View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by NASA Technical Reports Server

N89-16339

AN ADA IMPLEMENTATION FOR FAULT DETECTION, ISOLATION AND RECONFIGURATION USING A FAULT-TOLERANT PROCESSOR

Gregory L. Greeley

The Charles Stark Draper Laboratory
555 Technology Square
Cambridge, Massachusetts, 02139 USA
(617) 258-2482

Abstract

This paper covers the design and implementation, in Ada, of the Fault Detection, Isolation and Reconfiguration (FDIR) Manager for the triply redundant, tightly synchronized, Fault Tolerant Processor (FTP). It also examines the suitability of Ada, in the context of the FTP, for real time control tasks. This paper explains the operational concepts behind the FTP, and discusses the structure of the resultant Ada code.

This work is supported by NASA under JSC contract NAS9-17560.

Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

1. Draper Laboratory's Fault Tolerant Processor

1.1 Background

In April of 1983, the Charles Stark Draper Laboratory undertook the design and construction of a "distributed, fault and damage tolerant, real time information processing system" for aerospace vehicle control [2] [1]. This proof-of-concept system is known as the Advanced Integrated Processing System (AIPS). The goal of the AIPS project is to make a fault tolerant network of fault tolerant computers behave as a single highly reliable system. The AIPS system is composed of several Fault Tolerant Processors that are linked together via two networks: an inter-computer (IC) network, and an input-output (I/O) network.

The inter-computer network is used for communication among the FTP's. This network allows the FTP's to coordinate their actions and the division of tasks. The IC network is also used to report errors and failure conditions. The I/O network carries all input to and output from the AIPS FTP's. Thus, all sensors and actuators may be accessed by any FTP, and since FTP's are not tied to specific I/O devices, any FTP may run any I/O dependent task. This flexibility was built into AIPS so that tasks can "migrate" between FTP's without concern about which specific I/O devices are attached to the individual FTP's.

This paper concentrates on failure detection in the local FTP's, and further discussion on the operation of these two networks is beyond the scope of this paper.

1.2 The Fault Tolerant Processor

The AIPS Fault Tolerant Processor achieves a high level of reliability by using three identical processing elements that perform identical operations on identical input. The FTP will continue to operate correctly even after the failure of one of its channels, because data from the two good channels will vote out and mask data from the faulty one. The design goal of the FTP is to produce a fault tolerant virtual processor out of these three tightly synchronized channels. Thus, the programmer who writes applications for the FTP does not have to worry about the fact that there are actually three processing units that are continually voting all input and output. In the Draper Fault Tolerant Processor, specialized hardware maintains synchronization and handles communication between processing sites. This solution not only reduces the software overhead, but, in fact, allows the FTP to be treated as a virtual processor. Because none of the instructions in the user's application software reveal the fact that the FTP is actually three processing units, it is hoped that this virtual processor abstraction will reduce software cost and complexity in fault tolerant systems.

Data exchanges, which are necessary both for communicating with the other channels and for voting, are done by the hardware data exchange mechanism. Data is voted on a bit by bit basis: the hardware compares each set of three bits and masks out any bit that disagrees with the other two. If an error is detected, a hardware error latch is set, noting the type of exchange and the channel(s) at fault. Fault detection is implemented by comparing the voters' inputs and outputs; fault isolation uses the pattern of errors latched by the voters. By supplying this fault detection and masking in hardware, the FTP frees the software of this burden and helps provide the virtual processor abstraction. These concepts of hardware implemented fault tolerance and data exchanges have been successfully demonstrated in the Fault Tolerant Multi-Processor [4] at Draper Laboratory, and the theoretical basis for this interconnection scheme's protection against Byzantine failures can be found in [7].

ORIGINAL PAGE IS
OF POOR QUALITY

1.3 Data Exchange Mechanism

The data exchange mechanism is the FTP's primary means of correcting for failures. It has been shown [6] that Triple Module Redundant (TMR) systems such as the FTP need two basic types of data exchanges: a triplication and a direct vote. A triplication is used in the case where a single channel has a local value, such as a sensor reading or keyboard input, that must be sent to the other two channels. Since a direct transmission's reliability is vulnerable to a single point failure, the triplications are sent through the voters. A direct vote, on the other hand, is used in the case where three channels have computed identical outputs, such as actuator commands or terminal output, that must be voted to correct for errors before transmission.

