical to the depicted model except that repair transitions are removed. Status S12 to S16 in the model are termed fail-safe states for those applications where it is possible to downmode to such a state. Otherwise, these five states would be combined with S17, the catastrophic failure state.

PRECEDING PAGE BLANK NOT FILMED

PAGE 16 INTENTIONALLY BLANK

FTP Markov States

S1	No failures
S2	1 interstage has failed - System recovering
S3	1 interstage has failed - System recovered
S4	1 processor has failed - System recovering
S5	1 processor has failed - System recovered
S6	2 inter-stages have failed - System recovering
S7	2 inter-stages have failed - System recovered
S8	2 processors have failed - System recovering
S9	2 processors have failed - System recovered
S10	1 inter-stage and 1 processor has failed - System recovering
S11	1 inter-stage and 1 processor has failed - System recovered
S12	3 inter-stages have failed - System recovering
S13	2 inter-stages and 1 processor have failed - System recovering
S14	1 inter-stage and 2 processors have failed - System recovering
S15	3 processors have failed - System recovering
S16	System failed safe
S17	System catastrophic failure

FTP Markov State Transitions

From State	To State	Reason
1	2	Inter-stage failure
1	4	Processor channel failure
2	3	Recovery from inter-stage failure
4	5	Recovery from processor failure
3	1	Repair of inter-stage
5	1	Repair of processor channel
2	17	Second failure while recovering
4	17	Second failure while recovering
3	6	Inter-stage failure
3	10	Processor channel failure
11	3	Repair of inter-stage
7	3	Repair of inter-stage
5	8	Processor channel failure
5	10	Inter-stage failure
9	5	Repair of processor channel
6	7	Recovery from inter-stage failure
8	9	Recovery from processor failure
10	11	Recovery from failure
6	17	Fault during recovery
8	17	Fault during recovery
10	17	Fault during recovery
7	12	Inter-stage failure
7	13	Processor channel failure
9	14	Inter-stage failure
9	15	Processor channel failure
11	13	Inter-stage failure
11	14	Processor channel failure
12	16	Recovered from failure - system failed safe
13	16	Recovered from failure - system failed safe
14	16	Recovered from failure - system failed safe
15	16	Recovered from failure - system failed safe
12	17	Fault during recovery
13	17	Fault during recovery
14	17	Fault during recovery
15	17	Fault during recovery

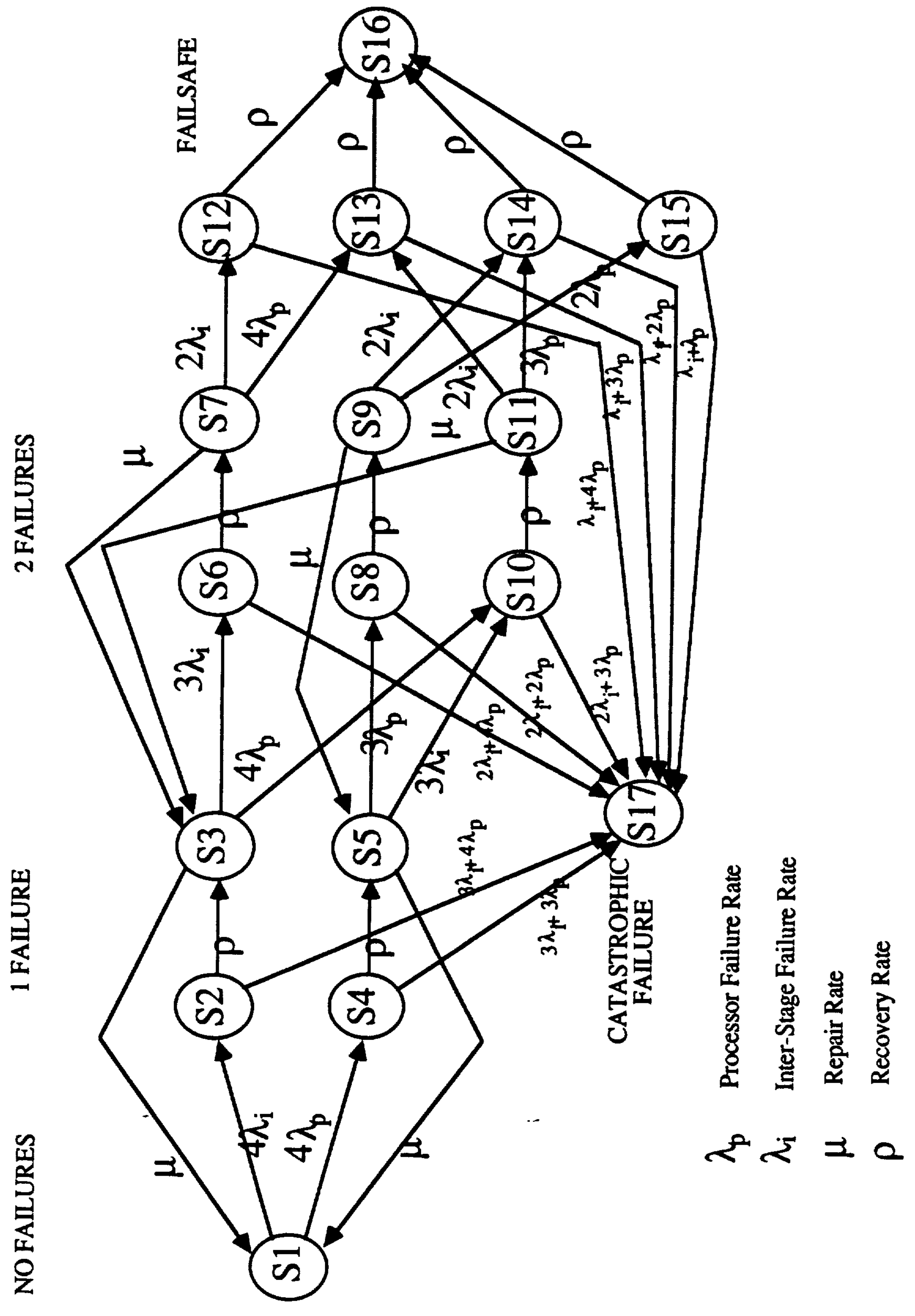


Figure 4. Markov Model of Quad Fault Tolerant Processor

3.2 Markov Model for Software Failures in a Multi-version System

The Markov model for multiple software failures developed at Draper represents a four-version system (such as the FTP/AP) that uses the majority (or "plurality") vote to resolve disagreements. The states in this model correspond to the types of errors the system could encounter. Thus, for example, one of the states is equivalent to the condition where two versions agree on one answer and the two other versions have two different answers. (This case is referred to as a 2:1:1 split.) As one can see, this model contains a large number of transitions rates ($\lambda_1, \lambda_2, \lambda_3, \lambda_4, \lambda_5, \lambda_6, \lambda_7, \rho_1$, and ρ_2). All of these rates, however, are given in terms of five parameters: λ_{fr}, λ_{ec}, ρ_{fr}, ρ_{ec}, and π. This section attempts to determine a relationship between these transition rates by examining the underlying failure process, rather than the visible errors, which are only symptoms of the faults within the code.

3.2.1 A Theory on Software Failures

In their paper on software failures [24], Eckhardt and Lee contend that a program has a distribution, θ (x), which is a function of the input space of the program. Thus, if a program has 3 inputs, then there exists a 3-dimensional input space, and each point in this input space is assigned a probability θ (x, y, z). This probability describes the "propensity of a population of programmers to introduce design errors such that" the software will fail on that input. Alternatively, one can think of the distribution as the probability that a randomly selected program will fail on the given input.

The point of this model is that software failure rates are explicitly linked to the input space of the program. (Current research tends to validate this model [25].) Eckhardt also notes that an N-version system with a pure majority vote[1] will be more reliable than its single-version counterpart only if the intensity distribution is less than 0.5 throughout the operational input space. In other words, if there is a region in the input space where the probability that an arbitrarily chosen version will fail is greater than 0.5, then the N-version system will be less reliable than the average single version.

G. Earle Migneault at NASA Langley Research Center introduced the idea of "dark crystals" (which Knight calls error crystals). If one looks at a program's n-dimensional input space, there would be certain regions where the input will cause the program to fail. These regions are the program's error crystals. According to this theory, which applies Eckhardt and Lee's model to an individual program, there is nothing random about software failures. If the input to a program is inside an error crystal, then the program will produce erroneous output. Otherwise, the code will produce the correct output. Recent

[1] In this case, "majority" is used in the strictest sense. A 4-versions system would require at least 3 versions to agree on an answer before that answer would be used as the system's output.

work by Knight [25] has mapped some two dimensional slices of error crystals in versions of his launch interceptor application.

3.2.2 Modeling Multiple Programs

To model the operation of a program, we can think of a point traveling some path through an n-dimensional input space. During its travel, the point will occasionally enter and leave these error crystals. Figure 5 shows a 2-dimensional slice of an input space. Within this space there are two crystals, A and B, that overlap. Because these regions share a large common boundary, an input following the path shown would experience a simultaneous double failure, then two sequential recoveries. Unfortunately, there are some complications with using this model.

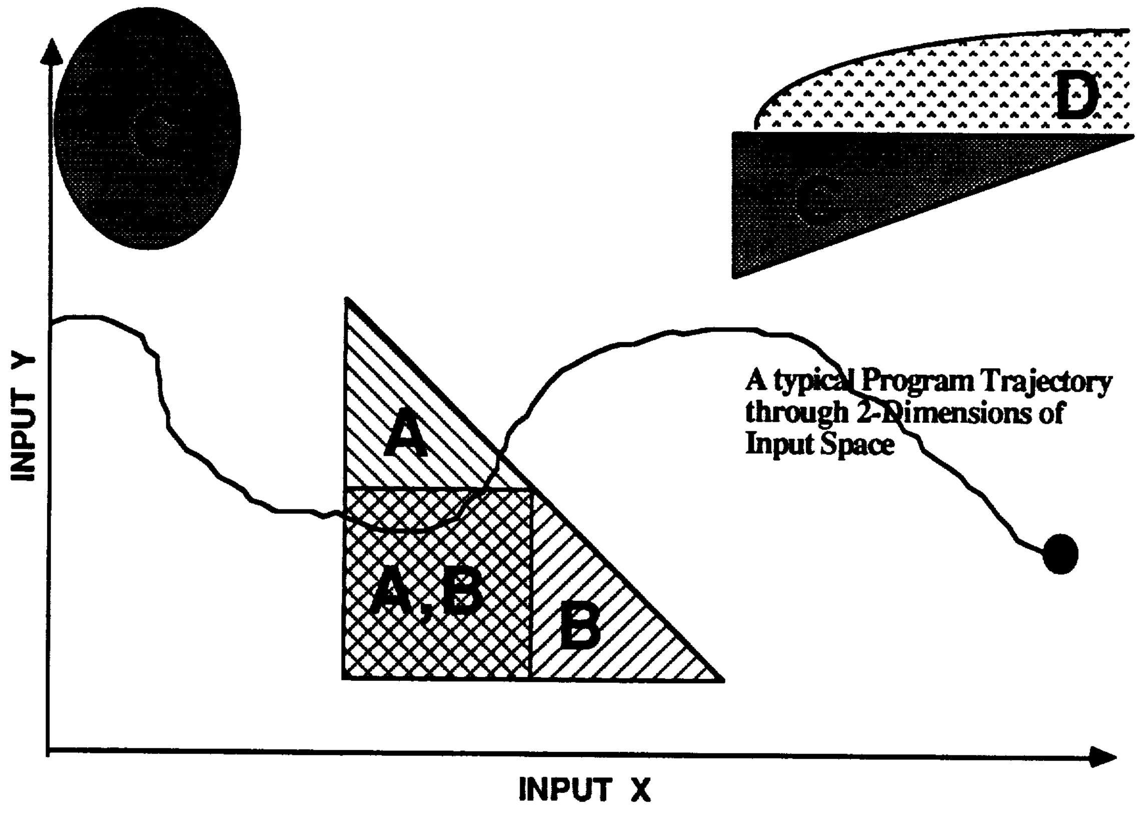

D
C
A
A,B
B
INPUT Y
A typical Program Trajectory through 2-Dimensions of Input Space
INPUT X

Figure 5. Failure Regions of Four Versions for Input Space (x,y)

The first complication is that one cannot assume versions will fail independently. Thus, if the probability of a single failure on a given iteration is a multiple of λ, the probability of two versions failing simultaneously is not necessarily a multiple of λ^2.

The second complication is that sequential failures of different versions are probably not independent. That is, when one version is in a failed state, the probability that the other versions will fail is likely to be higher. This view follows along with the theory put forth by Eckhardt and Lee: versions tend to fail together. Given this theory, we would like to take the more conservative view and say that the probability of a second version encountering an error crystal, while the input point is already inside the first version's error crystal, is higher than the probability of the first version encountering the original error crystal.

Using Eckhardt and Lee's model, one can easily understand, then, the concept of failure regions. The distribution of each version's error crystals may be uniform throughout the input space, but the distribution of the error crystals from four versions would definitely not be uniform. Thus, failure regions are areas of the input space where these error crystals from the four versions collect. These are the inputs which test or exercise special cases in programs, and it has been shown that programmers tend to make mistakes when they encounter special cases [27]. Because we believe that error crystals collect in certain areas, we will model the probability of failure for each version as increasing while any version is inside an error crystal.

By using this concept of failure regions, we are able to model the various transition rates with only a few parameters. First, however, we must make a critical assumption: the failures of different versions inside a failure region are independent. By making this assumption, we are saying that the dependencies among versions are due to the failure regions. In fact, one can think of a failure region as the area of an input where the specification for the program requires some type code which is more difficult to write. An example of this would be an *if* or *switch* statement. The probability that one programmer will write a faulty *if* statement is not affected by the other programmers' *if* statements. It is the fact that all the programmers have to write an *if* statement that increases the probability of each version failing on that input. Another example is a program that takes three points as its input. There may be a special case when all three points rest on the same line. Once the input is in a special case, which programs actually make the mistake can be treated as being independent; the dependency comes from the fact that all programs are in a special case simultaneously.

For our model, then, we use two parameters to describe the transition rates into and out of failure regions: λ_{fr} and ρ_{fr}. λ_{fr} is the probability that the input point will enter a failure region on a given iteration. ρ_{fr} is the probability that the input point, which is in a failure region, will exit that region. In an intuitive sense, λ_{fr} controls the density of failure regions within the input space while ρ_{fr} controls the average size of the regions.

