SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control

JOHN H. WENSLEY, LESLIE LAMPORT, JACK GOLDBERG, SENIOR MEMBER, IEEE, MILTON W. GREEN, KARL N. LEVITT, P. M. MELLIAR-SMITH, ROBERT E. SHOSTAK, AND CHARLES B. WEINSTOCK

Abstract—SIFT (Software Implemented Fault Tolerance) is an ultrareliable computer for critical aircraft control applications that achieves fault tolerance by the replication of tasks among processing units. The main processing units are off-the-shelf minicomputers, with standard microcomputers serving as the interface to the I/O system. Fault isolation is achieved by using a specially designed redundant bus system to interconnect the processing units. Error detection and analysis and system reconfiguration are performed by software. Iterative tasks are redundantly executed, and the results of each iteration are voted upon before being used. Thus, any single failure in a processing unit or bus can be tolerated with triplication of tasks, and subsequent failures can be tolerated after reconfiguration. Independent execution by separate processors means that the processors need only be loosely synchronized, and a novel fault-tolerant synchronization method is described. The SIFT software is highly structured and is formally specified using the SRI-developed SPECIAL language. The correctness of SIFT is to be proved using a hierarchy of formal models. A Markov model is used both to analyze the reliability of the system and to serve as the formal requirement for the SIFT design. Axioms are given to characterize the high-level behavior of the system, from which a correctness statement has been proved. An engineering test version of SIFT is currently being built.

I. Introduction

THIS paper describes ongoing research whose goal is to build an ultrareliable fault-tolerant computer system named SIFT (Software Implemented Fault Tolerance). In this introduction, we describe the motivation for SIFT and provide some background for our work. The remainder of the paper describes the actual design of the SIFT system. Section II gives an overview of the system and describes the approach to fault tolerance used in SIFT. Sections III and IV describe the SIFT hardware and software, respectively. Section V discusses the proof of the correctness of SIFT.

A. Motivation

Modern commercial jet transports use computers to carry out many functions, such as navigation, stability augmentation, flight control, and system monitoring. Although these computers provide great benefits in the operation of the aircraft, they are not critical. If a computer fails, it is always possible for the aircrew to assume its function, or for the function to be abandoned. (This may require significant changes, such as diversion to an alternative destination.) NASA, in its Aircraft Energy Efficiency (ACEE) Program, is currently studying the design of new types of aircraft to reduce fuel consumption. Such aircraft will operate with greatly reduced stability margins, which means that the safety of the flight will depend upon active controls derived from computer outputs. Com puters for this application must have a reliability that i: comparable with other parts of the aircraft. The frequentl} quoted reliability requirement is that the probability o failure should be less than 10^{-9} per hour in a flight of te hours duration. A good review of the reliability requirement: associated with flight control computers appears in Murray *e al.* [1]. This reliability requirement is similar to that demande for manned space-flight systems.

A highly reliable computer system can have application in other areas as well. In the past, control systems in critica industrial applications have not relied solely on computer but have used a combination of human and computer contro With the need for faster control loops, and with the increase complexity of modern industrial processes, computer reliabilit has become extremely important. A highly reliable compute system developed for aircraft control can be used in suc applications as well. Our objective in designing SIFT is t achieve the reliability required by these applications in a economic manner. Moreover, we want the resulting syste to be as flexible as possible, so it can be easily adapted t changes in the problem specification.

When failure rates are extremely small, it is impossible t determine their values by testing. Therefore, testing cann be used to demonstrate that SIFT meets its reliability requir ments. It is necessary to *prove* the reliability of SIFT b mathematical methods. The need for such a proof of r liability has been a major influence on the design of SIFT.

B. Background

Our work on SIFT began with a study of the requiremen for computing in an advanced commercial transport aircra [2], [3]. We identified the computational and memory requi ments for such an application and the reliability requir for the safety of the aircraft. The basic concept of the SI system emerged from a study of computer architectu for meeting these requirements.

The second phase in the development of the SIFT syste which has just been completed, was the complete design the hardware and software systems [4], [5]. This design h been expressed formally by rigorous specifications that descri the functional intent of each part of the system. A ma influence during this phase was the Hierarchical Design Metho ology developed at SRI [10]. A further influence has been t need to use formal program proving techniques to ensure t correctness of the software design.

The current phase of the development calls for the build of an engineering model and the carrying out of tests

Manuscript received May 20, 1978. This work was supported by NASA-Langley Research Center under Contract NAS1-13792.

The authors are with SRI International, Menlo Park, CA 94025.

demonstrate its fault-tolerant behavior. The engineering model is intended to be capable of carrying out the calculations required for the control of an advanced commercial transport aircraft. SRI is responsible for the overall design, the software, and the testing, while the detailed design and construction of the hardware is being done by Bendix Corporation. The engineering model is scheduled to be built by the middle of 1979, with testing to be completed by the end of that year. Work is also continuing at SRI on proving the correctness of the system.

The study of fault-tolerant computing has in the past concentrated on failure modes of components, most of which are no longer relevant. The prior work on permanent "stuck-at-one" or "stuck-at-zero" faults on single lines is not appropriate for considering the possible failure modes of modern LSI circuit components, which can be very complex and affect the performance of units in very subtle ways. Our design approach makes no assumptions about failure modes. We distinguish only between failed and nonfailed units. Since our primary method for detecting errors is the corruption of data, the particular manner in which the data are corrupted is of no importance. This has important consequences for failure-modes-and-effects analysis (FMEA), which is only required at the interface between units. The rigorous, formal specification of interfaces enables us to deduce the effects on one unit of improper signals from a faulty unit.

Early work on fault-tolerant computer systems used fault detection and reconfiguration at the level of simple devices such as flip-flops and adders. Later work considered units such as registers or blocks of memory. With today's LSI units, it is no longer appropriate to be concerned with such small subunits. The unit of fault detection and of reconfiguration in SIFT is a processor/memory module or a bus.

Several low-level techniques for fault tolerance, such as error detection and correction codes in memory, are not included in the design of SIFT. Such techniques could be incorporated in SIFT, but would provide only a slight improvement in reliability.

II. SIFT Concept of Fault Tolerance

A. *System Overview*

As the name "Software Implemented Fault Tolerance" implies, the central concept of SIFT is that fault tolerance is accomplished as much as possible by programs rather than hardware. This includes error detection and correction, diagnosis, reconfiguration, and the prevention of a faulty unit from having an adverse effect on the system as a whole.

The structure of the SIFT hardware is shown in Fig. 1. Computing is carried out by the main processors. Each processor's results are stored in a main memory that is uniquely associated with the processor. A processor and its memory are connected by a conventional high bandwidth connection. The I/O processors and memories are structurally similar to the main processors and memories, but are of much smaller computational and memory capacity. They connect to the input and output units of the system which, for this application, are the sensors and actuators of the aircraft.

Each processor and its associated memory form a *processing module*, and each of the modules is connected to a multiple bus system. A faulty module or bus is prevented from causing faulty behavior in a nonfaulty module by the fault isolation methods described in Section II-B.

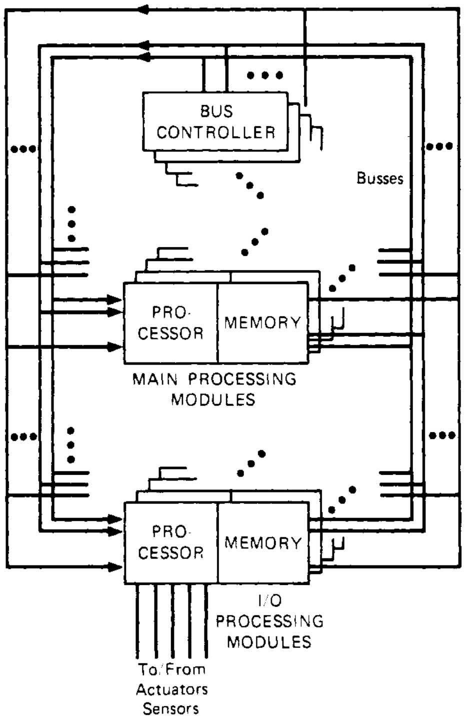

Fig. 1. Structure of the SIFT system.

The SIFT system executes a set of *tasks*, each of which consists of a sequence of *iterations*. The input data to each iteration of a task is the output data produced by the previous iteration of some collection of tasks (which may include the task itself). The input and output of the entire system is accomplished by tasks executed in the I/O processors. Reliability is achieved by having each iteration of a task independently executed by a number of modules. After executing the iteration, a processor places the iteration's output in the memory associated with the processor. A processor that uses the output of this iteration determines its value by examining the output generated by each processor which executed the iteration. Typically, the value is chosen by a "two out of three" vote. If all copies of the output are not identical, then an error has occurred. Such errors are recorded in the processor's memory, and these records are used by the executive system to determine which units are faulty.

SIFT uses the iterative nature of the tasks to economize on the amount of voting, by voting on the state data of the aircraft (or the computer system) only at the beginning of each iteration. This produces less data flow along the busses than with schemes that vote on the results of all calculations performed by the program. It also has important implications for the problem of synchronizing the different processors. We must ensure only that the different processors allocated to a task are executing the same iteration. This means that the processors need be only loosely synchronized (e.g., to within 50 μs), so we do not need tight synchronization to the instruction or clock interval.

An important benefit of this loose synchronization is that an iteration of a task can be scheduled for execution at slightly different times by different processors. Simultaneous transient failures of several processors will, therefore, be less likely to produce correlated failures in the replicated versions of a task.

The number of processors executing a task can vary with the task, and can be different for the same task at different times—e.g., if a task that is not critical at one time becomes critical at another time. The allocation of tasks to modules is in general different for each module. It is determined dynamically by a task called the global executive, which diagnoses errors to determine which modules and buses are faulty. When the global executive decides that a module has become faulty, it "reconfigures" the system by appropriately changing the allocation of tasks to modules. The global executive and its interaction with the individual processors is described in Section IV.