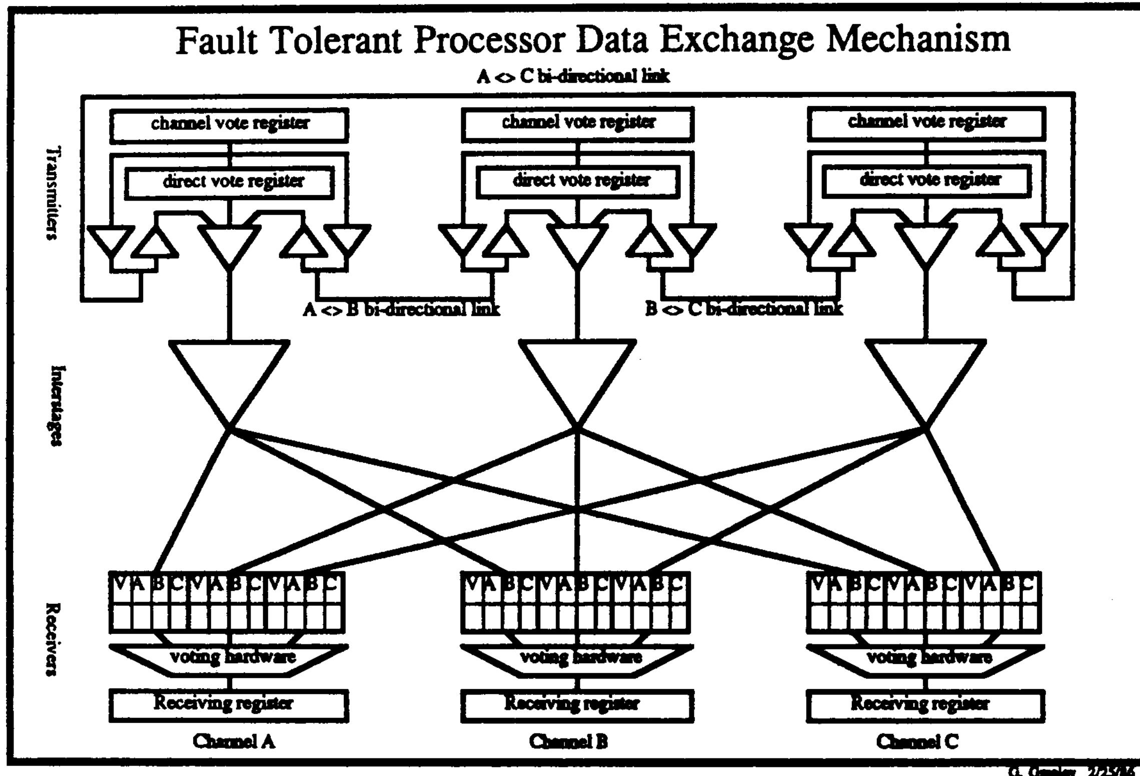

Figure 1-1: FTP Data Exchange Mechanism

Figure 1-1 shows a schematic representation of the FTP's data exchange mechanism. Note that there are three major elements in the mechanism: the transmitters, the interstages, and the receivers. These elements are connected in several different ways. First, each channel's transmitter has a bi-directional link to the other two channel's transmitters. These links are used for immediate access to raw data during triplication data exchanges. Second, each transmitter has a link to its interstage. This link is used to send data to be latched by the interstage for further re-transmission. Finally, each interstage has a link to each channel's receiver. These links are used by the interstages to send a copy of their data to each channel.

During an exchange, each of the elements in the data exchange mechanism has a different function. The transmitters must configure their data paths so that the correct data is sent to the interstages. Each transmitter may send either its own data or the data available on one of the direct links from the other channels. The interstages must latch the data, triplicate it, and send a separate copy to each of the three receivers. Finally, the receivers are responsible for latching and voting the three copies of the data from the interstages. The bit by bit majority vote is done in

hardware, and the result will be stored in the receiver register. If there are any disagreements in the voting, they are recorded in the voter's error latches.