After the input point has entered a failure region, we use the parameters λ_{ec} and ρ_{ec} to describe the transition rates into and out of error crystals. As with failure regions, λ_{ec} could be thought of as controlling the density of error crystals within the failure regions while ρ_{ec} controls the average size of each crystal. For continuous processes, such as an aircraft's heading, we would expect the input point to travel through error crystals in a linear fashion. For non-continuous processes, the input point will tend to "jump" through the input space. The net result is that ρ_{ec} reflects the average number of iterations the input point is expected to stay inside an error crystal.

In the model, these parameters are combined to form the transition rates. For example, the probability that the input point will enter at least one error crystal on its next iteration is

Probability(enter failure region) x
Probability(enter error crystal | inside failure region) x
number of versions

or

$$\lambda_{fr} \times \lambda_{ec} \times 4$$

Note that using this model, the rate at which failures occur (given that you are not in a failure region) is $4\lambda_{ec}\lambda_{fr}$, while the rate for double coincident failure is $6\lambda^2_{ec}\lambda_{fr}$. The values of $(4\lambda_{ec}\lambda_{fr})^2$ and $6\lambda^2_{ec}\lambda_{fr}$ are not necessarily equal, implying that the failures are not independent. Also, the rate of additional failures (given that there is already a single failure) is $3\lambda_{ec}$. The fact that the input point has entered the failure region has changed the probability that a single version will fail.

We also need to model the ratio of identical incorrect outputs versus non-identical incorrect outputs for multiple software failures. To do this, we assume that once a version has failed, it has a nearly infinite choice of different incorrect algorithms, each of which returns a unique value as its output. We also assume that there is one algorithm that is much more likely to be selected than any of the others. The probability of choosing this one "popular" incorrect algorithm, given that the version has failed, is π. The probability of choosing one of the nearly infinite number of other incorrect algorithms is $(1 - \pi)/n$ (where n is the number of other incorrect algorithms). If both versions choose the popular incorrect algorithm, then they will have identical incorrect answers. (This happens with probability π^2.) If either version (or both) choose any algorithm other than the popular incorrect algorithm, they will never agree on an incorrect answer. Each individual "unpopular" algorithm is considered so unlikely to be chosen that we assume that it will never be chosen more than once.

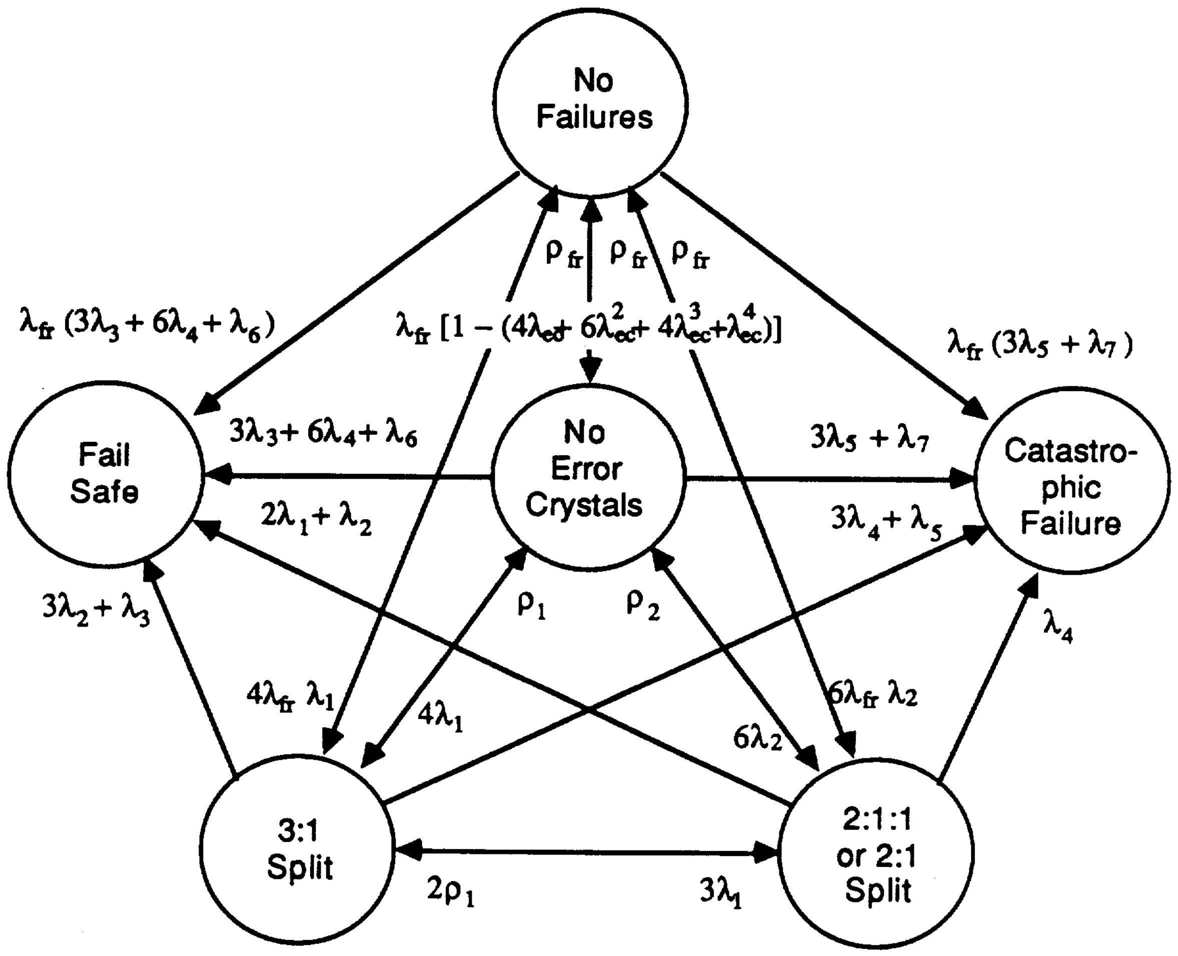
3.2.3 A Markov Model for FTP/AP Reliability

Using the information from Eckhardt and Lee's paper, we model the software failures of a 4-version system. As the four versions are running, we can imagine their input point wandering in the programs' input space. Evenly distributed throughout this space are failure regions. The input point enters these failure regions at a rate of λ_{fr} and, once in, exits at a rate of ρ_{fr}. Once the input point enters a failure region, error crystals are evenly distributed throughout the region. The input point enters these crystals at a rate of λ_{ec} and, once in, exits at a rate of ρ_{fr}.

Figure 6 shows the Markov model for software failures. All the transition rates for this model are based on the five parameters previously discussed, λ_{fr}, λ_{ec}, ρ_{fr}, ρ_{ec}, and π. Thus, λ_1, the failure rate for a single software module, is simply λ_{ec}. (Both rates assume that the input point is already inside a failure region.) The rate of dual coincident failures, is λ^2_{ec}. These failures, however, must be divided into identical failures, which occur at the rate λ_4, and unique failures, which occur at the rate λ_2. Of all the double failures, then, we assume that π^2 will be identical and the remaining $(1 - \pi^2)$ will be unique.

Triple failures occur at the rate λ^3_{ec}. These failure, however, must also be divided into those failures with unique outputs (λ_3) and those failures where two or more outputs agree (λ_5). The probability of all three outputs failing identically is π^3. The probability of any 2 (but only 2) outputs failing identically is $3\pi^2(1 - \pi)$. Thus, the probability that two or more outputs agree on an incorrect output is $3\pi^2 - 2\pi^3$. As a result, the probability the the three incorrect outputs are unique is simply $1 - (3\pi^2 - 2\pi^3)$.

Quadruple failures occur at the rate λ^4_{ec}. These failures are, again, divided into two types: those with all unique outputs (λ_6), and those with 2 or more identical outputs (λ_7). (Note, however, that this model does not include a 2:2 split quadruple failure as a possible outcome. This is because the identical incorrect answers are modeled by the one "popular" incorrect answer. A quadruple failure that has two pairs of incorrect answers implies that there are two "popular" incorrect answers.) The probability that all 4 answers are identical is π^4; any 3 (but only 3) is $4\pi^3(1 - \pi)$; any 2 (but only 2) is $6\pi^2(1 - \pi)^2$. Combining all these figures, the probability that two or more incorrect answers will agree is $6\pi^2 - 8\pi^3 + 3\pi^4$. The probability that all answers will be unique is just 1 minus that number.



Explanation of Symbols

λ_1	$= \lambda_{ec}$	Failure Rate for Software Modules
λ_2	$= \lambda_{ec}^2(1-\pi^2)$	Dual Coincident Error Rate (Unique Outputs)
λ_4	$= \lambda_{ec}^2\pi^2$	Dual Coincident Error Rate (2 Identical Outputs)
λ_3	$= \lambda_{ec}^3(1-3\pi^2+2\pi^3)$	Triple Coincident Error Rate (Unique Outputs)
λ_5	$= \lambda_{ec}^3(3\pi^2-2\pi^3)$	Triple Coincident Error Rate (2 or 3 Identical Outputs)
λ_6	$= \lambda_{ec}^4(1-6\pi^2+8\pi^3-3\pi^4)$	Quadruple Coincident Error Rate (1:1:1:1 or 2:2 splits)
λ_7	$= \lambda_{ec}^4(6\pi^2-8\pi^3+3\pi^4)$	Quadruple Coincident Error Rate (>2 Identical Outputs)
ρ_1	$= \rho_{ec}$	Recovery Rate for Software Modules
ρ_2	$= \rho_{ec}^2$	Dual Coincident Recovery Rate

Figure 6. Markov Model for Multiple Software Failures

These rates, λ_1 through λ_7, are used to create the transition rates between the states of the model. Of the 6 states, only the state labeled "No Failures" corresponds to the point being outside of a failure region. All transitions into this state, then, occur at the rate ρ_{fr}, and the sum of all the transition rates out of this state must equal λ_{fr}.

Once the input point is inside a failure region, there are 5 different states it could be in. Two of these are trapping states: the "Fail Safe" state occurs when there is no majority of versions that agree on a correct answer; the "Catastrophic Failure" state occurs when a majority of versions agree on an incorrect answer. The remaining 3 states (which are not trapping states) occur when a plurality of versions (2, 3, or 4) agree on a correct answer.

By investigating the underlying process that causes what we are trying to model, we are able to create a Markov model that reflects the software failure mechanism, which is significantly different from the hardware failure mechanism. While hardware failures are random, statistically independent events, software failures are not, and they should be modeled in a different fashion. This model treats software failures as deterministic functions of the software's input and it links failures between versions via common failure regions.

Another advantage to this model is that it uses a small number of unrelated parameters to define the transition rates between states. These parameters directly represent those quantities we are interested in: the size and density of failure regions, the size and density of error crystals, and the probability that two failed versions will choose an identical incorrect answer. This model also allows a wide range of parameters to be used. For example, we could represent the entire input space as a failure region by setting $\lambda_{fr} = 1$ and $\rho_{fr} = 0$. We could also treat all multiple failures as identical by setting $\pi = 1$. This flexibility, combined with the ability to directly control the important parameters are additional advantages of this model.

4.0 RESULT RESOLUTION

Execution of N-version fault tolerant software on the FTP/AP requires the resolution of a set of results to a single, correct value. Each result of the set is associated with a version of the specified function and the attached processor used to execute that version. The detection of incorrect results requires that the set of results be compared and voted. The correct result is defined to be the majority result and a disagreeing result is caused by either a fault in the associated version or a fault in the associated attached processor. (No attempt is made to isolate faults between the attached processor hardware and the attached processor executive. They are collectively termed the attached processor.) Masking of subsequent results from the suspect components until the fault has either been repaired or been judged transient is necessary to maintain the fault tolerance of the system. The compilation of component fault statistics and the minimization of resource loss due to masking requires the isolation of faults to either the attached processor or the version.

The comparison and voting of results from design diverse software necessitates the use of a precision voter. Correct results from the associated versions are only required to be equivalent within the accuracy described in their common specification; the bit for bit congruence of correct results is not guaranteed as it is in non-diverse fault tolerant implementations. The precision voter implemented on the FTP/AP fault tolerant software system is described in Section 4.1.

After evaluation by the precision voter, disagreements in the set of results returned from the execution of each version are the result of either faults within the versions or faults within the associated attached processor. Isolation of the fault to either origin requires the re-execution of the suspect version and evaluation of these isolation results based on the behavior of hardware and software faults. The hardware/software (or AP/version) isolation algorithm and the constraints placed on fault behavior are described in Section 4.2.

4.1 FTP/AP Precision Voter

Results from the versions associated with an implementation of design diverse software are only required to agree within the accuracy described in their common specification. A bit for bit congruence comparison is no longer a valid equivalence test of the results. Instead, some difference function which tests against the specified accuracy must be used. This difference delineates an equivalence interval which must be overlaid on the values represented by the set of results. The placement of this interval should be chosen to maximize the number of equivalent results, to minimize the number of pairwise result splits, and to minimize the amount of hardware/software fault isolation required. These issues will be discussed in the following sections.

4.1.1 Difference Function

The result returned by the yawdamp function of the installed autoland simulation is a floating point value with an accuracy specified as +/- 1E-3 of the allowed range. This corresponds to an absolute accuracy +/- 5E-2 degrees. The precision of the floating point representation used in the multiple versions of the yawdamp function is on the order of +/- 5E-7. A difference function which tests against twice the absolute accuracy, i.e. a difference of 1E-1 degrees, is necessary to determine the equivalence of these values.

The simplest implementation of the difference function would use the floating point representations directly. This would require either the expense of using a floating point co-processor or the overhead attendant in floating point emulation. An alternative is to convert the floating point representation into a fixed point representation and perform the comparison on the corresponding integer values. This removes the requirement of performing floating point operations and the comparison could possibly be implemented in hardware. In the FTP/AP system the difference function was implemented using a software comparison of fixed point values.