B. Fault Isolation

An important property required in all fault-tolerant computers is that of fault isolation: preventing a faulty unit from causing incorrect behavior in a nonfaulty unit. Fault isolation is a more general concept than damage isolation. Damage isolation means preventing physical damage from spreading beyond carefully prescribed boundaries. Techniques for damage isolation include physical barriers to prevent propagation of mechanical and thermal effects and electrical barriers—e.g., high-impedance electrical connections and optical couplers. In SIFT, such damage isolation is provided at the boundaries between processing modules and buses.

Fault isolation in SIFT requires not only damage isolation, but also preventing a faulty unit from causing incorrect behavior either by corrupting the data of the nonfaulty unit, or by providing invalid control signals. The control signals include those that request service, grant service, effect timing synchronization between units, etc.

Protection against the corruption of data is provided by the way in which units can communicate. A processing module can read data from any processing module's memory, but it can write only into its own memory. Thus a faulty processor can corrupt the data only in its own memory, and not in that of any other processing modules. All faults within a module are treated as if they have the same effect: namely that they produce bad data in that module's memory. The system does not attempt to distinguish the nature of a module fault. In particular, it does not distinguish between a faulty memory and a processor that puts bad data into an otherwise nonfaulty memory.

Note that a nonfaulty processor can obtain bad data if that data is read from a faulty processing module or over a faulty bus. Preventing these bad data from causing the generation of incorrect results is discussed below in the section on fault masking.

Fault isolation also requires that invalid control signals not produce incorrect behavior in a nonfaulty unit. In general, a faulty set of control signals can cause two types of faulty behavior in another unit.

1) The unit carries out the wrong action (possibly by doing nothing).

2) The unit does not provide service to other units.

In SIFT these two types of fault propagation are prevented by making each unit autonomous, with its own control. Improper control signals are ignored, and time-outs are used to prevent the unit from "hanging up" waiting for a signal that never arrives. The details of how this is done are discussed in Section III.

C. Fault Masking

Although a faulty unit cannot cause a nonfaulty processor to behave incorrectly, it can provide the processor with bad data. In order to completely mask the effects of the faulty unit, we must ensure that these bad data does not cause the processor to generate incorrect results. As we indicated above, this is accomplished by having the processor receive multiple copies of the data. Each copy is obtained from a different memory over a different bus, and the processor uses majority voting to obtain a correct version of the data. The most common case will be the one in which a processor obtains three copies of the data, providing protection from a single faulty unit.

After identifying the faulty unit, the system will be reconfigured to prevent that unit from having any further effect. If the faulty unit is a processing module, then the tasks that were assigned to it will be reassigned to other modules. If it is a bus, then processors will request their data over other buses. After reconfiguration, the system will be able to withstand a new failure—assuming that there are enough nonfaulty units remaining.

Because the number of processors executing a task can vary with the task and can be changed dynamically, SIFT has a flexibility not present in most fault tolerant systems. The particular application field—aircraft control—is one in which different computations are critical to different degrees, and the design takes advantage of this.

D. Scheduling

The aircraft control function places two types of timing requirements on the SIFT system.

1) Output to the actuators must be generated with specified frequency.

2) Transport delay—the delay between the reading of sensors and the generation of output to the actuators based upon those readings—must be kept below specified limits.

To fulfill these requirements, an iteration rate is specified for each task. The scheduling strategy must guarantee tha the processing of each iteration of the task will be complete within the "time frame" of that iteration. It does not matte when the processing is performed, provided that it is complete by the end of the frame. Moreover, the time needed to ex ecute an iteration of a task is highly predictable. The iteratio rates required by different tasks differ, but they can be ad justed somewhat to simplify the scheduling.

Four scheduling strategies were considered for SIFT:

1) fixed preplanned (nonpreemptive) scheduling;
2) priority scheduling;
3) deadline scheduling;
4) simply periodic scheduling.

Of these, fixed preplanned scheduling in which eac iteration is run to completion, traditional in-flight contr applications, was rejected because it does not allow sufficie flexibility.

The priority-scheduling strategy, commonly used in gener purpose systems, can meet the real-time requirements if t tasks with the fastest iteration rates are given the highe priorities. Under this condition, it is shown in [6] that tasks will be processed within their frames, for any pattern iteration rates and processing times—provided the processi load does not exceed ln(2) of the capacity of the process (up to about 70 percent loading is always safe).

The deadline-scheduling strategy always runs the task wh deadline is closest. It is shown in [6] that all the tasks will processed within their frames provided the workload d

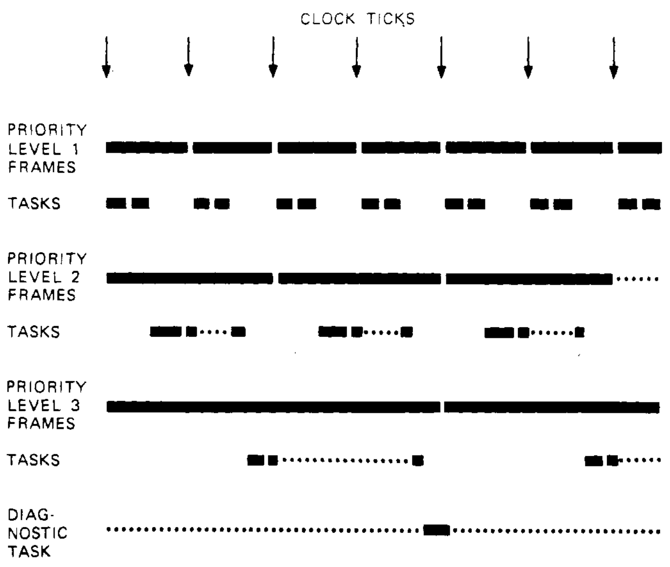

Fig. 2. A typical schedule.

not exceed the capacity of the processor (100 percent loading is permissible). Unfortunately, for the brief tasks characteristic of flight-control applications, the scheduling overhead eliminates the advantages of this strategy.

The simply periodic strategy is similar to the priority strategy, but the iteration rates of the tasks are constrained so that each iteration rate is an integral multiple of the next smaller rate (and thus of all smaller rates). To comply with this requirement, it may be necessary to run some tasks more frequently than their optimum rate, but this is permissible in a flight control system. It is shown in [6] that if the workload does not exceed the capacity of the processor (100 percent loading is possible), then simply periodic scheduling guarantees that all tasks will complete within their frames.

The scheduling strategy chosen for the SIFT system is a slight variant of the simply periodic method, illustrated by Fig. 2. Each task is assigned to one of several priority levels. Each priority level corresponds to an iteration rate, and each iteration rate is an integral multiple of the next lower one. In order to provide very small transport delays for certain functions, and to allow rapid detection of any fault which causes a task not to terminate, the scheme illustrated in Fig. 2 is modified as follows. The time frame corresponding to highest priority level (typically 20 ms) is divided into a number of subframes (typically 2 ms). The highest priority tasks are run in specific subframes, so that their results can be available to other tasks run in the next subframe, and they are required to complete within one subframe.

C. *Processor Synchronization*

The SIFT intertask and interprocessor communication mechanism allows a degree of asynchronism between processors and avoids the lockstep traditional in ultrareliable systems. Up to 50 μs of skew between processors can readily be accommodated, but even this margin cannot be assured over a ten-hour period with free-running clocks unless unreasonable requirements are imposed on the clocks. Thus, the processors must periodically resynchronize their clocks to ensure that no clock drifts too far from any other.

For reliability, the resynchronization procedure must be immune to the failure of any one clock or processor, and to a succession of failures over a period of time. In order to guarantee the high reliability required of SIFT, we cannot allow a system failure to be caused by any condition whose probability cannot be quantified, regardless of how implausible that condition may seem. This means that our synchronization procedure must be reliable in the face of the worst possible behavior of the failing component, even though that behavior may seem unrealistically malicious. We can only exclude behavior which we can *prove* to be sufficiently improbable.

The traditional clock resynchronization algorithm for reliable systems is the median clock algorithm, requiring at least three clocks. In this algorithm, each clock observes every other clock and sets itself to the median of the values that it sees. The justification for this algorithm is that, in the presence of only a single fault, the median value must either be the value of one of the valid clocks or else it must lie between a pair of valid clock values. In either case, the median is an acceptable value for resynchronization. The weakness of this argument is that the worst possible failure modes of the clock may cause other clocks to observe different values for the failing clock. Even if the clock is read by sensing the time of a pulse waveform, the effects of a highly degraded output pulse and the inevitable slight differences between detectors can result in detection of the pulse at different times.

In the presence of a fault that results in other clocks seeing different values for the failing clock, the median resynchronization algorithm can lead to a system failure. Consider a system of three clocks A, B, and C, of which C is faulty. Clock A runs slightly faster than clock B. The failure mode of clock C is such that clock A sees a value for clock C that is slightly earlier than its own value, while clock B sees a value for clock C that is slightly later than its own value. Clocks A and B both correctly observe that the value of clock A is earlier than the value of clock B. In this situation, clocks A and B will both see their own value as the median value, and therefore not change it. Both the good clocks A and B are therefore resynchronizing onto themselves, and they will slowly drift apart until the system fails.

It might be hoped that some relatively minor modification to the median algorithm could eliminate the possibility of such system failure modes. However, such hope is groundless. The type of behavior exhibited by clock C above will doom to failure any attempt to devise a reliable clock resynchronization algorithm for only three clocks. It can be proved that, if the failure-mode behavior is permitted to be arbitrary, then there cannot exist any reliable clock resynchronization algorithm for three clocks. The impossibility of obtaining exact synchronization with three clocks is proved in [9]. The impossibility of obtaining even the approximate synchronization needed by SIFT has also been proved, but the proof is too complex to present here and will appear in a future paper. The result is quite general and applies not only to clocks, but to any type of integrator which is subject to minor perturbations as, for example, inertial navigation systems.