Each channel's receiver has a 12-bit dedicated error latch. These twelve bits are divided into three sets of four bits. Each set is used to record errors from a specific channel, and each bit within a set is used to specify what type of exchange the error occurred in: direct vote or triplicating from A, from B, or from C. Thus, if channel A's voter discovers a disagreement in channel B's value while triplicating a value from C, it will set a specific bit for that exchange in its error latch. As more errors are discovered, more bits will be set, but none will be reset. Only a specific command from the software can reset the bits in the error latches.

In their well-known paper on the Byzantine General's problem [5] Lamport *et al* show that three processors (meaning three fault containment regions) cannot reach agreement in the presence of a fault. To surmount this problem, the FTP is divided into six fault containment regions: the three channels and the three interstages. That is, each channel and interstage is isolated (physically and electrically) so that a fault in one cannot cause a fault in another. This fault containment guarantees that a single fault in the FTP cannot prevent the three channels from reaching an agreement on the result of a vote. Thus, a channel or interstage may transmit bad data due to a single fault, but the bad data will be masked out by the rest of the system, which is fault free and generating correct data.

1.4 Use of the Data Exchange

A typical use for the data exchange mechanism would be a space craft control system reading a sensor. For complete fault coverage, three sensors would be used to read the same data, and each sensor would connect, through the I/O network, to a specific channel. Each channel would read a sensor and store a local value. Then, one by one, the channels would triplicate their local data by exchanging it with the other channels.

Figure 1-2 shows an example of channel A triplicating its local value via the data exchange. Note that channel A sends its local value directly to channels B and C, which route the data to their interstages. Then, all three channels initiate a vote on the raw data. The result of this vote is used by the three channels as channel A's value. This same procedure is then repeated for channels B and C. This exchange process ensures that, even in the presence of a failure, all three channels have an identical (although not necessarily correct) value for each channel's sensor reading. Thus, when this process is finished, each channel has three values that are identical to the three values that the other two channels have. The code that initiates these exchanges would be located in a library of I/O subroutines. This library is used to hide the data exchange mechanism from the user's application, preventing the user software from violating the abstraction of the FTP as a single processor. The following is an example of the code that performs a data exchange. Note that this code is executed at the same time by all three processors, giving each channel an A_value, a B_value, a C_value, and a local_value.

```
local_value := read_sensor;

A_value := exchange(from_a, local_value);

B_value := exchange(from_b, local_value);

C_value := exchange(from_c, local_value);
```