In the fixed point representation selected, the value represented is described by two integer quantities, the mantissa and the exponent, such that the value is equivalent to

$$\text{mantissa} * 2^{\text{exponent}}$$

If the mantissa was an infinitely precise real value then the above representation would also have infinite precision, instead the mantissa is constrained to be an integer value and the precision of the representation is determined by the value of the exponent. In the conversion from floating point to fixed point the integer mantissa is the result of truncating the appropriately shifted fractional portion of the floating point representation. Relative to the values representable in the floating point format, the precision of the fixed point representation as defined above is

$$\begin{matrix} + < 2^{\text{exponent}} \\ - 0 \end{matrix}$$

(The interpretation of $<2^{\text{exponent}}$ is the maximum value which is still less than 2^{exponent}.)

The number line of representable values and their relationship to the accuracy of the values to be represented is illustrated in Figure 7 for both floating point and fixed point comparisons. For fixed point numbers the precision of representable values is defined to be the accuracy of the values to be represented. This results in the property that values which differ by less than their specified accuracy in floating point format can not be guaranteed to be bit for bit congruent even in the defined fixed point format, but they can be guaranteed to differ by no more than one unit. In the figure the floating point values a, b, and c are within a single accuracy interval. After their conversion to the fixed point values a', b', and c', the values are not bit for bit equivalent, though they are within one unit of each other.

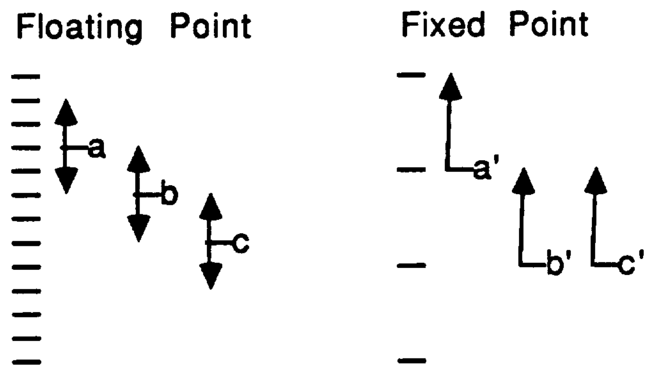


Figure 7. Floating Point Comparison vs. Fixed Point Comparison

A method is needed to ensure that values which differ by less than their specified accuracy in floating point format are considered equivalent when converted to their fixed point format. Such a method is to consider fixed point representations with equivalent exponents and whose mantissas differ by one or less as equivalent values. This changes the precision of the fixed point representation to

$$\begin{matrix} + < 2^{\text{exponent}} \\ - 2^{\text{exponent}} \end{matrix}$$

The corresponding number lines are shown in Figure 8. The values a', b', and c' are now considered equivalent.

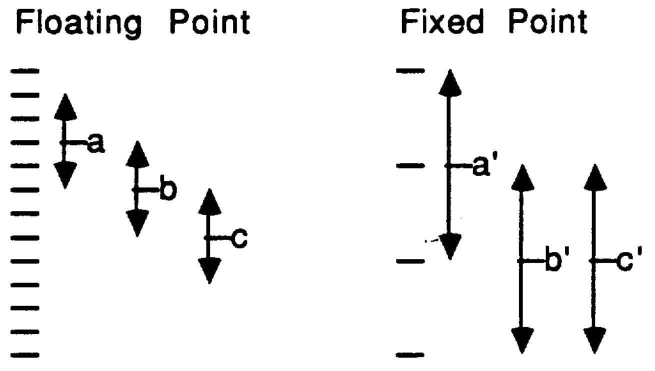


Figure 8. Floating Point Comparisons vs. Fixed Point Comparison Using the Precision Voter

The above method guarantees that values whose floating point representations differ by less than $2^{exponent}$ will be considered equivalent in their fixed point representations. An additional property of the method is that values which differ by as much as $2^{exponent + 1}$ may also be considered equivalent. The exponent value should be chosen such that $2^{exponent}$ corresponds to the accuracy interval specified for the version results. For the autoland simulation the exponent value was chosen to be 3. This yields a nominal precision of +/- 6.25E-2. The specified accuracy was +/- 5E-2.

One of the disadvantages of using the fixed point precision voter described above is that the accuracy of the results must be specified as a power of two. The second disadvantage is that values whose difference exceeds the specified accuracy may sometimes be considered equivalent. This is true only for differences less than twice the specified accuracy. If the above disadvantages are acceptable then use of the fixed point precision voter should speed the operation of the difference function required in design diverse implementations of fault tolerance.

4.1.2 Ordering and Placement

The precision voter returns a syndrome describing the relationship between the results returned by the versions which have been executed in this iteration. This operation syndrome describes which set of results agreed within the specified accuracy, which set of results disagreed within the specified accuracy, and which set of results was version timeouts. Agreement requires that the set is composed of at least two members and that all the members agree to within the specified accuracy; disagreement requires that none of the members of disagreement set agree with any other member of the set or with the result associated with an agreement set; a version timeout is a null result and indicates that due to a version failure the version did not execute to completion prior to the expiration of the timeout. This is distinct from an AP timeout which is the failure to respond due to an attached processor failure. The mechanism used to distinguish between the two is described in Section 4.2.

The operation syndrome is created by overlaying the equivalence interval corresponding to the specified accuracy of the versions onto the ordered set of result values. The placement of the equivalence interval should maximize the number of values within the interval. In some cases this constraint does not uniquely define the interval placement. Such a case occurs when the upper two values could be considered to be in agreement and the lower two values could be considered to be in agreement, or the middle two values could be considered to be in agreement and the upper and lower values could be considered to be in disagreement. This is shown in Figure 9. In such a case the latter choice could be selected to suppress the occurrence of pairwise splits.

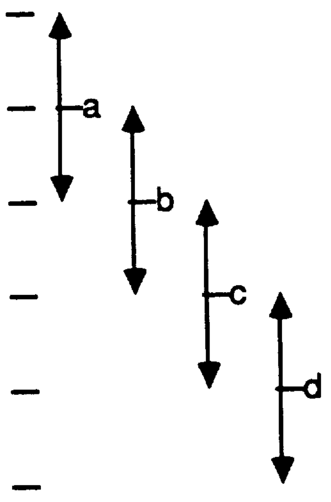


Figure 9. ab Agree/cd Agree or bc Agree/ad Disagree

Another case where the maximization of the number of results in agreement does not uniquely define the placement of the equivalence interval is when either the upper three or the lower three values could be considered in agreement and the remaining value in disagreement. This is shown in Figure 10. In the current implementation the upper three are chosen because of an arbitrary decision in the coding of the algorithm. In an enhanced implementation the choice might depend upon which value had been previously verified to be free from hardware faults. This would be applicable when the operation syndrome is re-evaluated after an isolation iteration. If the above choice was between a verified and unverified result as the disagreeing result, then choosing the verified result to be the disagreeing value would suppress the need for another isolation iteration to verify the otherwise unverified, disagreeing value. This would also suppress the number of operation syndromes which are unverifiable due to exhaustion of the number of allowed isolation iterations for each operational iteration.

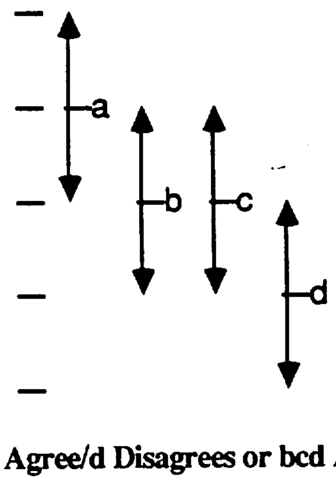


Figure 10. abc Agree/d Disagrees or bcd Agree /a Disagrees

4.1.3 Hardware Implementation

It would be desirable to implement the precision voter in hardware instead of software. This should increase the execution speed of this specialized and heavily used function. One problem with implementing a floating point precision voter is that the accuracy of each set of results to be voted must be known by the voter. This problem is not present in the implementation of a fixed point precision voter as described above. In the fixed point precision voter the relative precision of the fixed point mantissas is the relevant quantity and it is fixed at one unit regardless of the absolute accuracy of the corresponding results. The conversion from the floating point representation to the fixed point representation performs this translation.

Implementation of the fixed point precision voter in hardware is more difficult than the implementation of a bit for bit comparison. The precision voter must not only perform data dependent operations on selected members of its input, it must also uniquely encode the determined agreement relationship between the members of the input and return this code and the members to the hardware/software isolation software. The feasibility of a hardware implementation should be explored further if the performance of the fault tolerant software implementation is a primary goal.

4.2 Hardware/Software Isolation

At the completion of an operational iteration of the versions, there is recorded a result for each executed version. These results are analyzed by the precision voter and summarized in an operation syndrome. Results which disagree with the majority or are version timeouts may either be the result of an attached processor fault or a version fault. Isolation of the fault is required to determine which component is responsible. If the hardware is determined faulty then the it is masked from the system and the suspect result is replaced by a result which has been verified to be free from hardware faults. If a verified result disagrees with the majority then the associated version is faulty and it is recorded as such.

The isolation and verification is accomplished by an isolation iteration. Successive isolation iterations and a recursive reanalysis of the updated operation syndrome are performed until all suspect results are verified or the maximum recursion level has been reached. If all suspect results have been verified and a majority result is available then this result is returned, otherwise a default value is returned. The algorithm used in the analysis imposes several constraints on the behavior of software faults. These constraints must be enforced for isolation to be valid. The algorithm and the associated constraints will be discussed in the following sections.

4.2.1 Isolation Algorithm

The isolation algorithm must isolate the fault source for both disagreeing results and timed out results. A timed out result occurs when an attached processor executing a version does not return a result prior to the expiration of the timeout interval on the FTP. The timeout could be caused by a failure of the attached processor or of the version. If it is a failure of the version and it can be assumed that the fault has not propagated into the attached processor executive, then the attached processor should still be capable of responding to commands from the FTP. The FTP therefore sends a timeout command to the attached processor and if it is acknowledged then a version timeout is assumed. If the attached processor does not acknowledge then an AP timeout is assumed and the attached processor is masked from the system. When an AP timeout is detected, the subsequent actions of the system are as if the attached processor had been previously masked and its assigned version was not executed.

After a result or a version timeout has been recorded for each executed version, the precision voter determines the operation syndrome. If the syndrome indicates that there were not at least two agreeing, non-timeout results then isolation is not attempted and the default value is returned. (The default value is the value returned in the last iteration.) If the syndrome indicates a pairwise split, then the disagreement is assumed to be caused by a version fault and the confidence voter is called to resolve the split. Otherwise, those results which disagreed or were version timeouts are treated as suspect and isolation is attempted to determine the component responsible for the disagreement and to provide a verified result.

A suspect result is associated with a particular set of inputs, a suspect version, and a suspect attached processor. The isolation algorithm implemented consists of re-executing the suspect version using the same set of inputs on all available attached processors in an isolation iteration. If none of these attached processors are faulty then all results from the isolation iteration should be bit for bit congruent and the results should also be bit for bit congruent with the original suspect result. It is assumed that any disagreement indicates that the attached processor responsible for that result is faulty and that attached processor is accordingly recorded as such and masked from the system. The algorithm will detect transient as well as hard faults which occur in any of the executing attached processors during the isolation iteration. Because the algorithm compares the original result from the suspect attached processor with the result from the same attached processor in the isolation iteration, it will both detect and diagnose transient and hard failures in the suspect attached processor.

After the isolation iteration the suspect result and the corresponding version state is replaced by the majority result and version state from the isolation iteration. This result has now been verified to be free from the effects of hardware faults and any disagreement with other version results is caused by a software fault in the version. No majority result in the isolation iteration indicates multiple hardware failures and no recovery is attempted.

The operation syndrome is then updated based on the verified result and re-evaluated. If there are remaining suspect, unverified results then the cause of each of these is also isolated within the following constraints. Isolation is terminated when all suspect results have been verified to be free from hardware faults; there is a pairwise split; there are not two agreeing, non-timeout results; there are not sufficient attached processors for isolation; or the number of allowed isolation iterations has been exhausted. In the former two cases, if a majority result is present or the correct result can be determined by the confidence voter, then the version error history is updated and this result is returned. If a pairwise split cannot be resolved, then the error history is updated and a default value is returned. In all other cases the error history is not updated and a default value is returned. An enhanced implementation might return the majority value if it is available in the latter three cases. The number of allowed isolation iterations for each operational iteration in the AIRLAB FTP/AP fault tolerant software system is two. This allows isolation of all two hardware faults, one hardware fault and one software fault, or two software fault scenarios.

4.2.2 Software Faults

The above algorithm requires that a version fault does not propagate to its attached processor executive or to the other co-resident versions. This requirement must be enforced in the system implementation. If the requirement is enforced then the distinction between AP timeouts and version timeouts is valid. It can then also be assumed that the execution of identical software on identical hardware in the isolation iteration should yield bit for bit identical results and therefore any disagreement indicates a hardware failure in the associated hardware component. Enforcing this requirement requires some memory management on the attached processor to protect the memory associated with the attached processor executive and each of the versions. This is present on the VAX and could be provided on a microprocessor by the use of a memory management unit. The instructions executable by the versions must also be limited by either setting appropriate protections or by ensuring that the versions are executed in user mode on a microprocessor. The stacks used by each of the versions should also be distinct.

5.0 THE DECISION ALGORITHM

Recent research by Knight and Leveson indicates that one cannot assume that software failures are independent, random events [8]. In fact, Eckhardt contends that software failure rate is a function of its inputs [24]. This lack of independence severely reduces the reliability gained by using majority vote N-version software. For example, after studying several thousand 3-version systems, Knight and Leveson concluded that, for their specific application, the average reliability of these systems increased by only a factor of 19 over the average reliability of a single version [28]. For the FTP-AP architecture, therefore, a confidence voter has been developed that can provide a higher reliability than is possible with majority vote N-version software. The confidence voter does this by exploiting the correlated nature of software failures, as illustrated in Figure 5. In Figure 5 (which is not set to any scale), the area marked A is the subset of the input space where version A fails, area B is where version B fails, and so forth for versions C and D. In the input space where A's and B's failure regions overlap, every time B fails, A will fail as well. If A's and B's outputs were different, a simple majority or plurality voter can easily choose the output of versions C and D as being the correct answer. (This situation is known as a 2-1-1 split.) However, if the incorrect outputs of versions A and B were the same, a simple majority voter would be at a loss to choose between the correct pair, versions C and D, and the failed pair, versions A and B. (This situation is known as a 2-2 split.) The confidence voter takes advantage of the past history of these deterministic software failures and dependencies to choose the correct answer when a 2-2 split occurs.

