SIFT—Software Implemented Fault Tolerance

by JOHN H. WENSLEY

Stanford Research Institute
Menlo Park, California

INTRODUCTION

Many computer applications have stringent requirements for continued correct operation of the computer in the presence of internal faults. The subject of design of such highly reliable computers has been extensively studied,[11-14] and numerous techniques have been developed to achieve this high reliability. Such computers are termed "fault tolerant"; examples of applications are found in the aerospace industry, communication systems, and computer networks. Several designs of such systems have been proposed[2,5,8,11,12,13] and some have been implemented. In general, these designs contain extensive hard-wired logic for such functions as fault masking, comparison, switching, and encoding-decoding.

This paper describes a new approach to the design of a fault-tolerant computer, with strong emphasis on software techniques to achieve fault tolerance and corresponding deemphasis on special hardware units. One characteristic of the particular software approach taken is that erroneous results are not detected immediately after they occur, but rather at the conclusion of the processing of a task. However, the errors are not permitted to propagate.

The particular design discussed here is tailored to the use of computers for control functions in an advanced technology transport aircraft; this application determines the scale of the proposed system. Although extension to other applications might change the size or speed of the system (or its units), the basic concepts have sufficient generality to cover many applications.

In designing the system, a basic consideration is that the advent of large-scale-integrated (LSI) circuits implies that any reconfiguration or discarding of equipment should be carried out at the unit level (CPUs or memory blocks) rather than at the component level (gates or registers). In addition, economic use of LSI demands that the number of different types of units be minimized, with high replication of each type.

Fault-tolerant computer systems vary greatly in reliability requirements. A typical requirement in space applications is for a probability between 95 percent and 99 percent that computing capability will exist after 5 to 10 years of operation. This implies a mean time between failure (MTBF) from 100 to 1000 years.* In the control of an aircraft, to which this design was aimed, the requirement was for a probability of failure less than 10^{-8} during a 10-hour operational period. This translates to a MTBF of 10^4 years, i.e., 10 to 100 times more stringent than the above. The consequence of failure (possible loss of human lives and economic loss) is, in this application, extremely high and justifies the use of extensive redundancy in the computer system where cost is, even with redundancy, a small proportion of total aircraft cost. The computing load for this application is such that the computer must have approximately 16K words of memory and be capable of better than 0.5 MIPS.** Assuming LSI circuitry with a chip failure probability of 10^{-6} per hour, the overall system design must assume correct functional behavior in the presence of multiple chip failures, which can be expected in a computer system containing several hundred LSI chips.

We are concerned with faults in the two major subsystems, i.e., the processor and the memory. With reasonable predictions concerning LSI development in the next few years, analysis shows that the processor will require approximately 10 percent of the chips required for the memory. Therefore, we regard

---

* This statement does not imply that a single computer will survive for 100 to 1000 years, but that n such computers will, after y years, have suffered $n \cdot y/100$ or $n \cdot y/1000$ failures.

** Millions of instructions per second.

replication of the processor as an economic checking and fault-masking technique. Protection of the memory function can be carried out either by replication or coding or by a combination of both. This paper describes a system using memory replication, but the basic concepts are compatible with alternative methods for protecting the memory.

The important features of the system can be implemented by a range of techniques going from hardware, through microprogram, through system software, to application software. The computer system described in this paper places heavy emphasis on the use of software to carry out fault detection and correction procedures. The fault-tolerant procedures can be made transparent to the application programmer by suitable design of the support software such as compilers or assemblers. A system in which such functions are achieved by suitably designed hardware (or microprogram) is also possible.

A central feature of the described system is the prevention of fault propagation by the use of read-only connections between processing modules. Another important feature is the avoidance of any need for a "lock-step" operation of replicated units, and a reduction in the frequency of fault checking. This results from the strategy of only checking when the state of the controlled (aircraft) system changes rather than at each change of state of the computer. An implementation in which more of the fault-tolerant features were in hardware would be faster in operation at the expense of flexibility of change that is given by the software implementation as described. The system as presented gives the designer the freedom to tailor the system to the application by the following important trade-off possibilities:

- An increase of speed by placing more of the fault-tolerant functions in hardware or microprograms.
- Flexibility for varying the amount of protection given to different application programs by using software fault-tolerant techniques.
- Ability to change the fault-tolerant strategies as new technologies emerge with new reliability characteristics.

BACKGROUND

Existing designs of fault-tolerant computing systems use a variety of redundancy techniques to achieve fault tolerance. These techniques include, for example, special codes for error detection and correction, and the replication of units with means for detecting whether or not several units carrying out the same operation are in agreement.