ORIGINAL PAGE IS
OF POOR QUALITY

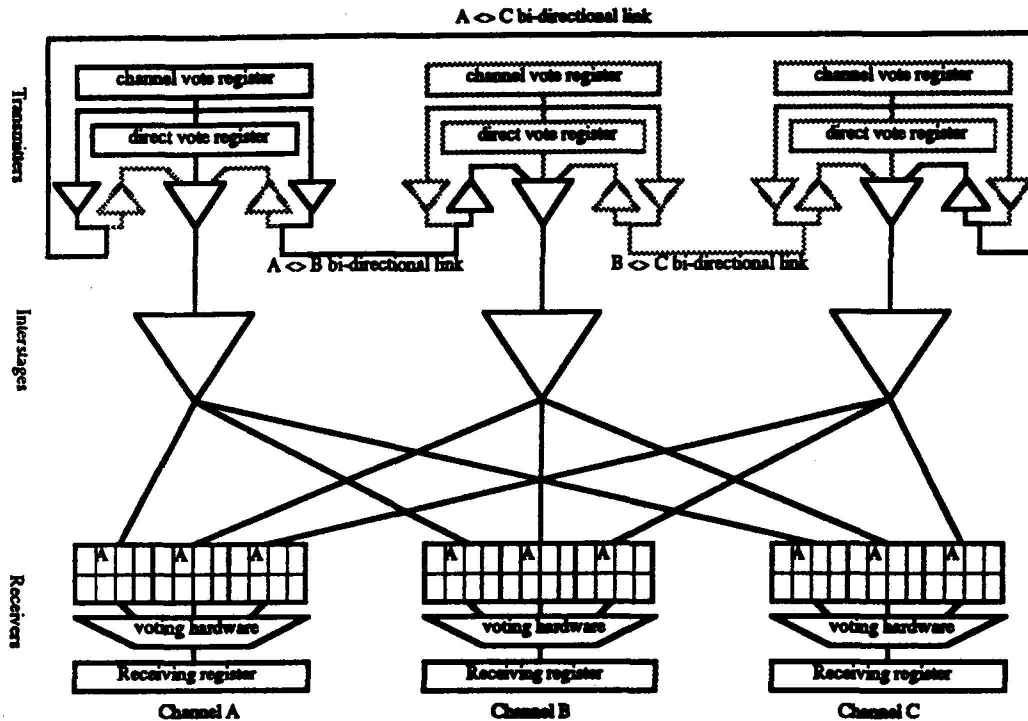

Figure 1-2: A data exchange from channel A

After all three channels have the three sensor readings, some type of redundancy algorithm (e.g., mid-value select) can be applied to these values to form a suitable result for the sensor reading. This "correct" value for the sensor reading is then used to produce an actuator command for maneuvering the space craft. Figure 1-3 shows the direct vote of this actuator command. Each channel directly sends its value to its interstage. The interstages then triplicate the data and send it to the receivers, which vote the results as before, noting any errors. Again, this whole process would be hidden from the user's application by a call to the FTP's I/O subroutine library. The output subroutine is also fairly simple:

```
voted_command := exchange(from_all, local_actuator_command);
send_command (voted_command);
```

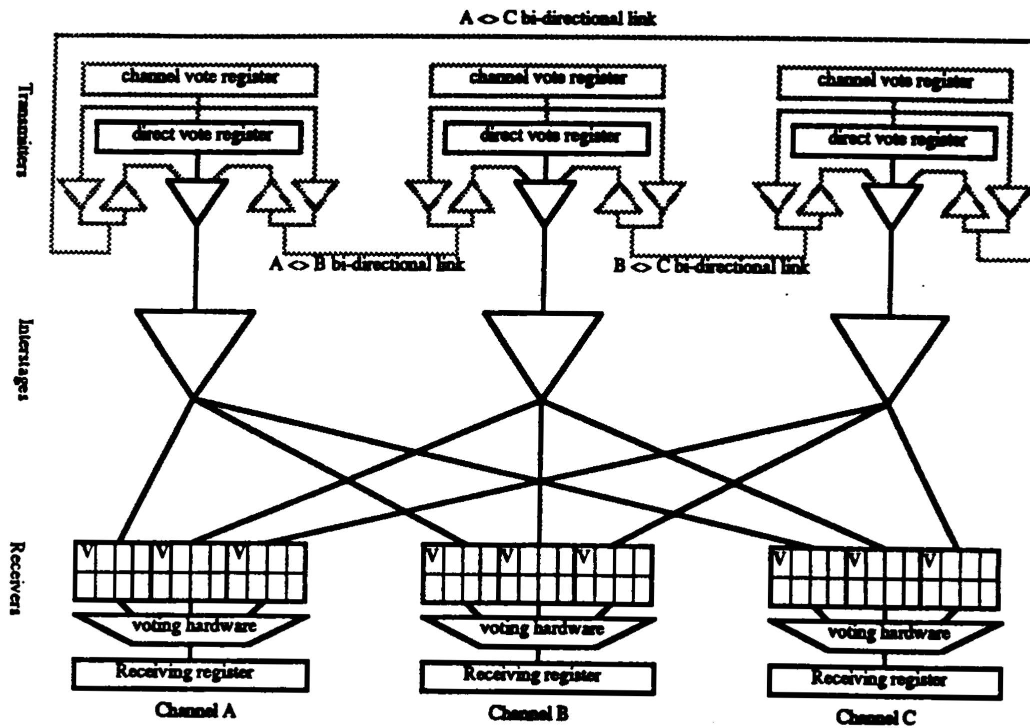

Figure 1-3: Direct vote of actuator command

ORIGINAL PAGE IS
OF POOR QUALITY

2. Structure of the FDIR code

This chapter discusses the design of the FDIR software. The design of the FDIR code was shaped by two main goals: provide complete fault coverage and use minimal processor overhead. FDIR must be able to locate and isolate any fault that occurs, and this must be done while using less that 5% of the processor's capacity. As a result of these design goals, the FDIR code is split into several tasks. The fast task can be run frequently, while the more complex tasks are run only on demand or at a lower frequency. This division allows for complete fault coverage while reducing the amount of processor time used.