Although no algorithm exists for three clocks, we have devised an algorithm for four or more clocks which makes the system immune to the failure of a single clock. The algorithm has been generalized to allow the simultaneous failure of M out of N clocks when $N > 3M$. Here, we only describe the single-failure algorithm, without proving it correct. (Algorithms of this type often contain very subtle errors, and extremely rigorous proofs are needed to ensure their correctness.) The general algorithm, and the proof of its correctness, can be found in [9].

The algorithm is carried out in two parts. In the first part, each clock[1] computes a vector of clock values, called the *interactive consistency vector*, having an entry for every clock. In the second part, each clock uses the interactive consistency vector to compute its new value.

A clock p computes its interactive consistency vector as follows. The entry of the vector corresponding to p itself is set equal to p's own clock value. The value for the entry corresponding to another processor q is obtained by p as follows.

1) Read q's value from q.

2) Obtain from each other clock r the value of q that r read from q.

3) If a majority of these values agree, then the majority value is used. Otherwise, the default value NIL (indicating that q is faulty) is used.

One can show that if at most one of the clocks is faulty, then: 1) each nonfaulty clock computes exactly the same interactive consistency vector; and 2) the component of this vector corresponding to any nonfaulty clock q is q's actual value.

Having computed the interactive consistency vector, each clock computes its new value as follows. Let δ be the maximum amount by which the values of nonfaulty processors may disagree. (The value of δ is known in advance, and depends upon the synchronization interval and the rate of clock drift.) Any component that is not within δ of at least two other components is ignored, and any NIL component is ignored. The clock then takes the median value of the remaining components as its new value.

Since each nonfaulty clock computes exactly the same interactive consistency vector, each will compute exactly the same median value. Moreover, this value must be within δ of the original value of each nonfaulty clock.

This is the basic algorithm that the SIFT processors will use to synchronize their clocks. Each SIFT processor reads the value of its own clock directly, and reads the value of another processor's clock over a bus. It obtains the value that processor r read for processor q's clock by reading from processor r's memory over a bus.

F. Reliability Prediction

A sufficiently catastrophic sequence of component failures will cause any system to fail. The SIFT system is designed to be immune to certain likely sequences of failures. To guarantee that SIFT meets its reliability goals, we must show that the probability of a more catastrophic sequence of failures is sufficiently small.

The reliability goal of the SIFT system is to achieve a high probability of survival for a short period of time—e.g., a ten-hour flight—rather than a large mean time before failure (MTBF). For a flight of duration T, survival will occur unless certain combinations of failure events occur within the interval T or have already occurred prior to the interval T and were undetected by the initial checkout of the system. Operationally, failures of the latter type are indistinguishable from faults that occur during the interval T.

To estimate the probability of system failure we use a finite-state Markov-like *reliability model* in which the state transitions are caused by the events of fault occurrence, fault detection, and fault "handling". The combined probability of all event sequences that lead to a failed state is the system failure probability. A design goal for SIFT is to achieve a failure rate of 10^{-9} per hour for a ten hour period.

For the reliability model, we assume that hardware fault events and electrical transient fault events are uncorrelated and exponentially distributed in time (constant failure rates). These assumptions are believed to be accurate for hardware faults because the physical design of the system prevents fault propagation between functional units (processors and buses) and because a multiple fault within one functional unit is no more serious than a single fault. The model assumes that all failures are permanent (for the duration of the flight), so it does not consider transient errors. The effects of uncorrelated transient errors are masked by the executive system, which requires a unit to make multiple errors before it considers the unit to be faulty. It is believed that careful electrical design can prevent correlation of transient errors between functional units. The execution of critical tasks in "loose" synchronism also helps protect against correlation of fast transient errors. Failure rates for hardware have been estimated on the basis of active component counts, using typical reliability figures for similar hardware. For the main processors, we obtain the rate 10^{-4} per hour; for I/O processors and buses, we obtain 10^{-5} per hour.

For a SIFT system with about the same number of main processing modules, I/O processing modules, and buses, it can be shown that the large difference in failure rates between a main processing module and an I/O processing modules or bus implies that we need only consider main processing module failures in our calculations. We can therefore let the state of the system be represented in the reliability model as a triple of integers (h, d, f) with $h \leqslant d \leqslant f$, where such a state represents a situation in which f failures of individual processors have occurred, d of those failures have been detected, and h of these detected failures have been "handled" by reconfiguration. There are three types of possible state transition.

1) $(h, d, f) \rightarrow (h, d, f+1)$, representing the failure of a processor.

2) $(h, d, f) \rightarrow (h, d+1, f), d < f$, representing the detection of a failure.

3) $(h, d, f) \rightarrow (h+1, d, f), h < d$, representing the handling of a detected failure.

This is illustrated in Fig. 3.

The first two types of transition—processor failure and failure detection, represented in Fig. 3 by straight arrows—are assumed to have constant probabilities per unit time However, the third type of transition—failure handling, represented in Fig. 3 by wavey arrows—represents the completion of a reallocation procedure. We assume that this transitior must occur within some fixed length of time τ.

A state (h, d, f) with $h < d$ represents a situation in whicl the system is reconfiguring. To make the system immun to an additional failure while in this state is a difficult problem since it means that the procedure to reconfigure around failure must work despite an additional, undetected failur Rather than assuming that this problem could be solved, w took the approach of trying to insure that the time τ that th system remains in such a state is small enough to make i highly unlikely for an additional failure to occur befor reconfiguration is completed. We therefore made the pe simistic assumption that a processor failure which occu

[1] In the following discussion, a clock is assumed to be capable of logical operations. In SIFT, such a clock is actually a processor and its internal clock.

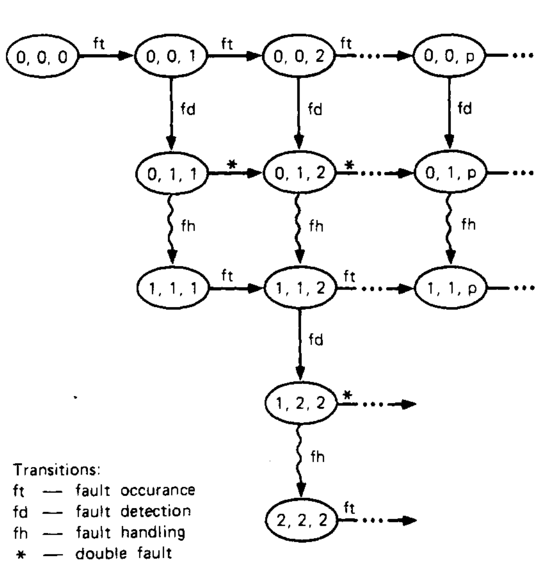

Fig. 3. The reliability model.

TABLE I

(T = 10 hours)

FAILURE CAUSE	FAILURE PROBABILITY
Exhaustion of spares	5×10^{-12}
Double Fault (τ = 100 ms.)	7×10^{-11}
Double Fault (τ = 1 sec.)	7×10^{-10}

while the system is reconfiguring will cause a system failure. Such failures are represented by the "double-fault" transitions indicated by asterisks in Fig. 3. In our calculations, we assume that each of these transitions results in a system failure.

We have calculated the probability of system failure through a double fault transition, and also through reaching a state with fewer than two nonfaulty processors, for which we say that the system has failed because it has "run out of spares."[2] A brief summary of these failure probabilities for a five processor system is shown in Table I.

III. The SIFT Hardware

The SIFT system attempts to use standard units whenever possible. Special design is needed only in the bus system and in the interfaces between the buses and the processing modules.

The major parameters of the SIFT system are shown in Table II. The column heading "Engineering Model" indicates the system intended for initial construction, integration, and testing. The column heading "Maximum" indicates the limits to which the engineering model can be expanded with only the procurement of additional equipment.

As described in Section II, the fault-tolerant properties of SIFT are based on the interconnection system between units and upon the software system. The particular design of the processors and memories is irrelevant to our discussion of fault tolerance. We merely mention that the main processors and memories are based on the BD*micro*X computer—a modern, LSI-based 16-bit computer designed and manu-

[2] The probability of system failure because of multiple *undetected* faults has not been computed precisely, but is expected to be comparable to the double fault values.

TABLE II

Major Parameters of the SIFT System, Engineering Model

System Parameters	Engineering Model	Maximum
Main Processors	5*	8
Main Memories	5	8
I/O Processors	5	8
I/O Memories	5	9
Busses	5	8
External Interfaces	5	8
Main Processors		
Word Length	16 bits	Same
Addressing Capability	32K words	64K
Speed	500K IPS	Same
Arithmetic Modes	Fixed point Double length Floating point	Same
Type	Bendix BDμX	Same
Main Memories		
Word length	16 bits	Same
Capacity	32K words	64K
Type	Semiconductor+ RAM	Same
I/O Processors		
Word length	8 bits	Same
Type	Intels 8080	Same
I/O Memories		
Word length	8 bits	Same
Capacity	4K bytes	Same
Busses		
Speed	< 10 microsec. per word Bit serial	Same
I/O Interfaces		
Type	1553A MILSTD	Same

* In addition, a spare unit of each type is to be built.

+ Program memory would be read only memory (ROM) for actual flight use.

factured by Bendix Corporation specifically for avionics or similar applications. The I/O processors are based upon the well-known 8080 microprocessor architecture.

To help the reader understand the operation of the units and their interaction with one another, we describe the operation of the interconnection system in abstract terms. Fig. 4 shows the connections among processors, buses, and memories. The varying replications of these connections are shown for each type of unit. Within each unit are shown a number of abstract registers that contain data or control information. Arrows that terminate at a register indicate the flow of data to the register. Arrows that terminate at the boundary of a unit indicate control signals for that unit.

We explain the operation of the interconnection system by describing how a processor p reads a word of data from location w of memory m via bus b. We assume normal operation, in which no errors or time-outs occur. Processor p initiates the READ operation by putting m and w into the register PREQUEST(p, b). Note that every processor has a separate PREQUEST register for each bus to which it is connected. When this register is loaded, a BUSREQUEST line is set to request attention from the appropriate bus. The processor must now wait until the requested bus and memory units have completed their part of the operation.