In using the deterministic nature of software failures, the confidence voter cannot assume that the probability (A fails **and** B fails) equals the probability (A fails) **times** the probability (B fails). In fact, using the latter figure implicitly assumes that the failures are independent. Therefore, for a four version system the decision algorithm keeps a record of the frequency of each *pair* failing together, whether or not they fail with identical answers. The confidence voter makes the assumption that for double failures the probability that A and B fail with *identical* answers is proportional to the probability that A and B fail with *non-identical* answers.

This assumption allows the confidence voter to use 2-1-1 splits (that is iterations where the two incorrect values are not identical) to gather information on the frequency of each pair failing together and provides an *a priori* probability on which to decide a 2-2 split (iterations where the two incorrect values are identical). If the *a priori* failure probability for one pair is some threshold larger than the *a priori* failure probability for the other pair, then the more reliable pair is chosen. If the difference between the two pairs' failure probabilities is not large enough, the confidence voter will fail-safe.

Published data from the 27 version experiment done by Knight and Leveson[8] was used to analyze the distribution of double failures. A simple analysis of the pairwise failures was performed and each of the 17550 possible 4-version combinations were divided into 5 categories :

1. Combinations with no double failures
2. Single pair produce all double failures
3. One pair produces >10 times more double failures than any other pair
4. One pair produces > 2 times more double failures than any other pair
5. One pair produces < 2 times more double failures than any other

Figure 11 shows the distribution of the 4-version combinations among the 5 different cases. A quarter of the combinations had no double failures (Case 1) and a third had only one pair cause all the double failures (Case 2). The combinations with more than an order of magnitude difference in the number of failures caused by each pair also account for a large percentage (Case 3).

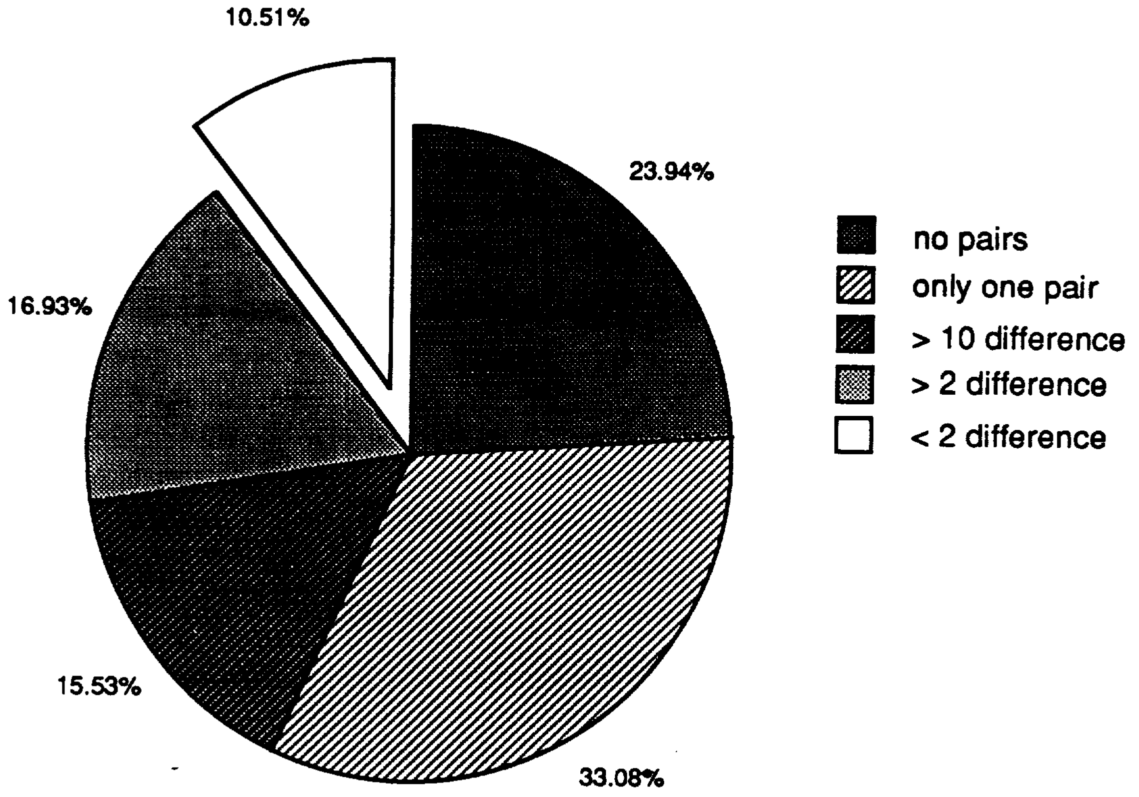


Figure 11. Types of 4-Version Combinations

The following subsections describe the operations of the confidence voter, the simulator that was used to test the confidence, and the results of the simulations.

5.1 The Confidence Voter

The actual operation of the confidence voter is fairly straight-forward. Each time a pair of versions fail, the failure counter for that pair is incremented. If there is a fail-safe, the counters for every pair are incremented. This way, the counters work under a worst-case assumption. When there is a 2-2 split, the confidence voter compares the failure rates for the two pairs. If the failure rate for one pair is some threshold larger than the failure rate of the other pair, then the more reliable pair is chosen. If the difference between the two pairs' failure rates is not large enough, the confidence voter will fail-safe.

In operation, then, the confidence voter behaves exactly like the majority (or "plurality") voter until it has gained enough knowledge to cross the threshold. This learning curve varies with the size of the threshold, the proportion of failures that are identical, and the difference in frequency of failures for the different pairs.

An important caveat is that there always must be some threshold. Without a threshold, it is possible for the decision process to influence the earliest stages of the learning process. This could push the confidence voter down the wrong path and ruin the system's reliability. The threshold is the *only* parameter affecting the learning curve that can be controlled and is based on the application. If in a given application, a fail-safe is relatively harmless compared to a catastrophic failure, then the threshold should be set high. This causes the confidence voter to be conservative and not decide a 2-2 split until it has a large amount of evidence in favor of one pair. On the other hand, if a fail-safe is almost as bad as a catastrophic failure, the a lower threshold should be set. A careful analysis of the costs of a fail-safe versus the costs of a catastrophic failure needs to be done for each application before any threshold can be substantiated.

5.2 The N-Version Simulator

The N-version simulator was created to test the operation of the confidence voter algorithm. It does this by mimicking the operation (and failure) of four versions of a program. The simulator is divided into two sections, a failure generator and a result generator. The failure generator determines which of the four versions will fail, and the result generator assigns both correct and incorrect floating point numbers as each version's results.

Since there are four versions, and on each iteration each version can either fail or not fail, there are 16 possible failure patterns. Each of the 16 failure patterns is assigned a probability based on Knights's 27 version experiment [8]. The simulator starts an "iteration" by randomly selecting a failure pattern. That failure pattern determines the result for each version. If there are no failures, each version is given an identical floating point number as a result. If one version has failed, the failed version is assigned a result that is different from the other three. When two or more versions fail, the result generator decides whether the failures are unique or identical, by using the parameter π, which is set to some value. The probability that two failed versions have the same incorrect value is the probability that versions choose the popular incorrect answer (this probability is π^2). If either version (or both) choose an unpopular incorrect answer, then the two results will be different (this probability is $1 - \pi^2$). After everything is computed, four floating point results, one from each version, are supplied to the confidence voter.

5.3 The Simulation Results

To compare the performance of the confidence voter against that of the majority (or "plurality") voter, one combination of the four versions was randomly picked from each of the four types of combinations that experienced double failures. Each of these cases was run through a million simulated iterations two times. For the first million iterations π was equal to 0.5, for the second million π was equal to 0.7.

Figure 12 shows the pattern of failures used for the first simulations. The numbers above each column refer to the version number assigned by Knight [8]. The X's indicate the version is failed for the given failure pattern and the numbers in the first column refer to the number of times each failure pattern should occur during a million iterations. Each of these first two combinations has only one pair of versions that create all the double failures.

During these simulations there were no catastrophic failures since there were no three or four multiple failures and the confidence voter never made a bad decision. Figure 13 shows the number of fail-safes that occurred during the first two simulations for both case 2a (versions 2,9,11,23) and case 2b (versions 2,9,11,8). The numbers is parenthesis after the case refers to the value of π for that simulation (either 0.5 or 0.7). In all simulations, the confidence voter shows an obvious improvement over the majority voter. The confidence voter also changes with time. When the confidence voter crosses its threshold and has enough information to choose one pair in a 2-2 split, it no longer has fail-safes.

The second two simulations had several pairs of versions that cause double failures, but one pair of versions was responsible for many more of the double failures than any other pair. The chosen case 3 combination, which must have at least a factor of 10 difference between the number of double failures, used version 1,14,19, and 20. There was a factor of 33 difference. The chosen case 4 combination, which must have at least a factor of 2 difference between the number of double failures, used versions 16,19,25, and 26. For this combination there was a factor of 4 difference. During these simulations, there was only one catastrophic failure which was due to the one triple failure in the pattern of failures for case 3. Figure 14 shows the number of fails safes that occurred during simulation for the case 3 and the case 4 combinations. In both of these cases, the confidence voter still shows an improvement over the majority voter. The last simulation had a number of catastrophic failures and fail-safes. The chosen case 5 combination

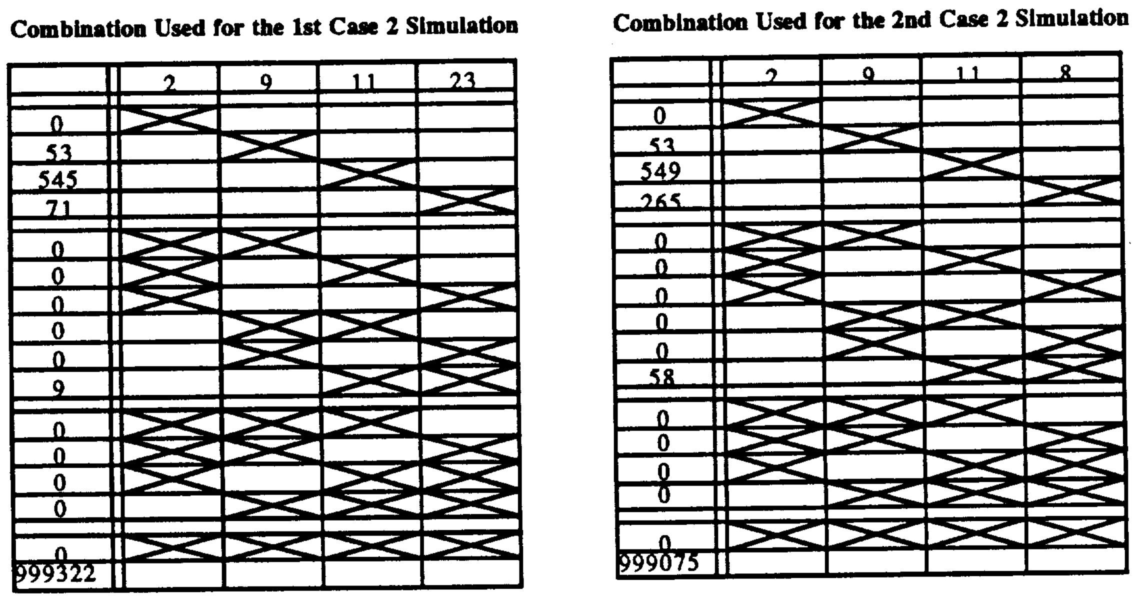

Combination Used for the 1st Case 2 Simulation
2
9
11
23
0
53
545
71
0
0
0
0
0
9
0
0
0
0
0
999322
Combination Used for the 2nd Case 2 Simulation
2
9
11
8
0
53
549
265
0
0
0
0
0
58
0
0
0
0
0
999075

Figure 12. Combinations with Only One "Bad" Pair

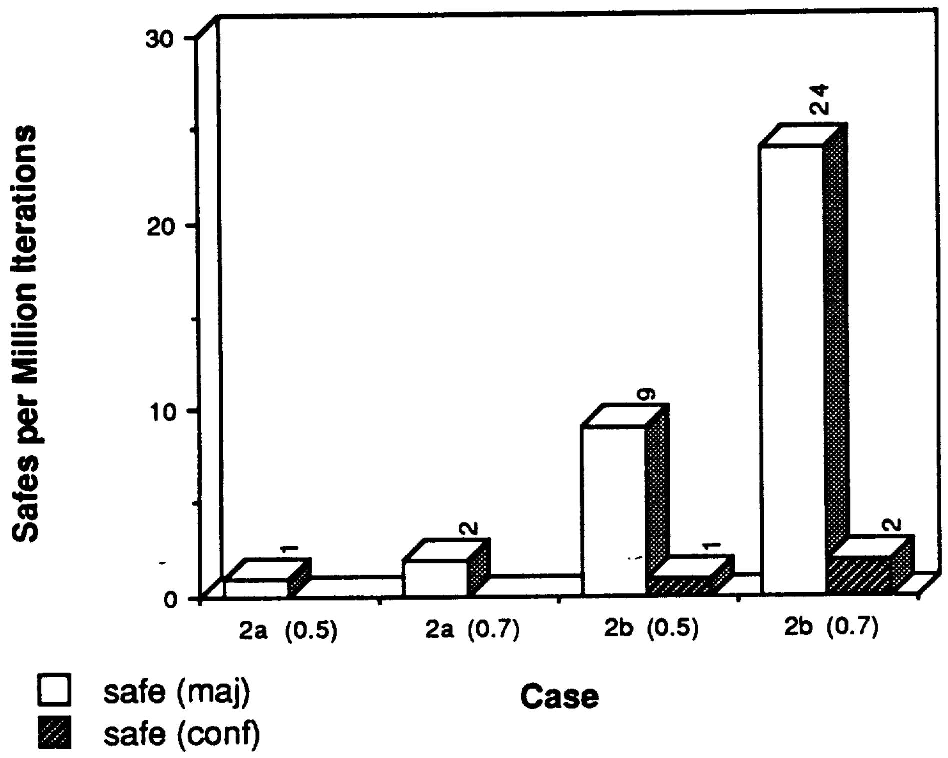


Figure 13. Results of the First Simulations

included versions 16,21,22, and 25. This combination had a small difference (a factor of 1.5), and had double failures from three different pairs. Because the confidence voter never crossed its threshold, it behaved exactly the same as the majority voter and the results were the same.