In terms of software engineering, the design goals were to create FDIR code that is modular and readable. Ada helps these goals with its data abstractions and its packages, which are the advantages often cited when discussing the merits of the language [3]. Ada's use of data abstractions helps produce readable code by allowing programmers to manipulate data in a conceptual manner rather than a manner specified by the machine's representation of the data. Ada also helps produce modular code by encapsulating programs in constructs called packages which introduce these data abstractions. As a result, the FDIR code for the AIPS FTP has turned out much more modular and readable than the FDIR code that was written for a previous FTP using C.

The packages that comprise the FDIR software can be divided into four major categories:

1. Declarations
2. Resources
3. Extensions
4. Applications

2.1 Declarations

Declaration packages are collections of namings and constants that are used in many sections of the FDIR software (as well as the rest of the operating system). The only example of this type is *Memory*, the package that contains the mappings of all the special memory locations. *Memory* defines the locations for the data exchange hardware, the shared memory objects, and all the other hardware that is memory mapped, such as the timers and the Monitor Interlock.

2.2 Resources

Resource packages contain data types and operations that have general utility. For example, all the necessary procedures and types for using the data exchange and voting mechanism are defined in the *Exchange* package. Any software that is run on the FTP will need to vote input and output. The *Exchange* package encapsulates the data exchange hardware with a software abstraction so that all other software uses the voting mechanism without relying upon any implementation details. This means that if the data exchange hardware changes, only *one* package has to be changed to reflect the differences. Another resource package is the *Error_latch* package, which defines a data type for the error latches as well as the operations necessary to convert their hardware representation into a software defined Ada data type. Again, only one package relies upon the actual implementation of the hardware error latches, and only one package would have to be changed if the error latches were changed. The two remaining resource packages are *Config* and *Transient*. Both of these packages provide procedures, types, and

variables that monitor the state and "health" of the FTP's three channels. The *Config* package's primary responsibility is to maintain the software record of the three channels' status: present or lost. *Transient*, on the other hand, is primarily responsible for maintaining an unreliability index for each channel. Of all these resource packages, only *Exchange* would have a system-wide utility. The other (non-FDIR) parts of the operating system, however, do require access to things such as the error latches and the current configuration of the channels.

2.3 Extensions

Extension packages are used to actually extend the Ada language. Certain operations (such as the bit wise AND of two integers) are either not permitted or difficult to implement in Ada. Extension packages, which are series of assembly language subroutines that masquerade as Ada packages, add this needed functionality to the standard language operations. The package *Memory_utilities*, for example, was created so bit wise AND and OR operations could be performed on two integers. Although Ada can actually do AND and OR operations on arrays of boolean variables, the Telesoft compiler that produces the FTP code cannot pack a 16 boolean array into a single word.

The second extension package, called *Sync_utilities*, was created for synchronizing code execution among the channels. Synchronization requires absolute control over the timing of each machine instruction. Assembly language code *had* to be used for the critical part of the synchronization procedure to meet these strict timing requirements. The *Sync_utilities* package also provides the procedure that aligns memory. The memory align could have been done in Ada, but the time penalty for not using highly optimized assembly language code to align all of memory was too great for the FTP, which is designed to be a real time system.

2.4 Applications

The FDIR application packages use the resource, declaration, and the extension packages to actually "do something." These application packages do not define any new types. Instead, they import types and low level procedures from the three other kinds of packages. In general, the FDIR application packages have only a few visible procedures, which are mostly linear code. The three application packages that make up the FDIR manager are: *FDIR*, which detects and isolates all faults; *Sync*, which synchronizes the code execution initially and whenever a channel is lost; and *Test*, which constantly runs self test on the FTP hardware.