The JPL STAR computer[2] uses several redundant codes at different parts of the system, as well as special-purpose hardware to perform checking and rollback. The approach taken by Hopkins[11] does not include the extensive use of redundant codes but relies heavily on replicated CPUs, busses, and memories, with special units to check for agreement between units. These and other systems are designed to detect errors soon after they have occurred—usually before the state of the CPUs has been irreversibly changed or a memory cell has been overwritten. These systems require that any replicated units must stay in close step with each other, usually at the instruction level (so-called lock step). When detected, a faulty unit in these systems is removed from the system (e.g., by switching or removing power). If no provision is made to return a once faulty unit to service when it returns to correct operation, a severe cost penalty is incurred in the event of transient errors.

The system we describe has many properties that, in total, distinguish it from other fault-tolerant systems.

- Replicated units do not operate in lock-step mode, but are only loosely synchronized. The communication between CPUs is asynchronous, thereby removing the need for an ultrareliable system clock.
- Agreement between replicated units is verified only at the completion of program segments (tasks).
- Faulty units are not necessarily removed but can either be ignored or assigned to tasks having no overall effect.
- Transient faults do not necessarily cause permanent removal of the faulty units. Furthermore, the looseness of synchronization among sets of tasks makes it possible to enhance immunity from transients, by providing that redundant versions of a computation may be done at different moments in time.
- The degree of fault tolerance can be different for different tasks being performed, and can be different at different times for the same task.
- No special hardware is used to carry out fault detection or correction.
- Communication between CPUs is minimized so that low bandwidth busses can be used, thereby facilitating physical separation of modules in environments where physical damage is a hazard.
- The design concept is independent of the way in

which the units are built, i.e., no specialization of CPU or memory design is required for fault tolerance, thereby allowing the choice to be based on other properties, e.g., speed, availability.

- The total computing power of the system can be varied by using units of different speed or by changing the number of units.

SYSTEM DESIGN

The system (Figure 1) consists of a number of modules, each composed of a memory and a processing unit. The individual processing units within the modules are connected to the corresponding memory units with wide bandwidth busses. The intermodule bus organization (B_1,B_2,B_3)* is designed to allow a processor to read from any memory but not to write into other memory units. The intermodule bus is expected to have a much lower bandwidth than an intramodule bus.

The input/output (I/O) system, discussed in a later section, is assumed to be connected to the busses B_1, B_2, and B_3. The input/output system shown in Figure 1 consists of all the noncomputing units, e.g., transducers, actuators, and sensors. The part of the total input/output that is carried out by program, e.g., formatting or code conversion, is handled in the same manner as any other task, i.e., is replicated in several processors.

The system is viewed as being regular in that no module is *a priori* assigned a special role. All computations that require high reliability are carried out in several modules. We assume for the purpose of this description that critical tasks are processed in three units.

The computations that must be carried out are broken into a number of tasks in such a way that no task requires more computing power than can be supplied by one processor. The tasks are given the designations, A, B, C . . .; the processors are numbered 1, 2, 3. . . . Each processor is capable of being multiprogrammed over a number of tasks, as illustrated in Figure 2.

The control of the computing system is carried out by an executive system that can be segmented by function into two parts:

(1) Local Executive: functions that apply to each processor (e.g., dispatching,* reporting errors, loading new task programs).

(2) System Executive: functions that are global to the system (e.g., allocation and scheduling of work load, reconfiguring).