Each bus unit contains a counter-driven scanner that continuously scans the PREQUEST and BUSREQUEST lines from processors. When the scanner finds a processor that requires its attention (BUSREQUEST high), it stops and the bus is said to have been *siezed* by that processor. The bus' counter then contains the identifying number of the processor that has seized it. When seized, the bus transfers the value w from the processor to a register connected to memory m. When this transfer has been completed, the MEMREQUEST line is raised calling for attention from that memory. The bus then waits for the memory to complete its actions.

Memory units contain counter-driven scanners that operate in the same manner as those in the bus units—i.e., they con-

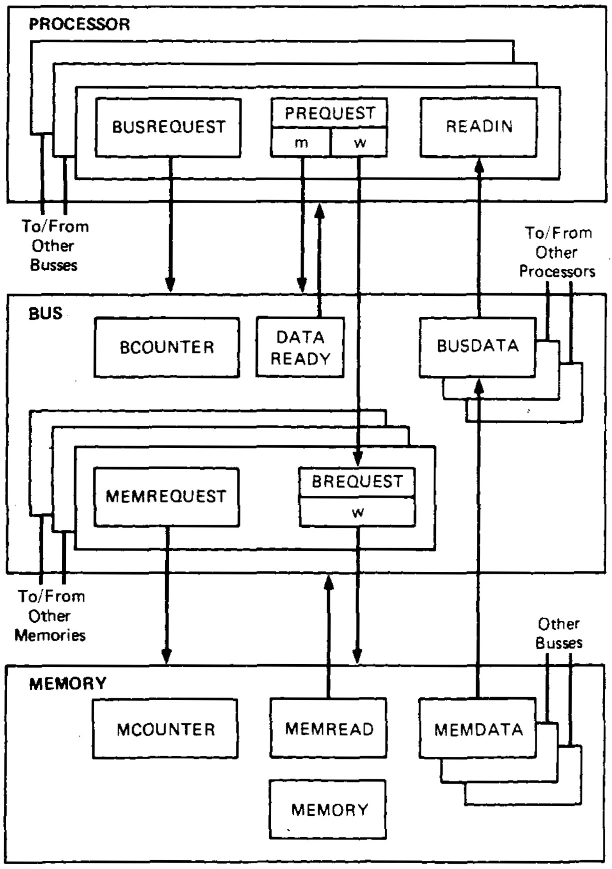

Fig. 4. An abstract view of data transfers.

tinuously scan all busses to determine which of them (if any) is requesting service. When a request is detected, the memory is said to be seized, and it reads the value w from the bus. The memory then reads the contents of its location w into MEMDATA register, and raises the MEMREAD line to inform the bus that the data are available. The memory leaves the state of MEMDATA and MEMREAD unchanged until it detects that the MEMREQUEST line from the bus has dropped, indicating that the bus has received the data from the MEMDATA register. The memory then drops the MEMREAD line and resumes scanning the buses for further requests.

When the bus detects that the MEMREAD line from the memory is up, it transfers the data in the MEMDATA register to the BUSDATA register, drops the MEMREQUEST line, and raises the DATAREADY line–indicating to the processor that the data is available. The bus leaves the state of the BUSDATA and DATAREADY lines unchanged until it detects that the BUSREQUEST line from the processor has dropped, indicating that the processor has received the data word. The bus then drops the DATAREADY line and resumes scanning the processors for further requests.

Meanwhile, the processor that made the original request has been waiting for the DATAREADY line to be raised by the bus, at which time it reads the data from the BUSDATA register. After completing this read, it drops the BUSREQUEST line and continues with other operations.

These actions have left the units in their original states. They are therefore ready to take part in other data transfer operations.

TABLE IIII

```
Data:
    READIN(p,b)
    A set of registers, one for each bus b, that receive
    data read from another processor.

    PREQUEST(p,b)
    A set of registers, one for each bus b, that hold the
    parameters of a request to read one word from another
    module's memory over that bus.


    BUSREQUEST(p,b)
    A set of booleans that indicate a request from bus b.


    ___
    A constant that is the maximum time a processor will
    wait for a bus action.

    BUS FAIL(p,b)
    A boolean indicating that processor p timed-out before
    receiving data from bus b.

External Data (generated by other units):
        DATAREADY, BUSDATA from BUS module

Abstract Program:
        REQUEST(p,b) := m,w
        D := REALTIME
        WAIT ON (DATAREADY(b) OR REALTIME >(D + ___))
        IF DATA READY (b)
           THEN BEGIN READIN(p,b) := BUSDATA(b)
                      BUSREQUEST(p,b) := FALSE
                      WAIT ON ((DATA READY = FALSE)
                                   OR (REALTIME > (D + ___))
                END
           ELSE BEGIN BUS REQUEST := FALSE
                      BUSFAIL(p,b) := TRUE
               END
```

The precise behavior of the units can be described by abstract programs. Table III is an abstract program for the processor to bus interface unit.[3] It shows the unit's autonomous control, and the manner in which the unit requests service. Note how time-outs are used to prevent any kind of bus or memory failure from "hanging up" the unit. Abstract programs for the other units are similar.

The interconnection system units designed especially for the SIFT system are:

1) the processor-to-bus interfaces;
2) the busses;
3) the bus-to-memory interfaces.

These units all operate autonomously and contain their own control, which is implemented as a simple microprogrammed controller. For example, the bus control scanner that detects the processors' requests for service is controlled by a microprogram in a programmable read-only memory (PROM). The contents of this PROM are used for two purposes: first, part of the data is fed back to the PROM's address register to determine which word of the PROM is to be read next; second, part of the data is used as logic signals that control the operation of the unit in which the PROM resides. For example, this second part could contain data to open gates to allow the flow of information from one unit to another. Input signals to the controller are applied to some of the bits of the PROM's address register, thereby affecting which PROM words are read.

The interface units (items 1 and 3 above) consist mainly of a few registers, the controller, and the gates necessary to effect the data flow. The bus with its controller (item 2) contains a larger set of such gates, since each bus can allow data flow from every memory to every processor. We estimate that the complexity of a bus unit, consisting of a bus together

[3]This program is only meant to illustrate the unit's main features; it does not accurately describe the true behavior of the unit.

with all its interfaces, is about 10 percent of that of a main processing module. The logical structure is such that an LSI version of an entire bus unit will be practical for future versions of SIFT. However, the engineering model will be a mixture of LSI and MSI (medium scale integration) technology.

The design of the interfaces permits simultaneous operation of all units. For example, a processor can simultaneously read data from its memory and from another memory, while at the same time another processor is reading from the first processor's memory. Such simultaneous operation is limited only by contention at a memory unit. This contention is handled by conventional cycle-stealing techniques and causes little delay, since the memory cycle time is small (250 ns) compared to the time needed to transfer a full word through the bus (10 μs).

Since several processors may attempt to seize the same bus, or several busses may attempt to seize the same memory, a processor can have to wait for the completion of one or more other operations before receiving service. Such waiting should be insignificant because of the small amount of data that is transmitted over the busses.

IV. The Software System

The software of SIFT consists of the application software and the executive software. The application software performs the actual flight control computations. The executive software is responsible for the reliable execution of the application tasks, and implements the error detection and reconfiguration mechanisms discussed in Section II. Additional support software to be run on a large support computer is also provided.

From the point of view of the software, a processing module —with its processor, memory, and associated registers—is a single logical unit. We will therefore simply use the term "processor" to refer to a processing module for the rest of the paper.

A. The Application Software

The application software is structured as a set of iterative tasks. As described in Section II-D, each task is run with a fixed iteration rate which depends upon its priority. The iteration rate of a higher priority task is an integral multiple of the iteration rate of any lower priority task. Every task's iteration rate is a simple fraction of the main clock frequency.

The fact that a task is executed by several processors is invisible to the application software. In each iteration, an application task obtains its inputs by executing calls to the executive software. After computing its outputs, it makes them available as inputs to the next iteration of tasks by executing calls to the executive software. The input and output of a task iteration will consist of at most a few words of data.

B. The SIFT Executive Software

Formal specifications of the executive software have been written in a rigorous form using the SPECIAL language [7] developed at SRI. These formal specifications are needed for the proof of the correctness of the system discussed in Section V. Moreover, they are also intended to force the designer to produce a well-structured system. Good structuring is essential to the success of SIFT. A sample of these SPECIAL specifications is given in the Appendix. The complete formal specification is omitted from this paper. Instead, we informally describe the important aspects of the design.

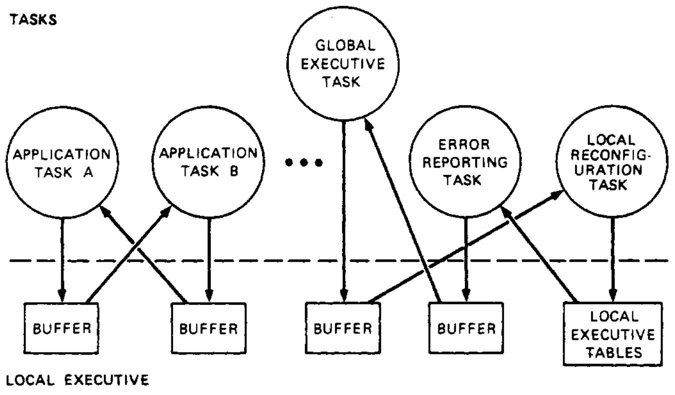

Fig. 5. Logical structure of the SIFT software system.

The SIFT executive software performs the following functions:

1) run each task at the required iteration rate;
2) provide correct input values for each iteration of a critical task (masking any errors);
3) detect errors and diagnose their cause;
4) reconfigure the system to avoid the use of failed components.

To perform the last three functions, the executive software implements the techniques of redundant execution and majority voting described in Section II. The executive software is structured into three parts:

1) the global executive task;
2) the local executive;
3) the local-global communicating tasks.