The *FDIR* package contains the actual code for the local FDIR manager. It has only one visible procedure, *Init*, which schedules a FDIR task to be run at a relatively high frequency (approximately 16 Hz, or every 60 msec). This task, called *Fast_FDIR*, is used to spot the occurrence of errors and isolate only the most obvious faults. Both the channel presence and the interstage tests are simple enough to be run at this relatively high frequency. *Fast_FDIR* also checks all reports from other parts of the system. If there are any necessary reconfigurations, *Fast_FDIR* will do the reconfigurations in a prioritized order. This higher frequency of operation improves reliability of the FTP by reducing the amount of time an error goes undetected. In reducing this time, the window in which two errors could simultaneously occur is also reduced. A second error, if it occurred before the FTP could reconfigure around the first error, would lead to unpredictable results.

There are, however, the two competing goals for the FDIR manager: complete fault

coverage, which demands high frequency, and minimal use of processor time, which demands faster, less complex operations. Thus, the *Fast_FDIR* task cannot take the time to analyze all possible fault conditions when an error is detected; it only analyzes the most simple cases. If *Fast_FDIR* encounters an error condition that it cannot analyze, then a new task is started, called *Slow_FDIR*. *Slow_FDIR* is referred to as an "on demand" task. *Fast_FDIR* will schedule it *only* if there is an error that is too complex to analyze immediately. *Slow_FDIR* will then fully analyze the error and report back to *Fast_FDIR* which channel, if any, is at fault. This split in the fault detection duties allows the FDIR manager to run quickly and often, fulfilling both goals.

There are three visible parts to the *Sync* package: *Init*, a procedure which initially synchronizes the code execution between the three channels; *Lost_soul*, a procedure which is continually run by a lone lost channel; and *Lost_soul_sync*, a task which a pair of synchronized channels will schedule (at a fairly low frequency) to find the third lost channel. These three pieces of code do exactly the same thing: send "lost soul" data patterns through the data exchange mechanism and wait for the electronic "echo" that indicates another channel was attempting to exchange at the same time. All three, in fact, use the same assembly language subroutine for the hardware interface.

The primary difference between these three operations is what they do once a channel is synchronized, when they run, and how often they run. *Init* runs only upon system initialization, and assumes that all channels are unsynchronized at the start. After two or more channels are synchronized, *Init* will reconfigure the internal FDIR records to match the new state of the hardware. Where *Init* is linear code that is run only once, *Lost_soul* is a tight loop with only one exit condition: synchronization. Any lone channel that needs to synchronize will run *Lost_soul*, and nothing else, until it resynchronizes with the other two channels. *Lost_soul* is run frequently so that whenever the other two channels find time to try to pick up the lone processor, the lone processor is waiting and ready to be picked up. The *Lost_soul_sync* task, on the other hand, is a shell that calls the *Lost_soul* procedure. The difference with the task is that the *two* channels (in synchronization) will call *Lost_soul* at a lower frequency. Also, when two or more channels execute *Lost_soul*, they only go once through the loop and exit. Thus, the *Lost_soul_sync* task can be scheduled to run at a low frequency and will only take a small amount of time to execute.

The third application package is *Test*, which contains the four FDIR self tests: voter and error latch, which verifies the voting mechanism; ROM sum, which checks the integrity of the FTP's ROM; RAM pattern, which tests the functionality of each RAM location; and RAM scrub, which ensures that all three channels have identical values in RAM. *Test*, like the *FDIR* package, has only one visible procedure, *System_test*. *System_test* calls each individual test in the appropriate order. Thus, the voters are tested before any memory values are exchanged, and the memory hardware is tested by the RAM pattern test before the memory contents are checked by the RAM scrub test. If any one of the four tests reports that there is a faulty channel, then *System_test* will stop and notify *Fast_FDIR* that a reconfiguration is required. *System_test* is called by a task in the *FDIR* package called *Selftest*. *Selftest* is scheduled to run at a low priority. Thus, if the processor has any free time, it will run some self tests.

3. The Suitability of Ada for the FTP

While developing the fault detection code for the Fault-Tolerant Processor, both the advantages and the disadvantages of using Ada were apparent. In general, the advantages of Ada, which are mostly due to the language specification, outweigh the disadvantages, which are mostly due to the compiler used for this project. This chapter discusses both the advantages and disadvantages of using Ada for the FTP, and why using Ada was, in the long run, a wise choice.