The results of these simulations show that the confidence can improve a 4-version system's reliability. In fact if the data that was used is indicative of multiple software in general, then we can expect that the confidence voter will improve the reliability in a large number of cases. Furthermore, if the assumption that the number of 2-1-1 splits a pair of versions has indicates how many 2-2 splits to expect, then the confidence voter will not decrease the reliability of the system.

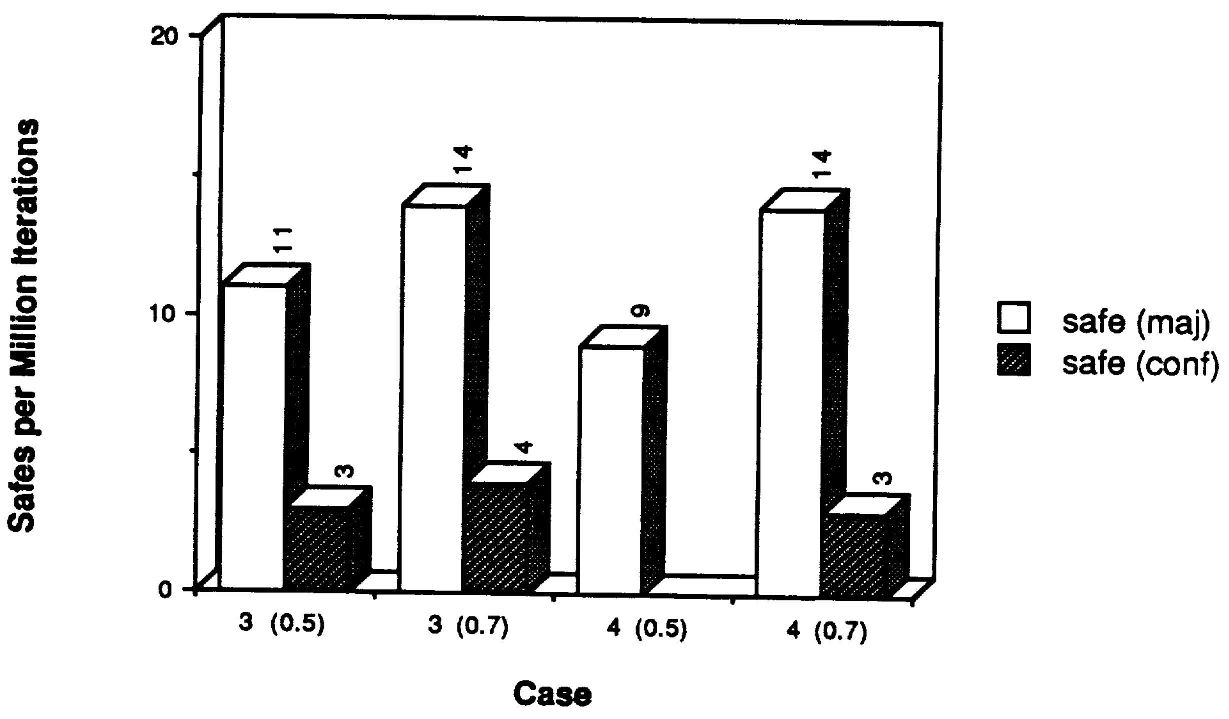


Figure 14. Results of the Second Two Simulations

6.0 EXPERIMENTS

Operation of the FTP/AP N-Version Fault Tolerant Software system is composed of several functions. These functions include the transfer of data and commands between the host, FTP, and attached processors; and the response of the host, FTP, and attached processors to the transferred data and commands. Verification of the operation of these functions over the range of all input conditions, and measurement of the performance of these areas to determine the bottle necks in the system are the desired goals of the experiments performed.

Many of the functions described have very restricted modes of operation and their correct implementation is sufficiently demonstrated during system execution of the autoland simulation. These functions include the data, state, and command transfer between the system elements; the generation of commands on the host; and the response of the attached processors to commands. The response of the FTP to the results returned by the attached processors has a greater range of possibilities than can be demonstrated in normal operation. A comprehensive test of the handling of the possible fault isolation scenarios which compose this response is required to verify the implementation. System operation is described in Section 6.1 and the verification of the fault isolation implementation is described in Section 6.2.

The comprehensive test described in Section 6.2 uses a specially constructed program executing on the attached processors to create the faults observed by the FTP. This method of verification has the inherent problem of generating only the fault behavior which is expected and for which handling capabilities were implemented. An alternative method of random fault insertion is described in Section 6.3 to test the hardware/software isolation algorithm.

Performance measurements of the areas of functionality described above are necessary to evaluate the capability of the system for real time execution of flight control. These measurements are described in Section 6.4.

6.1 System Operation

The FTP/AP N-Version Fault Tolerant Software system executes an autoland simulation. The yawdamper function of the simulation was selected for implementation as N-version fault tolerant software. Four versions of the yawdamper function were used: the original FORTRAN implementation in the simulation; a second FORTRAN implementation; a C implementation; and an ADA implementation generated by CASE, a computer aided software engineering tool.

Execution of the simulation is controlled by a host VAX through subroutines embedded within the simulation code. When execution of the yawdamper function is required, the host writes a run_yawdamp command and the parameters of the function to the host/FTP dual ported memory. When the FTP receives this command it assigns the available versions of the yawdamper function to the available attached processors, writes a run_yawdamp command, the yawdamper input, and any state variables required by the versions to the FTP/attached processor dual ported memory.

Each of the attached processors has resident an executive process and a process corresponding to each of the versions. These version processes are suspended waiting for an event flag to trigger execution of their version of the yawdamper function. When the run_yawdamp command is received by the attached processor executive, it reads its dual ported memory, determines the version that it has been directed to execute, and sets the appropriate event flag. This version then begins execution. When it is complete the version process writes its results and the updated state variable to dual ported memory, sets a version finished event flag, and waits for the next event flag that triggers execution of its version. The executive responds to the version finished event flag by writing a command_complete response to dual ported memory and returning to a state which will allow it to respond to the next run_yawdamp command.

At the completion of an operational iteration the FTP has a result from each active attached processor. It groups this set of results into subsets which agree, disagree, are version time-outs, or are AP time-outs. If there is a pair-wise split of non-time-out results, then no isolation is attempted. If the confidence voter does not have enough information to choose the "correct" value from the split, then the fail-safe value is returned. In this implementation the fail-safe value is the value from the last iteration. Likewise, if less than two results agree with each other, then no isolation is attempted and the fail-safe value is returned. Otherwise, a maximum of two isolation iterations are executed to isolate the origin of disagreeing results (including version time-outs). At the completion of the isolation iterations if a majority consensus exists and all disagreements have been isolated, or a "correct" value from a pair-wise split exists then this value is returned, otherwise the fail-safe value is returned.

Isolation is accomplished by running the version associated with the result in question on all available attached processors in an isolation iteration. In the absence of a hardware fault, all results should agree in the isolation iteration and agree with the suspect result from the original iteration. Those attached processors responsible for results which disagree are recorded as failed. The majority result from the isolation iteration replaces the suspect result in the original iteration. The evaluation of the results is then repeated and an additional isolation iteration may be performed to isolate remaining disagreements. If all disagreements have not been resolved after two isolation iterations, then the fail-safe value is returned.

After successful isolation, all disagreeing results due to faulty attached processors have been discarded. Any remaining disagreements must be the result of version failures and are logged and resolved by the confidence voter. This value is then returned to the host. It continues operation until the next call of the yawdamper function and then the process is repeated.

6.2 Verification of FTP/AP FTSW Fault Isolation Algorithm

6.2.1 Objective

The response of the FTP to data returned by the attached processors is composed of isolating the cause of any result disagreements and resolving a single value for the result. The cause of the disagreement is isolated to either a precision origin (by the precision voter), or a hardware or software origin (by hardware/software isolation). If the cause of the disagreement is the result precision, then the disagreement is handled by considering the results equivalent. If the cause is a hardware failure, then the offending hardware is removed from the system. If the cause is a software version failure, then the error is logged and used as a basis for further judgments on the validity of the results from that version. The result disagreements caused by component failures are then masked or replaced, valid state for the associated version is restored if available, and a resolved result is returned if available and all disagreements have been isolated. The previous resolved value is returned otherwise. The above functions are described as the precision voter, hardware/software isolation, and the confidence voter. The implementation of the precision voter and aspects of hardware/software isolation will be verified in this experiment.

6.2.2 Experiment Operation

Correct results from executed versions are only constrained to agree within the accuracy described in their common specification. A precision vote of the set of results from an operational iteration and of the updated set after each isolation iteration is necessary to determine if the results do agree within the specified accuracy. The vote implemented consists of converting the result representation to a fixed point representation with known precision and considering results in this representation which disagree by one or less to be equivalent.

The set of numeric results to be precision voted can be represented as a single result represented by 'a' and a subset of one or more results represented by 'b'. Let the result represented by 'a' equal 'a' and the results represented by 'b' each equal 'b' or 'b'-1 as discussed above. The following relationships between the results represented by 'a' and 'b' are then possible.

	condition:	state:
	a + 2 <= b	result represented by 'a' is not equal to results represented by 'b'
	a + 1 = b	result represented by 'a' is equal to results represented by 'b'
or	a = b	
	a - 1 = b	result represented by 'a' is equal to results represented by 'b' if all results represented by 'b' equal 'b'
	a - 2 >= b	result represented by 'a' is not equal to results represented by 'b'

Given a result 'w', a test of the handling of each of the above states for two results requires varying the range of a second result 'x' from 'w'-2 to 'w'+2. Likewise, a test of the handling of each of the above states for four results, 'w', 'x', 'y', and 'z', requires that

w is fixed
x varies from w-2 to w+2
y varies from x-2 to x+2
z varies from y-2 to y+2

The first set of values from the above would correspond to 'w', 'w'-2, 'w'-4, 'w'-6. A series of 124 sets of values would follow. This series of sets includes all the permutations of the agreement and disagreement states between the members of a set.

All the permutations can be achieved in a smaller series of sets if the ordering of members of the set is not relevant. In the FTP/AP FTSW system the ordering of results from an operational iteration is not fixed relative to any physical attribute of the system and the above condition is true. The smaller series of sets of 'w', 'x', 'y', 'z' can be generated with the following.

w is fixed
x varies from w to w+2
y varies from x to x+2
z varies from y to y+2

In addition to numeric results, version and AP time-outs are also possible. These possibilities are included by allowing the results 'w', 'x', 'y', and 'z' to vary throughout their numeric range and also through values interpreted as an AP time-out and a version time-out.

After being evaluated by the precision voter, the set of results is grouped into those members who agree, those which disagree, those which had a version time-out, and any remaining. A disagreement or version time-out may have either a hardware or software origin. The origin is isolated by running the suspect version on all available attached processors in an isolation iteration with the same state as in the original iteration. In the absence of a hardware failure all results from the isolation iteration should be identical. APs which disagree in an isolation iteration have had a hardware failure and are recorded as failed. The majority value from the isolation iteration is then used as the result value for this version and the result is recorded as confirmed. Any other non-confirmed disagreements or version time-outs are then isolated in at most one more isolation iteration.

Each set of the series described in conjunction with the precision voter corresponds to the results from an operational iteration. Each operational iteration may have two additional isolation iterations. After execution of each of these isolation iterations the set of results corresponding to the operational iteration is updated and reevaluated. It is desirable to test all permutations of the agreement and disagreement states for these subsequent isolation iterations. The ordering of the members of the set is now fixed and the values returned by isolation iterations must vary throughout the entire range of (least value - 2) to (largest value + 2), version time-out, and AP time-out to test the entire set of permutations.

The above set of tests will verify the isolation of software faults and those transient hardware faults which manifest themselves as a disagreement between the result from a processor/version pair in the operational iteration and the result from the same pair in an isolation iteration. In this experiment, isolation of hardware faults which manifest themselves as a disagreement between the results returned in an isolation iteration is not verified. An additional experiment to verify this implementation could be performed.

In the experiment the host only sends run_yawdamp commands and records the returned result. It does not execute the autoland simulation. The FTP performs as before with minor modification. In order to evaluate the complete series of result sets without resetting the system after every error, it was necessary to suppress the exclusion of faulty components from the system. The determination that components are faulty are logged, but they are removed from the system. These error and information messages are logged on an attached printer and used to verify the successful execution of the precision voter, and H/S isolation. These messages make the FTP log quite verbose; it may be desirable to suppress some of these messages during normal operation. Additionally, a location in FTP/AP dual ported memory to record the isolation recursion level is necessary as detailed below.

The attached processors have resident the executive task and a test task. The executive task is modified to always set the event flag triggering the test task when it receives a run_yawdamp command. The test task is the same on all attached processors. It generates the series of sets of values described above. Each set has six members; one for each version and one for each possible isolation iteration. When it is triggered by the event flag it writes the set of values for this iteration to a file and returns one of the members of the set as its result. The member depends on whether this iteration is for isolation and which version the attached processor was directed to execute. If the result is not a time-out the test task then sets the version finished event flag. The next set in the series is then determined and the test task waits for the next event flag trigger. If the attached processor executive sees the version finished event flag it returns the command_completed response to the FTP. Otherwise it will receive a timeout_command from the FTP. If the test task result was a version_timeout then the executive returns a command_complete response. If the result was an ap_timeout then the executive returns no response. In either case the FTP then waits for the next run_yawdamp command from the host.

The data used to evaluate the success of the experiment are the attached processor and host data files, and the FTP error log. Their entries are correlated by iteration number and isolation level. For a given set of values returned by the attached processors a known set of actions should occur on the FTP and a known result should be returned to the host. This information is used to confirm the successful completion of the experiment.

6.2.3 Summary of Results

The complete set of isolation scenarios consists of 1330 operational iterations and the associated isolation iterations. Due to the manner in which attached processor and version time-outs were synthesized for the experiment, the system was not tolerant of competing processes on the VAX. This and other VAX problems necessitated parsing of the set into manageable subsets. Execution of the experiment also revealed several faults in the isolation code. These were corrected and the corrections verified. Successful execution of the complete set of isolation scenarios with the corrected code took approximately 11 hours of system time.