One global executive task is provided for the whole system. It is run just like a highly critical application task—being executed by several processors and using majority voting to obtain the output of each iteration. It diagnoses errors to decide which units have failed, and determines the appropriate allocation of tasks to processors.

Each processing module has its own local executive and local-global communicating tasks. The local-global communicating tasks are the error reporting task and the local reconfiguration task. Each of these tasks is regarded as a separate task executed on a single processor rather than as a replication of some more global task, so there are as many separate error reporting tasks and local reconfiguration tasks as there are processors.

Fig. 5 shows the logical structure of the SIFT software system. The replication of tasks and their allocation to processors is not visible. Tasks communicate with one another through buffers maintained by the local executives. Note that the single global executive task is aware of (and communicates with) each of the local executives, but that the local executives communicate only with the single (replicated) global executive task and not with each other. In this logical picture, application tasks communicate with each other and with the global executive, but not with the local executives.

Fig. 6 and Fig. 7 show where the logical components of Fig. 5 actually reside within SIFT. Note how critical tasks are

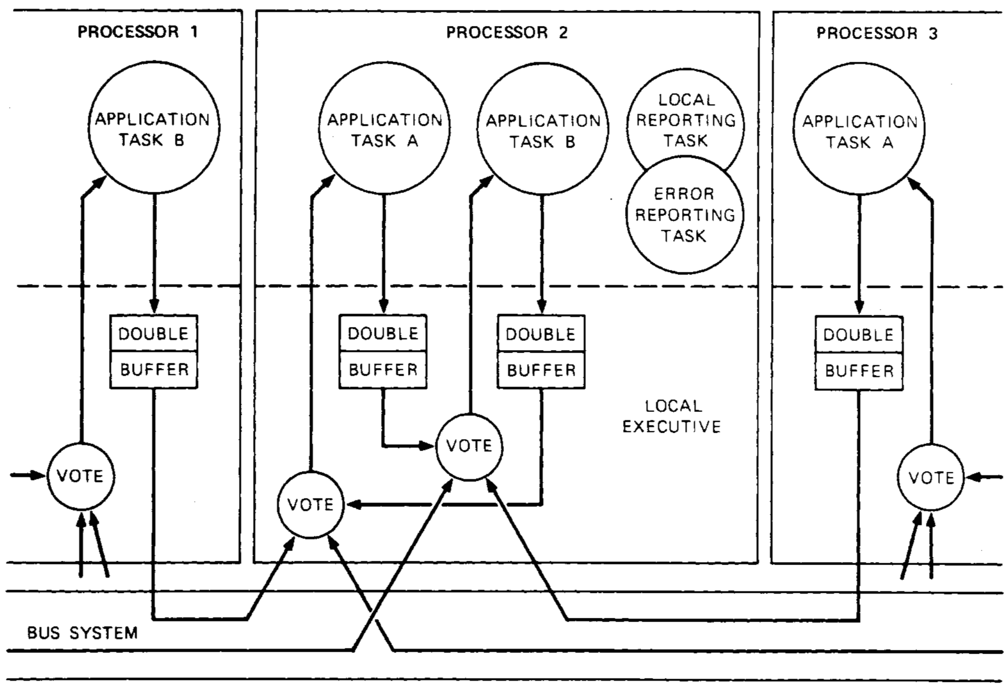

Fig. 6. Arrangement of application tasks within SIFT configuration.

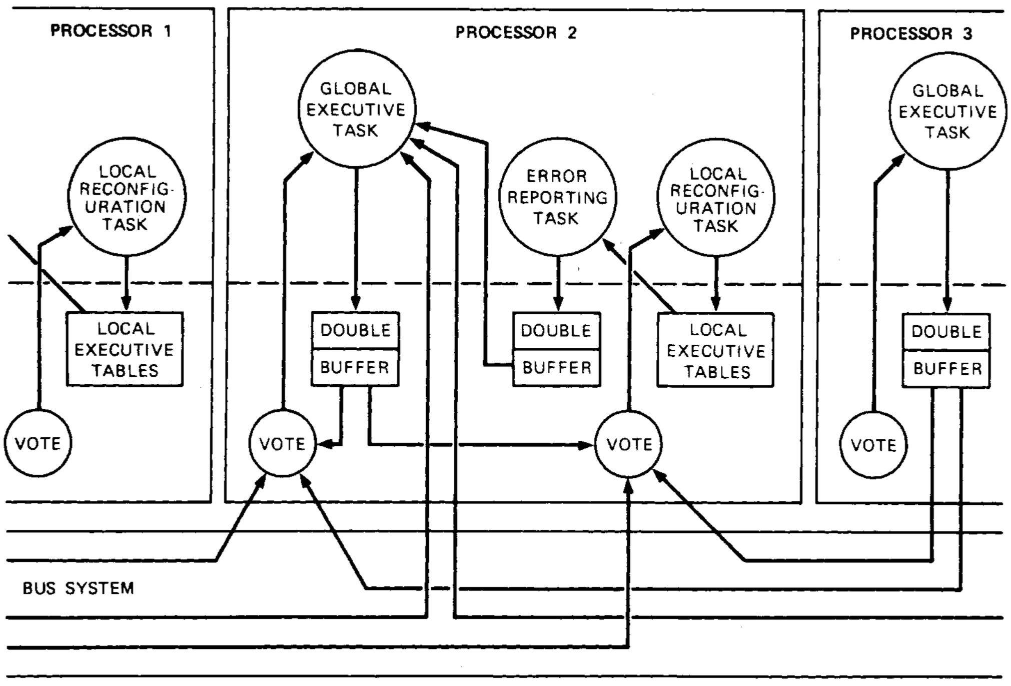

Fig. 7. Arrangement of executive within SIFT configuration.

replicated on several processors. For the sake of clarity, many of the paths by which tasks read buffers have been eliminated from Fig. 6 and Fig. 7.

1) The Local-Global Communicating Tasks: Each processor runs its local reconfiguration task and error reporting task at a specified frequency, just like any other task. These two tasks communicate with the global executive via buffers.

The local executive detects an error when it obtains different output values for the same task iteration from different processors.[4] It reports all such errors to the error reporting task. The error reporting task performs a preliminary analysis of these errors, and communicates its results to the global executive task. These results are also used by the local executive to detect possibly faulty units before the global executive has diagnosed the errors. For example, after several error reports involving a particular bus, the local executive will attempt to use other busses in preference to that one until the global executive has diagnosed the cause of the errors.

The local reconfiguration task maintains the tables used by the local executive to schedule the execution of tasks. It does this using information provided to it by the global executive.

The interaction of the global executive and the local-global communicating tasks is shown in Fig. 8.

[4] It can also detect that a time-out occurred while reading from the memory of another processing module.

1. Error handler in each processor puts reports in error table.
2. Error reporter task in each processor reads error table and decides what conditions to report to the global executive. This report is put in a buffer.
3. Global executive (triplicated) reads each processor's buffer over three busses (to guard against bus errors) and votes for a plurality.
4. Global executive, using the diagnosis provided by the error reporter, determines what reconfiguration, if any, is necessary. If a reconfiguration is necessary, a report is put in a buffer.
5. Local reconfiguration task in each processor reads report from each of the global executive buffers and votes to determine plurality.
6. Local reconfiguration task changes the scheduling table to reflect the global executive's wishes.

Fig. 8. Error reporting and reconfiguration.

2) The Global Executive Task: The global executive task uses the results of every processor's error reporter task to determine which processing modules and buses are faulty. The problem of determining which units are faulty is discussed in Section IV-C below. When the global executive decides that a component has failed, it initiates a reconfiguration by sending the appropriate information to the local reconfiguration task of each processor. The global executive may also reconfigure the system as a result of directives from the application tasks. For example, an application task may report a change of flight phase which changes the criticality of various tasks.

To permit rapid reconfiguration, we require that the program for executing a task must reside in a processor's memory before the task can be allocated to that processor. In the initial version of SIFT, there will be a static assignment of programs to memories. The program for a critical task will usually reside in all main processor memories, so the task can be executed by any main processor.

3) The Local Executive: The local executive is a collection of routines to perform the following functions: 1) run each task allocated to it at the task's specified iteration rate; 2) provide input values to, and receive output values from each task iteration, and 3) report errors to the local executive task.

A processor's local executive routine can be invoked from within that processor by a call from a running task, by a clock interrupt, or by a call from another local executive routine. There are four types of routines:

1) error handler;
2) scheduler;
3) buffer interface routines;
4) voter.

The *error handler routine* is invoked by the voter when an error condition is detected. It records the error in a *processor/bus error table*, which is used by the error reporting task described above.

The *scheduler routine* is responsible for scheduling the execution of tasks. Every task is run at a prespecified iteration rate that defines a sequence of time frames within which the task must be run. (For simplicity, we ignore the scheduling of the highest priority tasks in subframes that was mentioned in Section II-D.) A single iteration of the task is executed within each of its frames, but it may be executed at any time during that frame.

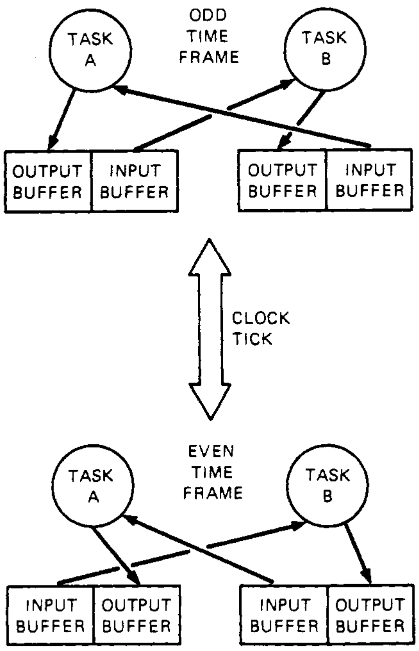

Fig. 9. The double buffering mechanism.