The choice of Ada as the development language was a controversial decision. Previous work on fault-tolerant processors at the Laboratory had been done in the C language, and using C would have saved the many man hours spent re-creating code that had already been written. Using C would have also meant that the software engineers would have had a familiar set of tools available to use (e.g., compilers, debuggers, etc.). But, there are two major reasons that led to the selection of Ada as the development language for the AIPS system. The first is the Department of Defense's requirement that Ada be used for military software. The second reason is Ada's tasking, exception handling, strong typing system, and enforced modularity that are widely touted in some circles [3]. The combination of these reasons led the original design team to specify that Ada would be used for the AIPS project. After almost a year of FDIR code development, the choice of Ada is still controversial.

3.1 Disadvantages

The main disadvantage of Ada is that it is an immature language. There are only a handful of fully validated compilers and few support tools for programmers. The compiler used for the AIPS FTP (the unpublished Telesoft Ada compiler version 1.5) has several specific shortcomings: the Run-Time System is inadequate for the FTP's requirements as a real-time system, the compiler produces inefficient code and is not a fully implemented version of Ada, and there are no debugging tools. Solving some of the problems associated with this system required a great deal of effort that would not have been expended if Ada were a more mature language.

The primary problem with the Telesoft Ada compiler is the Run-Time System's task scheduling mechanism. For a real-time control system such as the FTP, task scheduling is critical, and the first-in, first-out task queue supplied with the Telesoft system could not meet the strict timing requirements of a real-time system. Task priorities and interrupts are needed so that a minor task (such as a self test) would not prevent a critical task (such as *Fast_FDIR*) from running. After much work, Draper Labs developed a system of priorities and interrupts that were incorporated into the Telesoft Run-Time System. This new run-time system allows higher priority tasks to interrupt the operation of those with a lower priority and includes timing information that specifies the frequency at which a task should be scheduled. Unfortunately, the run-time system's size (approximately 48K bytes) is almost an order of magnitude larger than the operating system used for the C version of the FTP. Although the Telesoft Run-Time System code has more functionality than the C version's operating system, it is not clear that these features are needed for a real-time system. With this new run-time system, Ada's task scheduling could fulfill the FTP's requirements for real-time vehicle control.

Not only is the Telesoft Ada Run-Time System larger, but the size of the object code generated by the Telesoft Ada compiler was surprisingly large as well. In fact, the FTP system had to be redesigned to include one megabyte of RAM rather than the original 256K bytes, which would have been sufficient had this code been written in C. This increased code size has several sources: the immature compiler, which generates inefficient code, the code design, which can add

to the compiler's inefficient code generation, and the required Ada runtime overhead, such as range and exception checks. Better compilers will, of course, help this problem. However, Ada rarely produces code as efficient as C, just as C rarely produces code as efficient as assembly language. Fortunately, the FDIR code has not exceeded the original C language size by any large amount, and the *Fast_FDIR* task is still within the 5% processor capacity goal.

Because the Telesoft Ada compiler is not a fully implemented version of Ada, some coding problems must be resolved in awkward ways. For example, the representation for the error latches would logically be an array or record of boolean types. The Telesoft compiler, however, does not allow the representation of an Ada record or array to be specified on a bit-by-bit basis. Thus, when the data type for an error latch was defined, Telesoft Ada could not define a record that matched the 12-bit structure and location of the actual error latches. But, because the error latches had to be exchanged among the channels as 16-bit integers, a standard record or array could not be used either. Thus, the FDIR code used a function that converts the hardware error latches into patterns that fit a 16-bit integer. Unfortunately, this sacrifices one of the primary advantages of the Ada language: its ability to easily create data abstractions from built-in types. Other problems with the Telesoft compiler were along the same vein: problems that were irritating because hardware representations could not be mapped to data abstractions with the ease that Ada promised, and solutions that were difficult to use in Ada because they did not take advantage of the built-in types and functions.