The output used to confirm the successful evaluation of the isolation scenarios consisted of the attached processor log, the FTP log, and the host log. Confirmation required correlating the log entries and verifying that the entries were appropriate. The correlation and verification of the log entries was done manually. An annotated version of the output used to confirm the successful evaluation of illustrative basic isolation scenarios is included below.

The output from an attached processor is interspersed with the output from the FTP message log and the host log. The attached processor output lists the recursion level of the iteration, the value returned by each attached processor for that iteration, and for isolation iterations it also includes the version being executed and the updated values used to form the operation syndrome. A recursion level (RL) of zero corresponds to the operational iteration; a value of one corresponds to the first isolation iteration; and a value of two corresponds to the second and final isolation iteration. The operational values (OpVal) correspond to the values for version 1, 2, 3, and 4 respectively and are used to form the operation syndrome. In the operation iteration (RL: 0) these also correspond to the values returned by attached processor A, B, C, and D. The isolation values (IsVal) correspond to the values returned by attached processor A, B, C, and D after their execution of an isolation iteration of the version specified.

The output of the FTP message log is the sequence of messages corresponding to each of the selected operational iterations and its associated isolation iterations. It consists of an attached processor to version assignment such as AB->12. This indicates that attached processor A is executing version 1 and attached processor B is executing version 2 in the operational iteration. The corresponding operation syndrome is listed in the form a12_d3_v. This example indicates that the results from versions 1 and 2 agree, the result from version 3 disagrees with that result, there were no version time-outs, and version 4 was not executed or its associated attached processor experienced an AP time-out. The isolation syndrome has the same format with the versions replaced by attached processors which are executing a specified version. The updated operation syndrome is listed after each isolation iteration and the value returned to the host is listed when the isolation is complete.

The output of the host log is the result returned for use by the host in that operational iteration.

Example 1: All results agree.

```
AP:
        RL:   0                    OpVal:      50     50     50     50

FTP:
        AP to version assign, ABCD->1234
        Operation syndrome, a1234_d_v
        Result from version 1 returned

host:
        Rslt:  50
```

Example 2: Disagreement due to precision.

```
AP:
        RL:   0                       OpVal:      50     50     50     51

FTP:
        AP to version assgn, ABCD->1234
        Operation syndrome, a1234_d_v
        Result from version 1 returned

HOST:
        Rslt:   50
```

Example 3: Disagreement due to software fault.

```
AP:
        RL:   0                       OpVal:      50     50     50     52
        RL:   1        Ver:   4       IsVal:  52  52     52     52
                                      OpVal:      50     50     50     52

FTP:
        AP to version assgn, ABCD->1234
        Operation syndrome, a231_d4_v
        HW/SW isolation of Ver 4, AP D
        Isolation syndrome, aABCD_d_v
        Operation syndrome a231_d4_v
        Software fault in version 4
        Result from version 2 returned

HOST:
        Rslt:   50
```

Example 4: Disagreement due to transient hardware fault.

```
AP:
      RL:   0                   OpVal:        50     50     50     52
      RL:   1      Ver:   4     IsVal:  50    50     50     50
                                OpVal:        50     50     50     50

FTP:
      AP to version assgn, ABCD->1234
      Operation syndrome, a231_d4_v
      HW/SW isolation of Ver 4, AP D
      Isolation syndrome, aABCD_d_v
      Transient fault in AP D
      AP D taken off-line
      Operation syndrome a2314_d_v
      Result from version 2 returned

HOST:
      Rslt:   50
```

Example 5: Disagreement due to version time-out software fault.

```
AP:
      RL:   0                   OpVal:        50     50     50     0
      RL:   1      Ver:   4     IsVal:  0     0      0      0
                                OpVal:        50     50     50     0

FTP:
      AP to version assgn, ABCD->1234
      Version time-out from version 4
      Operation syndrome, a231_d_v4
      HW/SW isolation of Ver 4, AP D
      Version time-out from version 4
      Isolation syndrome, a_d_vABCD
      Operation syndrome a231_d_v4
      Software fault in version 4
      Result from version 2 returned

HOST:
      Rslt:   50
```

Example 6: Detection of an AP time-out.

```
AP:
        RL:    0                         Val:   50      50      50      -1

FTP:
        AP to version assgn, ABCD->1234
        AP Time-out from AP D
        AP D taken off-line
        Operation syndrome, a231_d_v
        Result from version 2 returned

HOST:
        Rslt:   50
```

In addition to confirming the expected execution of the basic isolation scenarios, the experiment revealed interesting behavior for some of the more complicated scenarios. These scenarios involve the occurrence of more than two errors in an operational iteration and the associated isolation iterations. This should be a rare occurrence, but the algorithm could be altered if it was determined desirable.

The first characteristic is illustrated by the following example.

Example 7: Valid result or fail-safe?

AP:

RL:	0			OpVal:		50	52	53	55
RL:	1	Ver:	1	IsVal:	52	52	52	52	
				OpVal:		52	52	53	55
RL:	2	Ver:	4	IsVal:	51	51	51	51	
				OpVal:		52	52	53	51

FTP:

AP to version assgn, ABCD->1234
Operation syndrome, a23_d14_v
HW/SW isolation of Ver 1, AP A
Isolation syndrome, aABCD_d_v
Transient fault in AP A
AP A taken off-line
Operation syndrome, a123_d4_v
HW/SW isolation of Ver 4, AP D
Isolation syndrome, aABCD_d_v
Transient fault in AP D
AP D taken off-line
Operation syndrome, a412_d3_v
Bad rslts after isol, fail-safe

HOST:

Rslt: 50

Due to the syndrome algorithm not taking into account which values had already been confirmed this scenario resulted in a fail-safe. This is because the disagreeing result from version 3 was not confirmed in either of the two isolation iterations. Alternatively versions 1,2, and 3 could have been considered to agree, then the disagreeing result from version 4 would have already been confirmed and a valid result would have been returned.

The example also illustrates the second characteristic. In the example a majority result is present, the result is not returned because the isolation disagreeing results has not been completed. Instead the value from the last iteration is returned and the error history is not updated. Alternatively, it may be desirable to return the majority result and to not update the error history corresponding the version with the unconfirmed, disagreeing result. This would reduce the number of failsafes. (The number of isolation iterations could also be increased, but this has performance penalties.)

6.2.4 Conclusions

The above experiment demonstrated that the hardware/software isolation has been implemented as described with respect to software faults and with respect to transient hardware faults which manifest themselves as disagreements between the result from an AP/version pair in the operational iteration and the same pair in an isolation iteration. Isolation of hardware faults that manifest themselves as disagreements between AP/version pairs in the isolation iteration were not verified. Experimental verification of this aspect of the algorithm could be done using the same system configuration. The experiment also did not verify the isolation algorithm implementation when operating with less than four usable attached processors. The experiment could be re-executed under these conditions to provide verification.

The handling of isolation scenarios involving three or more faults could be enhanced to minimize the occurrence of failsafes. This must be weighed against the assumed rarity of such scenarios versus the overhead which the enhancements would impose upon every isolation scenario. The desirability of such enhancements should be evaluated before implementation is attempted.

6.3 Fault Injection Experiment

6.3.1 Objective

The purpose of this experiment is to demonstrate that the hardware/software isolation algorithm can correctly identify the source of a failure as either hardware or software. Our goal is to simulate a hardware failure by means of a fault injector on a single AP, determine that the fault was identified as hardware rather than software, and that the system continues to function correctly in the presence of the fault. The FTP's error log and status screen is displayed on a local VT100 terminal. This display identifies the fault as a permanent hardware fault, a transient hardware fault or a software fault. If the fault is identified as a hardware fault, the faulty AP is logged as being taken off-line along with the reason it is taken off-line. This log should demonstrate that the isolation algorithm can correctly identify a hardware fault. No experiment was run to demonstrate that the isolation algorithm will correctly identify a transient hardware fault, but the fault injector software could be easily modified to run such an experiment.

6.3.2 Description

The fault injector is implemented with a software module which runs simultaneously with the programs on the host, the FTP, and the APs. This FORTRAN program injects errors by continually changing the AP's dual ported memory. Changing the memory of a particular AP simulates a dual ported memory fault in the chosen AP. The AP chosen to host the fault injector was the AP that was associated with the channel running version 3 of the yaw damper (System 6). The AP running version 3 was chosen since version 3 was the only version that had a known software fault. Version 3 was identified as having a software fault in all previous runs of the autoland simulation. In all previous runs no AP hardware fault was injected or identified.

This experiment requires all four APs to be active, the FTP error display to be shown, and a VT100 terminal connected to the AP which runs the fault injector program.

Running this experiment is similar to running the demonstration of the fault tolerant software described in Section 6.1. It requires no changes in the FTP load module and no changes in the software which is run on the APs. The fault injector program was started up after a few iterations of the autoland simulation, in order to verify that all APs did come on-line.

6.3.3 Results

This experiment was conducted for several runs of the autoland simulation. In each run of the experiment, the fault injector program was started after several iterations of the simulation. Before the fault injector was started, the four APs were confirmed to be fault free and on-line, and the results of the simulation were confirmed to be correct. The output from the display programs running in the FTP and the host was the means of confirming that the system was running fault free. Once the fault injector was started the FTP error screen displayed that an isolation iteration was executed. After the isolation iteration, the error screen displayed that a hardware fault had been identified and isolated to the AP on channel C. The error log then displayed that the AP attached to channel C was taken off-line, and version 3 (which was previously assigned to channel C's AP) was assigned to channel D's AP. Channel D's AP, however, did accumulate a number of software errors after version 3 had been assigned to it. The autoland simulation was rerun without the hardware fault injector and version 3 (running on channel C's AP) was observed accumulating software errors.

This experiment demonstrated that the isolation algorithm can identify the source of an AP's failure as either the hardware or the software resource.

The FTP's error screen was very helpful in verifying the FTP's ability to isolate the source of a failure. The following is a scenario of the messages which appeared on the FTP's error screen before, during, and after the execution of the hardware fault injector program:

```
AP to version assign, ABCD -> 1234
Operation syndrome, a124_d3_v
HW/SW isolation of Ver 3, AP C
Isolation syndrome aABD_dC_v
```

Hardware fault in AP C
AP C taken off-line
Operation syndrome a1234_d_v
Result from version 1 returned

Figure 15, a diagram of the FTP status screen, shows the status of the processors, clocks, and APs while the 737 autoland simulation is running without the hardware fault injector. The status of the APs, before the 737 autoland simulation is run, is failed. As figure 15 indicates, the status of all the APs has been changed to on-line.

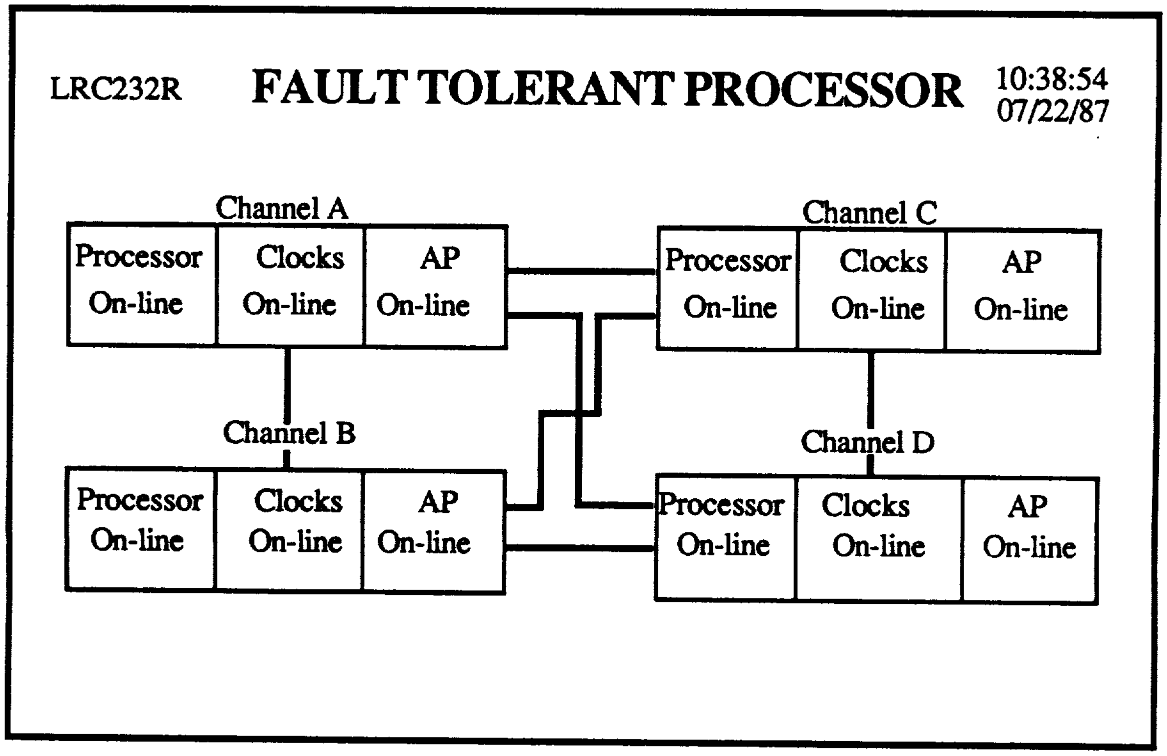


Figure 15. FTP Status Display Before Hardware Fault

Figure 16 is a diagram of the N-version software status screen. It indicates which version is running on the various APs, the status of the version, and the number of software errors per iteration of the yawdamper code. Also shown on the screen is the number of errors per iteration between each version. It should be noted that version 3 was the version which has intermittent software faults. This can be verified by its error count of 2.