The scheduler is invoked by a clock interrupt or by the completion of a task. It always runs the highest priority task allocated to the processor that has not yet finished executing the iteration for its current time frame. Execution of a task may be interrupted by the clock, in which case its state is preserved until execution is resumed—possibly after the execution of a higher priority task. A task that has completed its current iteration is not executed again until after the start of its next time frame.

The *buffer interface routines* are invoked by a task when it generates output for an iteration. These routines put the output into a buffer reserved for that task. These output values are used by the voter routines described below to obtain input for the tasks. Because a task may be run at any time during its time frame, the double-buffering scheme shown in Fig. 9 is used. Each buffer consists of a double buffer. In any one time frame, one of the buffers is available for new data being generated by the task while the other contains the data generated last time frame. It is the latter values that are used to provide input to other tasks (and possibly to the same task). At the start of the next time frame, the buffers are switched around. Provision is also made for communication between processes operating at different frequencies.

The *voter routine* is invoked by a task to obtain the inputs for its current iteration. The task requests a particular output from the previous iteration of second task—which may be the same task. The voter uses tables provided by the local reconfiguration task to determine what processors contain copies of that output, and in which of their buffers. It reads the data from each of these buffers and performs a majority vote to obtain a single value. If all the values do not agree, then an error has occurred, and the error reporter is called.

C. Fault Detection

Fault detection is the analysis of errors to determine which components are faulty. In SIFT, fault detection is based upon the processor/bus error table, an m by n matrix, where m is the number of processors and n the number of busses in the system. Each processor has its own processor/bus error table that is maintained by its local executive's error handler. An entry $Xp[i, j]$ in processor p's table represents the number of errors detected by processor p's local executive that involve processor i and bus j. Suppose that processor p is reading from processor q using bus r. There are five distinct kinds of errors that cause a matrix value to change:

1) the connection from bus r to processor q is faulty;
2) the connection from processor p to bus r is faulty;
3) bus r is faulty;
4) processor q is faulty;
5) processor p is faulty;

Processor p's error reporting task analyzes the processor/bus error table as follows to determine if any of these cases hold. Let $e > 0$ be a threshold of errors that will be tolerated for any processor/bus combination. It can deduce that case 1 holds if the following conditions all hold: (i) $Xp[q, r] > e$, (ii) there exists a bus j such that $Xp[q, j] \leqslant e$, and (iii) there exists a processor i such that $Xp[i, r] \leqslant e$. Either case 2 or 3 may hold if $Xp[i, r] > e$ for all active processors i. These two cases can only be distinguished by the global executive task, which has access to information from all the processors. (Case 3 holds if all active processors report bus r faulty, otherwise case 2 holds.) The error handler can deduce that case 4 holds if $Xp[q, j] > e$ for all active buses j. The error handler cannot be depended upon to diagnose case 5, since the failure of the processor executing it could cause the error handler to decide that any (or none) of the other four cases hold.

Once the error handler has performed this analysis, the appropriate action must be taken. In case 1, processor p will stop using bus r to talk to processor q. In cases 2 and 3, processor p will stop using bus r, and will report to the global executive that bus r is faulty. In case 4, processor p will report to the global executive task that processor q is faulty.

The global executive task makes the final decision about which unit is faulty. To do this, it reads the faulty processor reports provided by the error reporting task. If two or more processors report that another processor is faulty, then the global executive decides that this other processor has indeed failed. If two or more processors report that a bus is faulty, then the global executive decides that the bus has failed.

The global executive may know that some unit produced errors, but be unable to determine which is the faulty unit. In that case, it must await further information. It can obtain such information by allocating the appropriate diagnostic tasks. If there is a faulty unit (and the error reports were not due to transient faults), then it should obtain the necessary information in a short time.

It can be shown that in the presence of a single fault, the above procedure cannot cause the global executive to declare a nonfaulty unit to be faulty. With the appropriately "malicious" behavior, a faulty unit may generate error reports without giving the global executive enough information to determine that it is faulty. For example, if processor p fails in such a way that it gives incorrect results only to processor q, then the global executive cannot decide whether it is p or q that is faulty. However, the majority voting technique will mask these errors and prevent a system failure.

D. The Simulator

An initial version of the SIFT system has been coded in Pascal. Since the avionics computer is not available at this time, the executive is being debugged on an available general-purpose computer (a DEC PDP-10). To facilitate this, a simulator has been constructed. The simulator uses five asynchronous processes, each running a SIFT executive and a "toy" set of application tasks. The controlling process simulates the actions of the SIFT bus system and facilitates interprocess communications. Faults are injected, either at the processor or the bus levels, and a visual display of the system's behavior is provided. This gives us a means of testing software in the absence of the actual SIFT hardware.

V. The Proof of Correctness

A. Concepts

Estimates of the reliability of SIFT are based upon the assumption that the software operates correctly. Since we know of no satisfactory way to estimate the probability that a piece of software is incorrect, we are forced to try to guarantee that the software is indeed correct. For an asynchronous multiprocess system such as SIFT, the only way to do this is to give a rigorous mathematical proof of its correctness.

A rigorous proof of correctness for a system requires a precise statement of what it means for the system to be correct. The correctness of SIFT must be expressed as a precise mathematical statement about its behavior. Since the SIFT system is composed of several processors and memories, such a statement must describe the behavior of many thousands of bits of information. We are thus faced with the problem that the statement of what it means for the SIFT software to be correct is too complicated to be humanly comprehensible.

The solution to this problem is to construct a higher level "view" of the SIFT system that is simpler than the actual system. Such a view is called a *model*. When stated in terms of the simple model, the requisite system properties can be made comprehensible. The proof of correctness is then performed in two steps: 1) we first prove that the model possesses the necessary correctness properties; and 2) we then prove that the model accurately describes the SIFT system [12].

Actually, different aspects of correctness are best expressed in terms of different models. We use a hierarchy of models. The system itself may be viewed as the lowest level model. In order to prove that the models accurately describe the SIFT system, we prove that each model accurately describes the next lower-level one.

B. Models

We now make the concept of a model more precise. We define a model to consist of a set S of possible states, a subset S_0 of S consisting of the set of possible initial states, and a *transition relation* $\rightarrow$ on S. The relation $s \rightarrow s'$ means that a transition is possible from state s to state s'. It is possible for the relations $s \rightarrow s'$ and $s \rightarrow s''$ both to hold for two different states s' and s'', so we allow nondeterministic behavior. A *possible behavior* of the system consists of a sequence of states $s_0, s_1, \cdots$ such that s_0 is in S_0 and $s_i \rightarrow s_{i+1}$ for each i. Correctness properties are mathematical statements about the possible behaviors of the system.

Note that the behavior of a model consists of a linear sequence of transitions, even though concurrent operations occur in the SIFT system. Concurrently activity can be represented by transitions that change disjoint components of the state, so that the order in which they occur is irrelevant.

Each state of the model represents a collection of states in the real system. For example, in the reliability model discussed in Section II-F, the state is a triple of integers (h, d, f)

which contains only the information that f processors have failed, d of those failures have been detected, and h of the detected failures have been handled. A single model state corresponds to all possible states the system could reach through any combination of f failures, d failure detections, and h reconfigurations.

We now consider what it means for one model to accurately describe a lower level one. Let S, S_0, and $\rightarrow$ be the set of states, set of initial states, and transition relation for the higher level model; and let S', S'_0, and $\rightarrow'$ be the corresponding quantities for the lower level model. Each state of the lower level model must represent some state of the higher level one, but different lower level states can represent the same higher level one. Thus there must be a mapping $\text{REP}: S' \rightarrow S$, where $\text{REP}(s')$ denotes the higher-level state represented by s'.

Having defined a correspondence between the states of the two models, we can require that the two models exhibit corresponding behavior. Since the lower level model represents a more detailed description of the system, it may contain more transitions than the higher level one. Each transition in the lower level model should either correspond to a transition in the higher level one, or else should describe a change in the system that is invisible in the higher level model. This requirement is embodied in the following two conditions.

1) $\text{REP}(S'_0)$ is a subset of S_0.
2) For all s', t' in S': if $s' \rightarrow' t'$ then either:
   (a) $\text{REP}(s') = \text{REP}(t')$; or
   (b) $\text{REP}(s') \rightarrow \text{REP}(t')$.

If these conditions are satisfied, then we say that REP defines the lower level model to be a *refinement* of the higher level one.

If a model is a refinement of a higher level one, then any theorem about the possible behaviors of the higher level model yields a corresponding theorem about the possible behaviors of the lower level one. This is used to infer correctness of the lower level model (and ultimately, of the system itself) from the correctness of the higher level one.

A transition in the higher level model may represent a system action that is represented by a sequence of transitions in the lower level one. For example, the action of detecting a failure may be represented by a single transition in the higher level model. However, in a lower level model (such as the system itself), detecting a failure may involve a complex sequence of transitions. The second requirement means that in order to define REP, we must define some arbitrary point at which the lower level model is considered to have detected the failure. This problem of defining exactly when the higher level transition takes place in the lower level model turns out to be the major difficulty in constructing the mapping REP.

C. *The Reliability Model*

In the reliability model, the state consists of a triple (h, d, f) of integers with $h \leqslant d \leqslant f \leqslant p$, where p is the number of processors. The transition relation $\rightarrow$ is described in Section II-F, as is the meaning of the quantities h, d, and f.

Associated with each value of h is an integer $\text{sf}(h)$ called its *safety factor*, which has the following interpretation. If the system has reached a configuration in which h failures have been handled, then it can successfully cope with up to $\text{sf}(h)$ additional (unhandled) failures. That is, the system should function correctly so long as $f - h$, the number of unhandled failures, is less than or equal to $\text{sf}(h)$. The state (h, d, f) is called *safe* if $f - h \leqslant \text{sf}(h)$.

To demonstrate that SIFT meets its reliability requirements, we must show two things.

1) If the system remains in a safe state (one represented by a safe state in the reliability model), then it will behave correctly.

2) The probability of the system reaching an unsafe state is sufficiently small.

Property 2) was discussed in Section II-F. The remainder of Section V describes our approach to proving 1).