Finally, the fourth problem with the Telesoft compiler is the total lack of debugging aids. In terms of debugging tools, a disassembler is absolutely required. Thus, the Laboratory had to produce, in house, a disassembler for the FTP's 68010 code. A VAX interface program, which implements standard debugging utilities (e.g., breakpoints, memory and register displays, and program downloads), was also produced in house. Unlike the C compiler that was previously used, the Ada compiler could not produce assembly language listings of the code that have the original Ada statements inserted in the appropriate places. This was a major drawback because all matches between the disassembled object code and the original source code had to be done manually. The lack of debugging aids requires that effort be diverted from software development to debugging tool development, which is not the purpose of this project.

3.2 Advantages

On the other hand, the advantages of Ada are due largely to the language definition rather than the specific compiler. The strong typing system, for instance, allows code written by several individuals to be linked together with almost no errors. Also, Ada's package system fosters a highly modular design that clearly delineates all module dependencies, while the data abstraction capability makes it easier to create readable code. Finally, although the run-time system was not adequate at first, the Ada built-in tasking construct is useful because the FTP needs multiprocessing capability.

Ada's rigid syntax and strong typing system, which are hated by some programmers, are responsible for reducing errors in software to the point that almost any program that compiles will run, and will have almost no errors. The syntax is responsible for reducing the number of typographical mistakes that are accepted by the compiler as legitimate code. The strong typing system, meanwhile, reduces the number of errors due to interfacing procedures and data abstractions. And, because the structure and syntax of Ada lets fewer errors slip past the compiler, Ada reduces the time spent debugging code.

Ada's data abstractions are a powerful force in making code that is readable and has a well defined interface. In C, for example, the configuration of the three channels (on- or off-line) was numerically represented as three bits in a 16-bit word. The representation of the data, as well as the operations performed on it, are not conceptually obvious. Ada, on the other hand, represents the configuration as a record of three booleans. Using booleans in a record to represent the configuration produces more readable code that parallels the actual structure of the information. This abstraction also reduces the mistakes and confusion between programmers who must interface code. In C, there was a convention that channel A was represented by the low order bit in a 16-bit integer. This convention, however, is not as obvious as a record with a boolean component named A. Again, Ada's data abstractions prevent these types of interfacing errors from occurring, and thereby cut the time required to debug software.

Overall, Ada is the right language for this project. The Ada language has several strong advantages, while most of the disadvantages are due to its immaturity and the specific compiler used. In time, the language will mature and more capable compilers will be available. However, even a poor version of Ada has already decreased the work required to create, debug, and interface the code on the FTP.

The decision to switch to Ada was controversial. Despite the advantages of Ada's tasking, data abstraction, and modularity, many engineers were concerned about Ada's immaturity and lack of debugging tools. Even more important, however, was the run time environment and its ability to meet the critical timing requirements of a real time control system. In spite of these problems, the development of the FDIR manager has shown that Ada has promise as a development language for embedded computer systems.

References

[1] Charles Stark Draper Laboratory.
Advanced Information Processing System (AIPS) System Specification.
Technical Report CSDL-C-5709, Charles Stark Draper Laboratory, Inc., Cambridge, Massachusetts, May, 1984.

[2] Alger, Linda, *et al.*
Local Fault Detection, Isolation, and Reconfiguration in a Distributed Processing System.
December, 1985.

[3] Barnes, J. G. P.
Programming in Ada.
Addison-Wesley Publishing Co., 1982.

[4] Hopkins, Albert L., Smith, T. Basil, and Lala, Jaynarayan H.
FTMP -- A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft.
Proceedings of the IEEE 66(10):1221-1239, October, 1978.

[5] Lamport, Leslie, *et al.*
The Byzantine Generals Problem.
ACM Transactions on Programming Languages and Systems 4(3):382-401, July, 1982.

[6] Smith, T. Basil.
Generic Data Manipulative Primitives of Synchronous Fault-Tolerant Computer Systems.
Technical Report, Charles Stark Draper Laboratory, Inc., Cambridge, Massachusetts, 1980.

[7] Smith, T. Basil.
Fault-Tolerant Processor Concepts and Operation.
Technical Report, Charles Stark Draper Laboratory, Inc., Cambridge, Massachusetts, 1981.