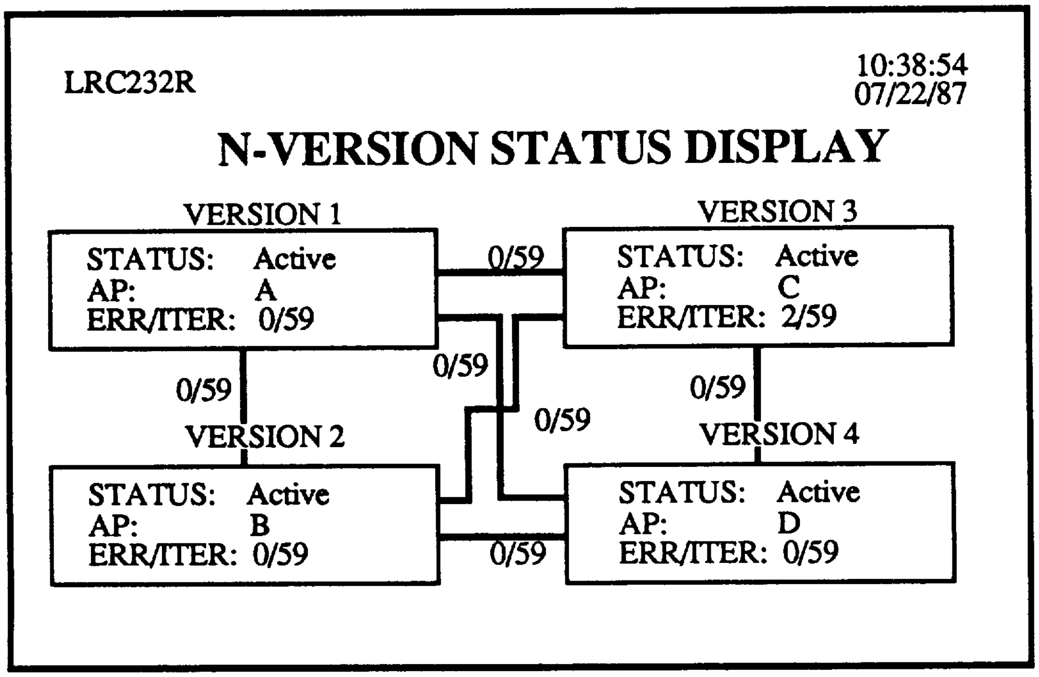


Figure 16. N-Version Status Display Before Hardware Fault

Figure 17 shows the failure of AP C while running the autoland simulation. The failure occurred shortly after the fault injector program was started on channel C's AP.

Figure 18 indicates the new assignments of the versions to the APs after the hardware fault injector was run. Although version 3 was running on the faulty AP, it still remains active while being reassigned to AP D. Version 4, however, became idle with no AP being associated with it. It should be noted that the error portion of the software errors per iteration did not change because the source of the error was isolated to hardware.

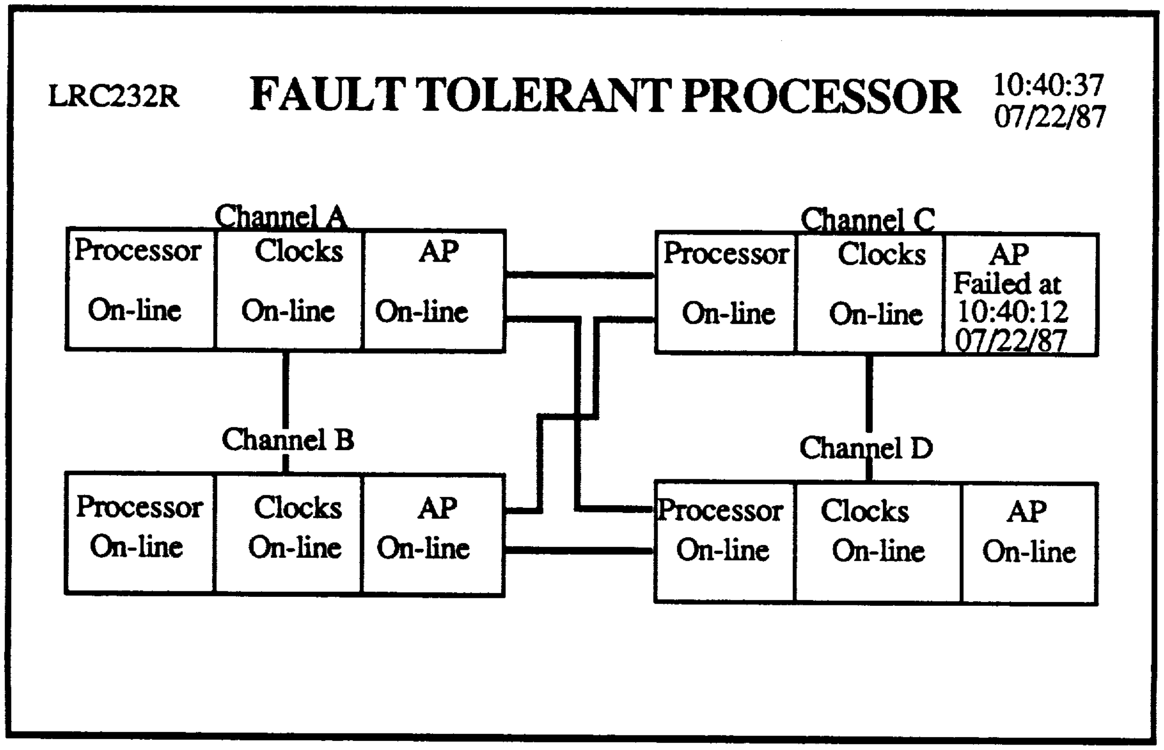
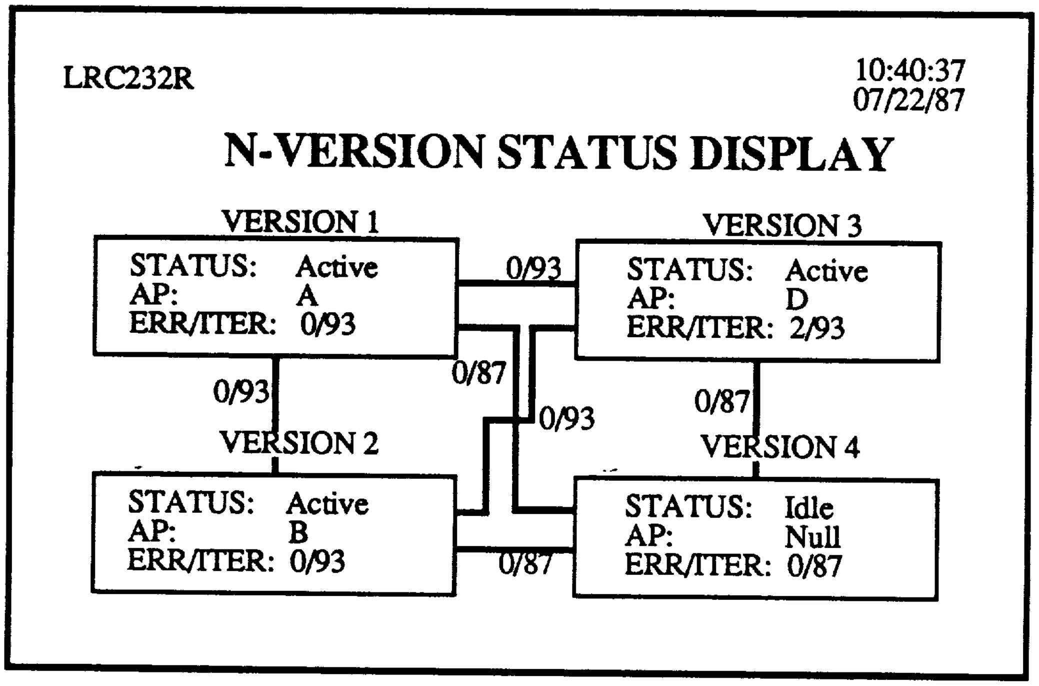

LRC232R FAULT TOLERANT PROCESSOR 10:40:37 07/22/87

Figure 17. FTP Status Display After Hardware Fault


LRC232R 10:40:37 07/22/87

N-VERSION STATUS DISPLAY

Figure 18. N-Version Status Display after Hardware Fault

6.4 Performance Measurements

6.4.1 Objective

One of the goals of the architecture was to minimize the transport lag time between the reading of the sensors and the output to the actuators. In many of the past N-version experiments, the versions were run sequentially on one processor. This overhead problem is solved by running the four versions of the application in parallel on the four attached processors so that they take the same amount of time to run as a single version on a single processor. Still, there is the added overhead for communication software, the confidence voter, and the hardware/software isolation required for the CSDL unified hardware and software fault tolerant architecture. The purpose of this experiment was to determine if the architecture is able to meet the real time requirements of the autoland application.

6.4.2 Description

The system operation as described in Section 6.1 was used to gather timing information from the FTP/AP N-Version Fault Tolerant Software System. The autoland simulation runs on the host VAX and in normal operation it invokes the yawdamper subroutine, also resident on the host VAX, as part of its operation. The setup that was used in the timing measurements was as follows: the autoland simulation signaled the FTP instead of invoking the resident subroutine and then waited for the result from the FTP before completing an iteration. Since neither the host VAX nor the APs are able to interrupt the FTP, the FTP reads all its signals from the VAXs by polling locations in the dualported memories. Polling for the signal from the host VAX is normally done every 40 milliseconds. If the FTP sees the signal to run the yawdamper, it writes the input data to the dualported memory of the four attached processors and interrupts them. When each AP finishes execution of a version, it writes its output and state variables in dualported memory and signals the FTP. Again the FTP must poll the dualported memories of the four APs for this signal. Figure 19 is a more detailed description of the control flow of the tasks or processes within the processors (Host VAX, FTP and APs). It also shows the interaction of these tasks. The places where the polling is done by the FTP is shown by the darkened areas. This polling is normally done at 25 hertz. For the experiment, the polling was done at four rates: 25 hertz, 33.33 hertz, 50 hertz and continuous. Timings were taken on the host VAX, the APs and the FTP.

Timings done on the host, which used the VAX clock, has accuracy to 10 milliseconds. Since the autoland simulation runs 3786 iterations of the control law, timings were saved for each iteration and averaged to gain more accuracy Three timings were made on the host. For a baseline, the time required to complete one iteration of the autoland simulation under normal conditions was measured. Normal conditions means that the yawdamper subroutine is resident on the host VAX and the FTP/AP N-Version Fault Tolerant Software System is not active. Timing 2 measured the interval from

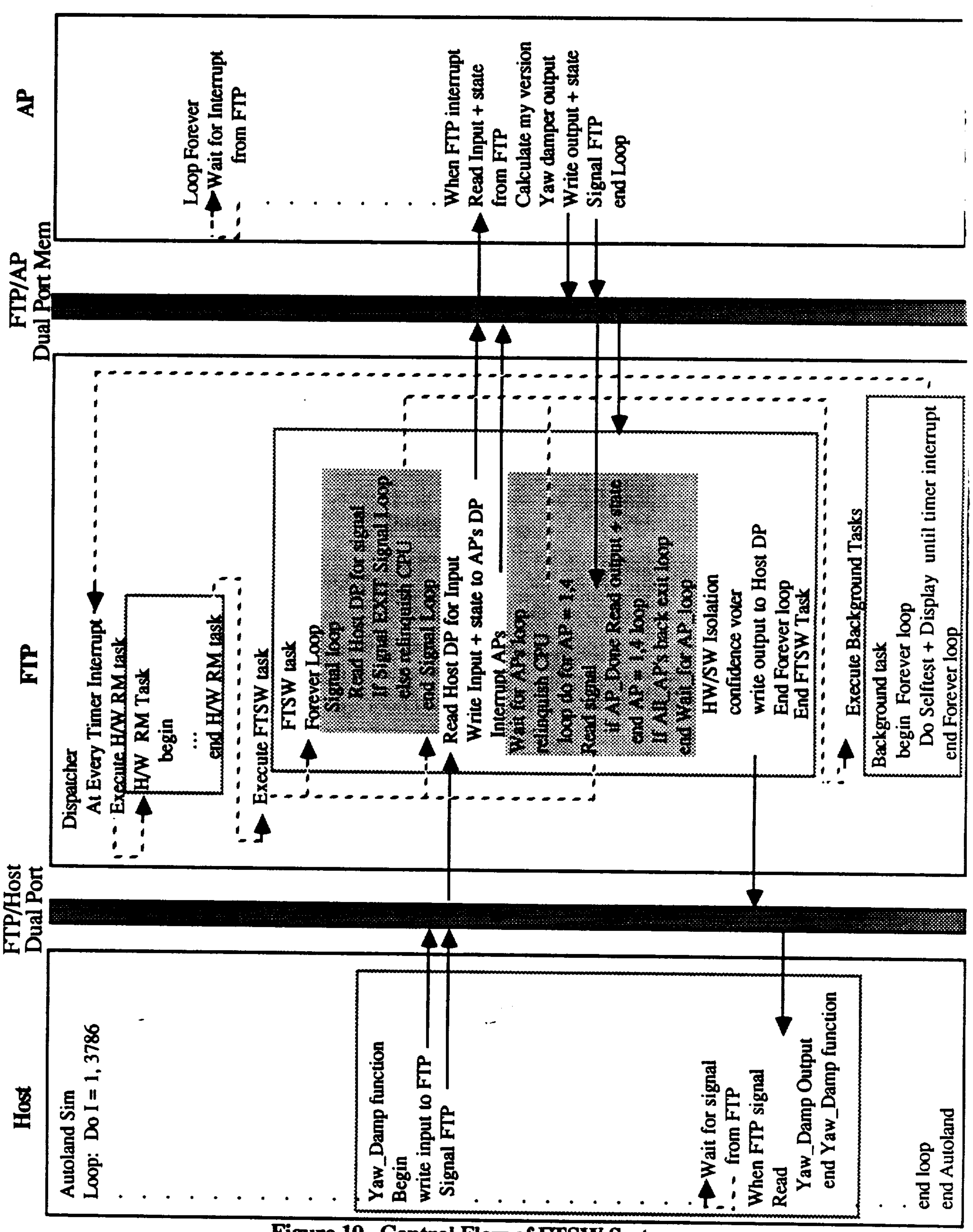


Figure 19. Control Flow of FTSW System

ORIGINAL PAGE IS
OF POOR QUALITY

the host signal to the FTP until the FTP sent the yawdamper output back to the host. Timing 3 measured the time required to complete one iteration of the autoland simulation running the FTP/AP N-Version Fault Tolerant Software System. The polling for these measurements with the FTP/AP system was done at 25 hertz, 50 hertz and continuous.

The APs' clocks were used for the timings done on the APs. They also have accuracy to 10 milliseconds so the average of 3786 iterations was calculated to gain better accuracy. For timing 4, the APs measured the period from when the signal was received to execute their version to the time they wrote their response back to the FTP. In timing 5, the APs measured the period required by the FTP to read the results. This was done by having the FTP signal the APs when the result was read. The timing 5 measurement was taken when the polling intervals were 25 hertz and 33.33 hertz.