The reliability model is introduced specifically to allow us to discuss property 2). The model does not reflect the fact that SIFT is performing any computations, so it cannot be used to state any correctness properties of the system. For that, a lower level model is needed.

D. *The Allocation Model*

1) An Overview: SIFT performs a number of iterative tasks. In the *allocation model*, a single transition represents the execution of one complete iteration of all the tasks. As described in Section II-D, most tasks are not actually executed every iteration cycle. For the allocation model, an unexecuted task is considered to perform a null calculation, producing the same result it produced during the previous iteration.

The input used by a task in its tth iteration is the output of the $(t - 1)$st iterations of some (possibly empty) set of tasks. Input to SIFT is modeled by a task executed on an I/O processor which produces output without requiring input from other tasks. The output which an I/O processor produces is simply the output of some task which it executes.

In the allocation model, we make no distinction between main processors and I/O processors. Bus errors are not represented in the model. SIFT's handling of them is invisible in the allocation model, and can be represented by a lower level model.

The fundamental correctness property of SIFT—property 1) of Section V-C above—is stated in terms of the allocation model as follows: if the system remains in a safe state, then each nonfaulty processor produces correct output for every critical task it executes. This implies the correctness of any critical output of SIFT generated by a nonfaulty I/O processor. (The possibility of faulty I/O processors must be handled by redundancy in the external environment.)

The allocation of processors to tasks is effected by the interaction of the global executive task, the local–global communicating tasks, and local executives, as described in Section IV. The output of the tth iteration of a local-global communicating task uses as input the output of the $(t - 1)$st iteration of the global executive. During the tth iteration cycle, the local executive determines what the processor should be doing during the $(t + 1)$st cycle—i.e., what tasks it should execute, and what processor memories contain the input values for each of these tasks. The processor executes a task by fetching each input from several processor memories, using a majority vote to determine the correct value, and then computing the task's output.[5] We assume that a nonfaulty processor will compute the correct output value for a task if majority voting obtains the correct value for each of the task's inputs.

The only part of the executive software that is explicitly represented in the allocation are the local–global communicating tasks. Although each processor's local–global communicating task is treated in SIFT as a separate task, it is more convenient to represent it in the allocation model as the execu-

[5] The fault diagnosis performed by the global executive is not represented in the allocation model.

tion on that processor of a single replicated task whose output determines the complete allocation of tasks to processors.

2) The States of the Allocation Model: We now describe the set of states of the allocation model. They are defined in terms of the primitive quantities listed below, which are themselves undefined. (To show that a lower level model is a refinement of the allocation model, we must define these primitive quantities in terms of the primitive quantities of that lower level model.) The descriptions of these quantities are given to help the reader understand the model; they have no formal significance.

P	A set of processors. It represents the set of all processors in the system.
K	A set of tasks. It represents the set of all (critical) tasks in the system.
LE	An element of K. It is the single task that represents all the local-global communicating tasks, as described above.
e	A mapping from the cross product of K and the set of nonnegative integers into some unspecified set of values. The value of $e(k, t)$ represents the correct output of the tth iteration cycle of task k. Thus, e describes what the SIFT tasks should compute. It is a primitive (i.e., undefined) quantity in the allocation model because we are not specifying the actual values the tasks should produce. (These values will, of course, depend upon the particular application tasks SIFT executes, and the inputs from the external environment.)
sf	The safety factor function introduced in the reliability model. It remains a primitive quantity in the allocation model. It can be thought of as a goal the system is trying to achieve.

We define the allocation model state to consist of the following components.[6] (Again, the descriptions are to assist the reader and are irrelevant to the proof.)

t	A nonnegative integer. It represents the number of iteration cycles that have been executed.
F	A subset of P. It represents the set of all failed processors.
D	A subset of F. It represents the set of all failed processors whose failure has been detected.
c	A mapping from $P \times K$ into some unspecified set of values. The value $c(p, k)$ denotes the output of task k as computed by processor p. This value is presumably meaningless if p did not execute the tth iteration of task k.

3. The Axioms of the Model: We do not completely describe the set of initial states S_0 and the transition relation $\rightarrow$ for the allocation model. Instead, we give the following list of axioms about S_0 and $\rightarrow$. Rather than giving their formal statement, we simply give here an informal description of the axioms. (Uninteresting axioms dealing with such matters as initialization are omitted.)

1) The value of $c(p, LE)$ during iteration cycle t, which represents the output of the tth iteration of processor p's local-global communicating task, specifies the tasks that p should execute during cycle $t + 1$ and the processors whose memories contain input values for each such task.

2) If a nonfaulty processor p executes a task k during iteration cycle t, and a majority of the copies of each input value to k received by p are correct, then the value $c(p, k)$ it computes will equal the correct value $e(k, t)$.

3) Certain natural assumptions are made about the allocation of tasks to processors specified by $e(LE, t)$. In particular, we assume that a) no critical tasks are assigned to a processor in D (the set of processors known to be faulty), and b) when reconfiguring, the reallocation of tasks to processors is done in such a way that the global executive never knowingly makes the system less tolerant of failure than it currently is.

To prove that a lower level model is a refinement of the allocation model, it will suffice to verify that these axioms are satisfied.

4) The Correspondence with the Reliability Model: In order to show that the allocation model is a refinement of the reliability model, we must define the quantities h, d, and f of the reliability model in terms of the state components of the allocation model—thereby defining the function REP.

The definitions of d and f are obvious; they are just the number of elements in the sets D and F, respectively. To define h, we must specify the precise point during the "execution" of the allocation model at which a detected failure is considered to be "handled." Basically, the value of h is increased to $h + 1$ when the reconfiguration has progressed to the point where it can handle sf$(h + 1)$ *additional* errors. (The function sf appears in the definition.) We omit the details.

5. The Correctness Proof: Within the allocation model, we can define a predicate $CF(t)$ that expresses the condition that the system functions correctly during the tth iteration cycle. Intuitively, it is the statement that every nonfaulty processor produces the correct output for every task it executes. The predicate $CF(t)$ can be stated more precisely as follows.

> If $e(LE, t - 1)$ indicates that p should execute a task k in K during the tth iteration cycle, and p is in $P - F$, then the value of $c(p, k)$ after the tth iteration equals $e(k, t)$.

[A precise statement of how $e(LE, t - 1)$ indicates that p should execute task k requires some additional notation, and is omitted.]

We can define the predicate SAFE(t) to mean that the system is in a safe state at time t. More precisely, SAFE(t) means that after the tth iteration cycle, sf$(h) \geqslant f - h$, where f and h are defined above as functions of the allocation model state. The basic correctness condition for SIFT can be stated as follows.

> If SAFE(t') is true for all t' with $0 < t' < t$, then $CF(t)$ is true.

A rigorous proof of this theorem has been developed, based upon the axioms for the allocation model. The proof is too long and detailed to include here. It will appear in the final report to NASA at the conclusion of the current phase of the project.

E. Future Work

The basic correctness property of SIFT has been stated and proved for the allocation model. What remains to be done is to show that the actual system is a refinement of the allocation model. Current plans call for this to be done in terms of two lower level models. The first of these is the *operating system model.* The allocation model represents all the computations in a given iteration cycle performed by all the processes as a single transition. The operating-system model wi

[6] To simplify the discussion, one component of our actual model has been omitted.

represent the asynchrony of the actual computations. It will essentially be a high-level representation of the system that embodies the mechanisms used to synchronize the processors. The proof that the operating-system model is a refinement of the allocation model will be a proof of correctness of these synchronizing mechanisms.

The next lower level model will be the *program model.* It will essentially represent the PASCAL version of the software. We expect that proving the program model to be a refinement of the operating-system model will be done by the ordinary methods of program verification [11].

Finally, we must verify that the system itself is a correct refinement of the program model. This requires verifying first that the Pascal programs are compiled correctly, and second that the hardware correctly executes programs. (In particular, this involves verifying the fault-isolation properties of the hardware.) We have not yet decided how to address these tasks. Although most of this verification is theoretically straightforward, it presents a difficult problem in practice.

VI. Conclusions

The SIFT computer development is an attempt to use modern methods of computer design and verification to achieve fault-tolerant behavior for real-time, critical control systems. We believe that the use of standard, mass-produced components helps to attain high reliability. Our basic approach, therefore, involves the replication of standard components, relying upon the software to detect and analyze errors and to dynamically reconfigure the system to bypass faulty units. Special hardware is needed only to isolate the units from one another, so a faulty unit does not cause the failure of a nonfaulty one.

We have chosen processor/memory modules and bus modules as the basic units of fault detection and reconfiguration. These units are at a high enough level to make system reconfiguration easy, and are small and inexpensive enough to allow sufficient replication to achieve the desired reliability. Moreover, new advances in Large Scale Integration will further reduce their size and cost.

By using software to achieve fault-tolerance, SIFT allows considerable flexibility in the choice of error handling policies and mechanisms. For example, algorithms for fault masking and reconfiguration can be easily modified on the basis of operational experience. Novel approaches to the tolerance of programming errors, such as redundant programming and recovery blocks [8] can be incorporated. Moreover, it is fairly easy to enhance the performance of the system by adding more hardware.

While designing SIFT, we have been concerned with proving that it meets its stringent reliability requirements. We have constructed formal models with which to analyze the probability of system failure, and we intend to prove that these models accurately describe the behavior of the SIFT system. Our effort has included the use of formal specifications for functional modules. We hope to achieve a degree of system verification that has been unavailable in previous fault-tolerant architectures.

Although the design described in this paper has been oriented toward the needs of commercial air transports, the basic architectural approach has a wide applicability to critical real-time systems. Future work may extend this approach to the design of fault-tolerant software and more general fault-tolerant control systems.

Appendix A: Sample special Specification

This appendix contains an example of a formal specification extracted from the specifications of the SIFT executive software. The specification is written in a language called SPECIAL, a formally defined specification language. SPECIAL has been designed explicitly to permit the description of the results required from a computer program without constraining the programmer's decisions as to how to write the most efficient program.