Timing 6, done on the FTP, used the Langley Data Acquisition System (DAS) and a logic analyzer. The FTP measured the interval from the time the FTP reads the input from the host VAX until the time the FTP interrupts the host VAX with the output. This measurement was taken when the polling interval was 25 hertz.

6.4.3 Summary of Results

Figure 20 is a summary of the results of the experiment. It indicates that polling for results wastes approximately half the polling period.

Timings	Done by	Polling Interval	Time
1. Normal Autoland	Host	-------	35.8 msec*
2. Total YawDamper with FTP/AP FTSW System	Host	25 Hz	88 msec
		50 Hz	75 msec
		continuous	46 msec
3. Total Autoland Simulation with FTP/AP FTSW System	Host	25 Hz	138 msec
		50 Hz	125 msec
4. YawDamper Execution on APs	APs	--------	11 msec
5. FTP Read Result from APs	APs	25 Hz	18 msec
		33.33 Hz	10.6 msec
6. YawDamper Execution, Communication SW, HW/SW isolation and confidence voter	FTP	25 Hz	69.5 msec

Figure 20. Summary of Timing Results

* Measured independently under real-time conditions found to be 20 msec.

6.4.4 Conclusions

The fact that a VAX to FTP interrupt was not implemented in the FTP/AP system causes a severe performance penalty. If the system is run in the continuous mode, the timing for the total yawdamper is 46 milliseconds and the total autoland simulation is 82 milliseconds. In order to meet the real time requirements of an application, this interrupt must be implemented. The overhead required for the other software in the system is equal to the total yawdamper time minus the time for the execution of the yawdamper routines on the APs, that is 46 - 11 or 35 milliseconds. The entire normal yawdamper autoland simulation takes 35.8 milliseconds. So if the Autoland Simulation was setup to run at 20 hertz or every 50 milliseconds, the VAX to FTP interrupt was implemented, and the Autoland Simulation did not wait on the results from the FTP/AP FTSW system but sent input followed by an immediate read for output, it would run in real time. The Autoland Simulation would be getting the results from the n-1 calculation so that the FTP/AP FTSW would be running concurrently with Autoland Simulation.

The dhrystone benchmark was run in order to compare the AIPS FTP that is 68010 microprocessor based with an 8 megahertz clock and the CSDL IR&D VLSI FTP that is 68020 microprocessor based with a 16 megahertz clock. The VLSI IR&D FTP ran 7 times faster. VAX to FTP interrupts have been implemented in this VLSI implementation of the FTP. Therefore, the 46 millisecond overhead for running on the FTP/AP FTSW architecture would be greatly reduced with the 68020 implementation of the FTP/AP architecture. Therefore, the FTP/AP FTSW architecture with a 68020 microprocessor and a VAX to FTP interrupt would meet the necessary real time requirements of an autoland application.

7.0 CONCLUSIONS

In the NASA sponsored study of a Unified Hardware and Software Fault Tolerant Architecture, an FTP/AP system was implemented to execute N-version fault tolerant software. The study proposed and implemented solutions to several of the basic problems associated with N-version software. These problems are the resolution of results when the agreement condition is not defined as bit for bit congruence, the resolution of a pairwise split in the results caused by the occurrence of coincident errors, and ensuring that hardware faults are not interpreted as software faults. With the exception of the latter, these are generic solutions to the described problems and are not dependent on the implementation of an FTP/AP architecture.

The ability to correctly resolve a pairwise split in results based on the past behavior of the versions has the potential to decrease the failure rate which would otherwise be exhibited by several orders of magnitude. The proposed confidence voter maintains an error history of the versions and uses this history to resolve a pairwise split based on the relative confidence in the pairs. Execution of the confidence voter on input simulated according to published data from the 27 version experiment performed by Knight and Leveson demonstrated that system reliability could be increased by the use of the confidence voter. The confidence voter was never shown to decrease the system reliability. Further research should be conducted to determine the generality of these results and to gain more familiarity with the behavior of the confidence voter.

In conjunction with the research on the confidence voter, a Markov model was developed for software failures. The model is based upon the underlying mechanism responsible for the occurrence of software failures. The validity of the model and the ability of the requisite parameters to successfully quantify the behavior of software failures requires more research into how multiple versions fail.

The AIRLAB FTP/AP system was designed to provide the functionality required for highly reliable execution of N-version software. An FTP/AP system should also be capable of real time execution of the N-version software while maintaining this level reliability. The real time performance of such a system would be enhanced by an architecture with an optimized communication mechanism between the FTP and attached processors and a floating point coprocessor. The Advanced Information Processing System FTP dual processor architecture could provide such a system. Communication between the dual processors composing a channel of the AIPS FTP is optimized for speed and one of the processors from each channel of the FTP could be used as the attached processor.

The structure and algorithms of the system software implemented on the AIRLAB FTP/AP FTSW system should be directly applicable to an AIPS FTP implementation. An AIPS FTP executing N-version fault tolerant software should also be functional as a node within the larger AIPS system. A proposed implementation would have the computation processors of the node functioning as the asynchronous APs and the synchronous IO processors performing their IOP function while also controlling the operation of the APs. The AIPS FTP would also include a floating point coprocessor so that the control law calculations would meet the performance requirements. It is also possible to execute N-versions of an application in the core FTP itself without the attached processors and resynchronize the redundant processors in the FTP after each iteration of the application. This will reduce the number of processors from eight (needed for the FTP-AP architecture) to four for this architecture. The overheads of synchronization after each iteration of the N-version application need to be quantified to determine the viability of this architecture. The applicability of implementing N-version fault tolerant software on the Core FTP, the AIPS FTP, and other architectures such as the the Fault Tolerant Parallel Processor should be the subject of future research.

Research into version recovery and the ability to implement multiple functions of an application as N-version software are also necessary if N-version software is to become a viable solution to the problem of software reliability. This and other research on N-version software could use the developed AIRLAB FTP/AP FTSW system as a testbed to further this goal.

8. 0 REFERENCES

[1] A. L. Hopkins Jr., T. B. Smith III, and J. H. Lala , "FTMP-A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft", *Proceedings of the IEEE*, Vol. 66, No. 10, October 1978.

[2] J. H. Wensley, et al., "SIFT: the Design and Analysis of a Fault-Tolerant Computer for Aircraft Control", *Proceedings of the IEEE*, Vol. 66, No. 10, October 1978.

[3] J. H. Lala, L. S. Alger, R. J. Gauthier, and M. J. Dzwonczyk, "A Fault Tolerant Processor To Meet Rigorous Failure Requirements", The 7th AIAA-IEEE Digital Avionics System Conference, Fort Worth, Texas, October 1986.

[4] A. L. Hopkins Jr., J. H. Lala, and T. B. Smith III, "The Evolution of Fault Tolerant Computing at the Charles Stark Draper Laboratory, 1955-85", Dependable Computing and Fault Tolerant Systems, Vol. I: *The Evolution of Fault-Tolerant Computing*, ISBN 0-387-81941-x, pp.121-140, Springer-Verlag, Wien, Austria, 1987.

[5] M. Lipow, "Prediction of Software Failures", *The Journal of Systems and Software*, Vol. 1, pp 71-75, 1979.

[6] A. Avizienis,"Fault Tolerance and Fault Intolerance: Complementary Approaches to Reliable Computing", *Proceedings of 1975 International Conference on Reliable Software*, Los Angeles, California, April, 1975.

[7] W. R. Elmendorf, "Fault-Tolerant Programming", Digest of Papers FTCS-2: *The 2nd Annual International Symposium on Fault Tolerant Computing*, Newton, Massachusetts, June 1972.

[8] J. Knight, N. Leveson, and L. St. Jean, "A Large Scale Experiment in N-Version Programming", Digest of Papers FTCS-15: *The 15th Annual International Conference on Fault Tolerant Computing*, Ann Arbor, Michigan, June 1985.

[9] A. Avizienis and J. Kelly, "Fault Tolerance by Design Diversity: Concepts and Experiments", *IEEE Computer*, August 1984.

[10] L. J. Yount, "Architectural Solutions to Safety Problems of Digital Flight Critical Systems for Commercial Transports", *Proceedings of the 6th AIAA/IEEE Digital Avionics Systems Conference*, Baltimore, Maryland, December 1984.

[11] A. Hills, "A310 Slat and Flap Control System Management & Experience", *Proceedings of the 5th AIAA/IEEE Digital Avionics Systems Conference*, Seattle, Washington, November 1983.

[12] R. Troy and C. Baluteau, "Assessment of Software Quality for the Airbus A310 Automatic Pilot", *Fault Tolerant Considerations and Methods for Guidance and Control Systems*, AGARDograph, No. 289, July 1987.

[13] D. A. Mackall and S. D. Ishmael, "Qualifications of the Flight Critical AFTI/F-16 Digital Flight Control System", *The 21st Aerospace Sciences Meeting*, AIAA-83-0063, Reno, Nevada, January 1983.

[14] J. H. Lala, "A Byzantine Resilient Fault-Tolerant Computer for Nuclear Power Plant Applications", Digest of Papers FTCS-16: *The 16th Annual International Symposium on Fault Tolerant Computing*, Vienna, Austria, July 1986.

[15] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem", *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 3, July 1982.
[16] M. Pease, R. Shostak, and L. Lamport, "Reaching Agreement in the Presence of Faults", *Journal of the ACM* , Vol. 27, No. 2, April 1980.
[17] D. Dolev, "The Byzantine Generals Strike Again", *Journal Of Algorithms*, Vol.3, pp. 14-30, January 1982.
18] L. S. Alger and J. H. Lala, "A Real Time Operating System for a Nuclear Power Plant Computer", *Proceedings of the IEEE Real-Time Systems Symposium*, New Orleans, Louisiana, December 1986.
[19] J. Kershaw, "VIPER", IEE Colloquium on VLSI Architectures, Digest 32, London, England, March 1987.
[20] T.B. Smith, III, "Synchronous Fault Tolerant Flight Control Systems," AIAA Computers in Aerospace Conference III, San Diego, CA, October 1981.
[21] A. Avizienis,"The N-Version Approach To Fault Tolerant Software", *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 12, December 1985.
[22] L. Chen and A. Avizienis, "N-Version Programming: A Fault Tolerant Approach to Reliability of Software Operation," Digest of Papers FTCS-8: *The 8th Annual International Conference on Fault Tolerant Computing*, Toulouse, France, June 1978.
[23] J.P.J. Kelly, et al, "Multi-Version Software Development", in Proceedings IFAC Workshop SAFECOMP'86, Sarlat, France: October 1986, pp. 43-49.
[24] D. Eckhardt Jr. and L Lee, "An Analysis of the Effects of Coincident Errors on Multi-Version Software", The AIAA Computers in Aerospace V Conference, Long Beach, California, October 1985.
[25] P. Ammann and J. Knight, "Data Diversity: An Approach To Software Fault Tolerance", Digest of Papers FTCS-17: *The 17th Annual International Symposium on Fault Tolerant Computing*, Pittsburg, Pennsylvania, July 1987.
[26] "Reliability Prediction of Electronic Equipment", MIL-HDBK-217D, Pages 5.1.14-1 to 5.1.14-7, Department of Defense, Washington, D.C. 20301.
[27] J. Knight and P. Ammann, "An Experimental Evaluation of Simple Methods for Seeding Program Errors", *IEEE Transactions on Reliability*, Vol. SE-11, No. 12, December 1985.
[28] J. Knight and N. Leveson, "An Empirical Study of Failure Probabilities in Multi-Version Software", Digest of Papers FTCS-16: *The 16th Annual International Symposium on Fault Tolerant Computing*, Vienna, Austria, June 1986.
[29] Siewiorek D.P., and Swarz R.S., "The Theory and Practice of Reliable System Design", Page 18, Digital Press, 1982.
[30] Hills, A.D., and Mirza, N.A., "Fault Tolerant Avionics", *Proceedings of the 8th AIAA/IEEE Digital Avionics Systems Conference* , San Jose, CA, October, 1988.
[31] Dzwonczyk, M.J., and Stone, H., "A Fault-Tolerant Avionics Suite For An Entry Research Vehicle", *Proceedings of the 8th AIAA/IEEE Digital Avionics Systems Conference* , San Jose, CA, October, 1988.

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No. NASA CR-181759	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle Study of a Unified Hardware and Software Fault-Tolerant Architecture		5. Report Date January 1989
		6. Performing Organization Code
7. Author(s) Jaynarayan Lala, Linda Alger, Steven Friend, Gregory Greeley, Stephen Sacco, and Stuart Adams		8. Performing Organization Report No.
		10. Work Unit No. 506-46-21-05
9. Performing Organization Name and Address The Charles Stark Draper Laboratory 555 Technology Square Cambridge, Massachusetts 02139		11. Contract or Grant No. NAS1-18061
		13. Type of Report and Period Covered Contractor Report
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Langley Research Center Hampton, Virginia 23665-5225		14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Monitor: Sally Johnson
Final Report

16. Abstract

A unified architectural concept, called the Fault Tolerant Processor Attached Processor (FTP-AP), that can tolerate hardware as well as software faults is proposed for applications requiring ultrareliable computation capability. An emulation of the FTP-AP architecture, consisting of a breadboard Motorola 68010-based quadruply redundant Fault Tolerant Processor, four VAX 750s as attached processors, and four versions of a transport aircraft yaw damper control law, is used as a testbed in the AIRLAB to examine a number of critical issues. Solutions of several basic problems associated with N-Version software are proposed and implemented on the testbed. This includes a confidence voter to resolve coincident errors in N-Version software. A reliability model of N-Version software that is based upon the recent understanding of software failure mechanism is also developed. The basic FTP-AP architectural concept appears suitable for hosting N-Version application software while at the same time tolerating hardware failures. Architectural enhancements for greater efficiency, software reliability modeling, and N-Version issues that merit further research are identified.

17. Key Words (Suggested by Author(s)) Fault Tolerant Processor S/W Reliability Model Attached Processor Ultrareliability N-Version Software Confidence Voter Isolation Algorithm		18. Distribution Statement Unclassified - Unlimited Subject Category 62	
19. Security Classif. (of this report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No. of pages 70	22. Price

NASA FORM 1626 OCT 86