The function that is specified here is the local executive's voter routine, described informally in Section IV-A. This function is called to obtain a value from one of the buffers used to communicate between tasks. The value required is requested over the bus system from every replication of this buffer, and a consensus value that masks any errors is formed and returned to the calling program. Errors are reported and provision is made for buses that do not obtain a value (due to a nonresponding bus or memory) and for the possibility that there is no consensus.

Notes following the specification are keyed to statements in the specification.

```
OVFUN read_buffer (buffer_name i; address k; value safe)
                  [processor a; task t]
                  → result r;                                          [1]

EXCEPTIONS                                                              [2]
        CARDINALITY(activated_buffers(a,i)) = 0;
        0 > k OR k >= buffer_size(i);

EFFECTS                                                                 [3]
   EXISTS SET_OF response
            w = responses(a, activated_buffers(a,i), k):
      EXISTS SET_OF response
               z = {response b | b INSET w AND b.flag }:

         IF(EXISTS value v;                                             [4]
                  SET_OF response x |
                  x = {response c | c INSET (w DIFF z)
                                  AND c.val = v}:
```

```
            FORALL value u;                                                [5]
                    SET_OF response y |
                    y = {response d | d INSET (w DIFF x DIFF z)
                                     AND d.val = u}:
              CARDINALITY (x) > CARDINALITY(y))

       THEN(EXISTS value v;                                                [6]
                   SET_OF response x |
                   x = {response c | c INSET (w DIFF z)
                                    AND c.val = v}:

              FORALL value u;                                              [6]
                      SET_OF response y |
                      y = {response d | d INSET (w DIFF x DIFF z)
                                       AND d.val = u}:
                CARDINALITY(x) > CARDINALITY(y);

              EFFECTS_OF errors(a, w DIFF x);                              [7]
              r = v)

       ELSE(EFFECTS_OF errors(a, w);                                       [8]
            r = safe);
```

Notes

1) The function 'read_buffer' takes three arguments and returns a result. The buffer_name 'i' is the name of a logical buffer which may be replicated in several processors, while the address 'k' is the offset of the required word in the buffer and 'safe' is the value to be returned if no consensus can be obtained. The parameters 'a' and 't' need not be explicitly cited by the caller of this function but are deduced from the context.

2) Exception returns will be made if there are no active instances of the named buffer or if the offset is not within the buffer.

3) A response is obtained by interrogating a buffer in another processor. Each response is a record (also known as a "structure", containing a value field ("val") and flag field ("flag"), the latter set if no response was obtained from the bus or store. The set 'w' of responses is the set obtained from all of the activated buffers known to processor 'a'. The set 'z' is the subset of no-response responses.

4) First we must check that a plurality opinion exists. This section hypothesises that there exists a consensus value 'v' together with the subset of responses 'x' that returned that value.

5) Here we consider all other values and establish for each of them that fewer responses contained this other value than contained the proposed consensus value.

6) Having established that a consensus value exists, we may now validly construct it, repeating the criteria of stages [4] and [5]. It is important to note that these are not programs but logical criteria. The actual implementations would not repeat the program.

7) This section requires that any responses not in the set 'x' (the set 'x' is the set reporting the consensus value) should be reported as errors, and the result is the consensus value 'v'. The expression

EFFECTS_OF errors(a, w DIFF X)

indicates a state change in the module that contains the O-function "errors". The specification indicates that an error report is loaded into a table associated with processor "a."

8) If there is no consensus value, as determined by stages [4] and [5], then all the responses must be reported as errors, and the safe value returned as the result.

Acknowledgment

The authors wish to acknowledge the help of other members of the Computer Science Laboratory who contributed to the development of SIFT. In particular, Dr. William H. Kautz helped in the formulation of the reliability model and with the diagnosis problem. Marshall Pease developed a proof showing that synchronization could not be achieved with three clocks. Lawrence Robinson indirectly aided the project by his creation of the hierarchical development methodology. We are indebted to numerous individuals of NASA-Langley Research Center: Nicholas D. Murray, the Project Monitor has provided early and continuing guidance and encouragement, Billy Dove has provided inspiration and support within the context of a long-range NASA program of technology development for reliable aircraft control, Earl Migneault first alerted us to problems with the "obvious" solutions to the clock synchronization problem, Sal Bavuso has continually reviewed our work on reliability modeling, and Brian Lupton and Larry Spencer have provided considerable valuable comments during the course of the work.

References

[1] N. D. Murray, A. L. Hopkins, and J. H. Wensley, "Highly reliable multiprocessors," AGARDograph No. 224, *Integrity in Electronic Flight Control Systems*, P. R. Kurzhals, Ed., Advisory Group for Aerospace Research and Development, Neuilly Sur Seine, France, pp. 17.1-17.16, Apr. 1977.
[2] J. H. Wensley, *et al.*, "Architecture," vol. I of *Design of a Fault Tolerant Airborne Digital Computer*," SRI International Technical Report for NASA, CR-132252, SRI International, Menlo Park, CA. Oct. 1973.
[3] R. S. Ratner, *et al.*, "Computational requirements and technology," vol. II of *Design of a Fault Tolerant Airborne Digital Computer*, SRI Technical Report for NASA, CR-132253, SRI International, Menlo Park, CA, Oct. 1973.
[4] J. H. Wensley, "SIFT software implemented fault tolerance," in *Proc. Fall Joint Computer Conf.*, AFIPS Press, Montvale, NJ. 1972, vol. 41, pp. 243-253.
[5] J. H. Wensley, M. W. Green, K. N. Levitt, and R. E. Shostak, "The design, analysis, and verification of the SIFT fault tolerant system," *Proc. 2nd Int. Conf. Software Engineering*, IEEE Catalog No. 76, ch 1125-4 C, IEEE Computer Society, Long Beach, CA, pp. 458-469, 1976.
[6] P. M. Melliar-Smith, "Permissible processor loadings for various

scheduling algorithms," Memorandum, SRI International, Menlo Park, CA. 1977.
[7] L. Robinson and O. Roubine, "SPECIAL—A specification and assertion language," Technical Report CSL-46, SRI International, Menlo Park, CA, Jan. 1977.
[8] B. Randell. "System structure for software fault tolerance," *IEEE Trans. Software Eng.*, vol. SE-1(2), pp. 220–232, June 1975.
[9] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," manuscript in preparation.
[10] L. Robinson, K. N. Levitt, P. G. Neumann, and A. K. Saxena, "A formal methodology for the design of operating system software," in *Current Trends in Processing Methodology*, vol. 1, R. T. Yeh, Ed. Englewood Cliffs, New Jersey: Prentice-Hall. 1976.
[11] R. W. Floyd, "Assigning meanings to programs," *Mathematical Aspects of Computer Science*, vol. 19, J. T. Schwartz, Ed. Providence, RI: Amer. Mathematical Society, 1967, pp. 19–32.
[12] R. E. Shostak *et al.*, "Proving the reliability of a fault-tolerant computer system," *Proc. 14th IEEE Comput. Soc. Int. Conf.*, San Francisco, CA, 1977.

Architectures for Fault-Tolerant Spacecraft Computers

DAVID A. RENNELS

Abstract—**This paper summarizes the results of a long-term research program in fault-tolerant computing for spacecraft on-board processing. In response to changing device technology this program has progressed from the design of a fault-tolerant uniprocessor to the development of fault-tolerant distributed computer systems. The unusual requirements of spacecraft computing are described along with the resulting real-time computer architectures. The following aspects of these designs are discussed: 1) architectural features to minimize complexity in the distributed computer system, 2) fault-detection and recovery, 3) techniques to enhance reliability and testability, and 4) design approaches for LSI implementation.**

I. Introduction

FAULT-TOLERANT computing has long been a requirement of planetary spacecraft. These systems are built at costs of tens to hundreds of million-dollars and then sent into space for several-years mission during which repair is not possible. Failure of an on-board computer can mean loss of a mission. Thus redundant backup units are carried along for the computer and other critical subsystems, and faulty units are automatically replaced with spares. The Jet Propulsion Laboratory (JPL) of the California Institute of Technology has built and operated spacecraft which have successfully explored the Moon, Mars, Venus, and Mercury [1]. Current spacecraft are directed toward the mysterious outer planets of Jupiter and Saturn. In support of spacecraft reliability requirements, a program in fault-tolerant computing has been conducted at JPL for nearly 20 years. This paper summarizes the results of this research and the directions it has taken in response to changes in the underlying circuit technology.

Manuscript received May 6, 1978.
The author is with the Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA 91103.

The primary constraints on the on-board computing system are the requirements for long unattended life and severe restrictions on power, weight, and volume. Reliability is the most severe constraint which affects the computer architecture in several ways. In most cases only proven (5–10-year old) technology can be used to minimize the chance of unexpected failure modes. Parts are extensively tested and screened for reliability, driving their cost to ten or more times those in the commercial marketplace. Redundant processors, memories, and input/output (I/O) circuits double or triple the amount of hardware that is used. Thus it can be safely said that reliability requirements induce the majority of costs for on-board computing.

Typical power, weight, and volume requirements are in order of 30–50 W, 100 lbs, and a few thousand cubic inches. These physical constraints become especially severe since redundant spare modules must be included. Thus our early fault-tolerance efforts, which were based on relatively bulky and power consuming bipolar small-scale integrated (SSI) circuits, were oriented toward finding hardware-efficient forms of fault-tolerant computer architectures. This constraint has been somewhat reduced with the current availability of low-power higher density CMOS devices.

The JPL program in fault-tolerant computing has had two major parts. The *first* was the development of a fault-tolerant uniprocessor designated the JPL self-testing and repairing (STAR) computer. This development was carried out under the direction of A. Avizienis between 1961 and 1972. It was aimed at the flight technology of the early 1970's (e.g., bipolar SSI/MSI and plated-wire memory) and the results were widely published [2]. A breadboard STAR computer was constructed and tested in 1970–1972.