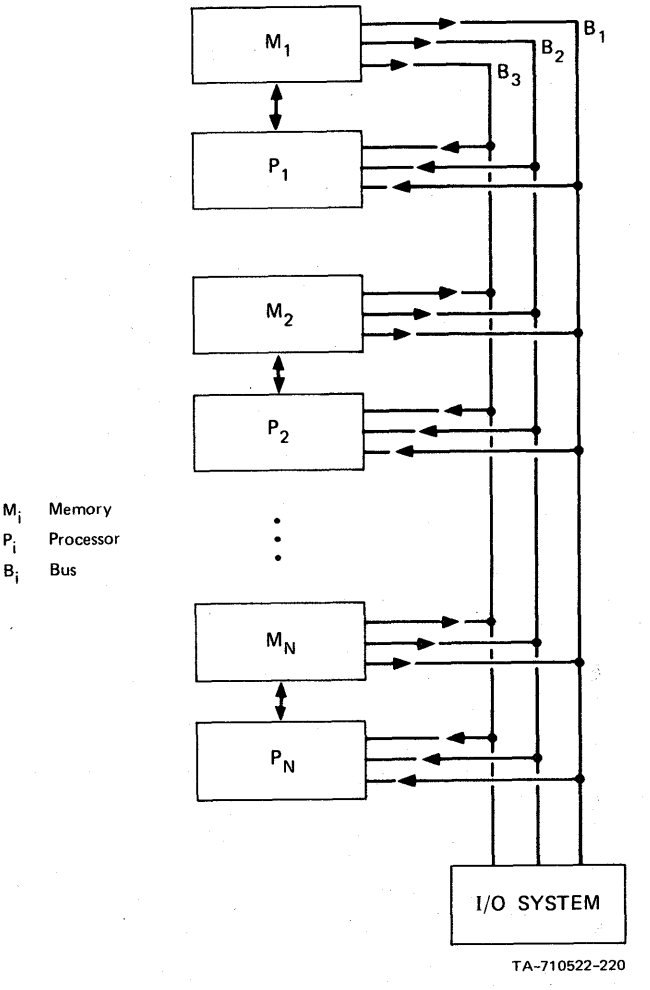

Figure 1—System configuration

A complete set of the software functions of Class 1 is present in each processor; those in Class 2 are carried out in a sufficient number of processors to provide the degree of fault tolerance required. The functions are realized by programs that have the same task structure as all other programs.

The normal operating mode for a processor carrying out a task is to follow the flow of control shown in Figure 3. Data required for the task are assumed to have been computed by several processors (including

* The bus logic envisioned does not use voting. The number three is chosen for convenience of discussion.

* Dispatching is the executive function that initiates a new task at the completion of the previous one.

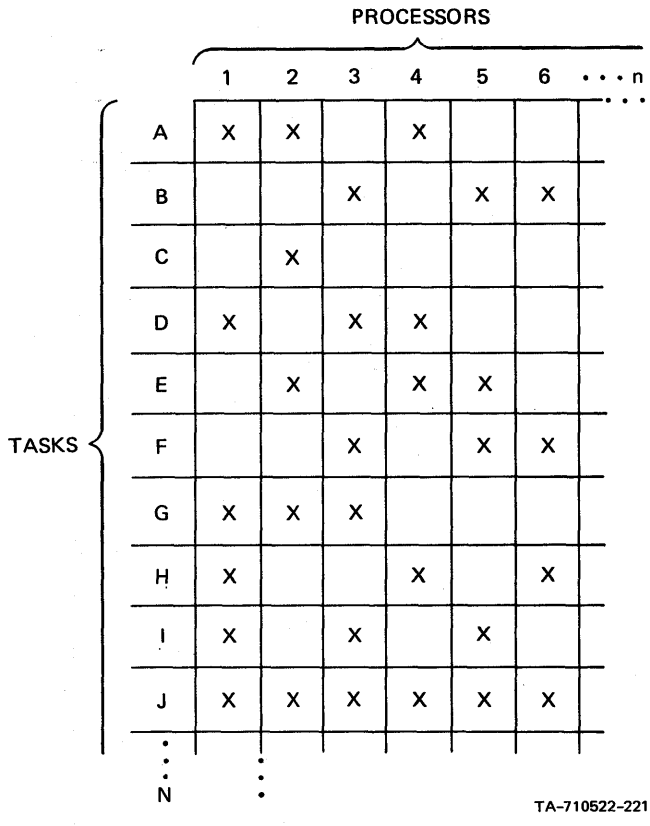

Figure 2—An example of task/processor allocation

possibly the same one carrying out the task). A check is made to see if the data are available in all processors. If not, the fact is noted in the memory of the module and the dispatcher program within the module is entered to determine the next task to be processed. The next processing is the reading of input data from the several processors where copies exist. A validation is now carried out, typically by a two-out-of-three vote. If any of the copies of the input data are found not to agree, then this fact is noted for later processing by the executive. If all the copies are different, the fact is noted and control moves to the dispatcher program. The computation of the task is now carried out, the results are left in the memory of the module, and note is made (in the module) of the fact that the task is computed.

Certain important principles apply in the above scheme:

- No processor writes into the memory of another module.
- Input data in a module are not destroyed during the computation: If the computation is repetitive, the results of one cycle that may be used as input for the next cycle are placed in a different location in memory. Similarly, because the input data within one module may be needed later by another processor carrying out the same task, the input data must not be destroyed until all cooperating processors have read, validated, and used the data. This may require that the data be preserved over several iterations if they are used by another task that is delayed behind the first.
- All conditions (e.g., errors, task complete) are left as notes to be read later by the system executive.
- The dispatcher program, which exists in each module, maintains a queue of tasks to be computed. The data for this queue are read from the memories of the modules that are running the executive. The flow of control of the dispatcher is

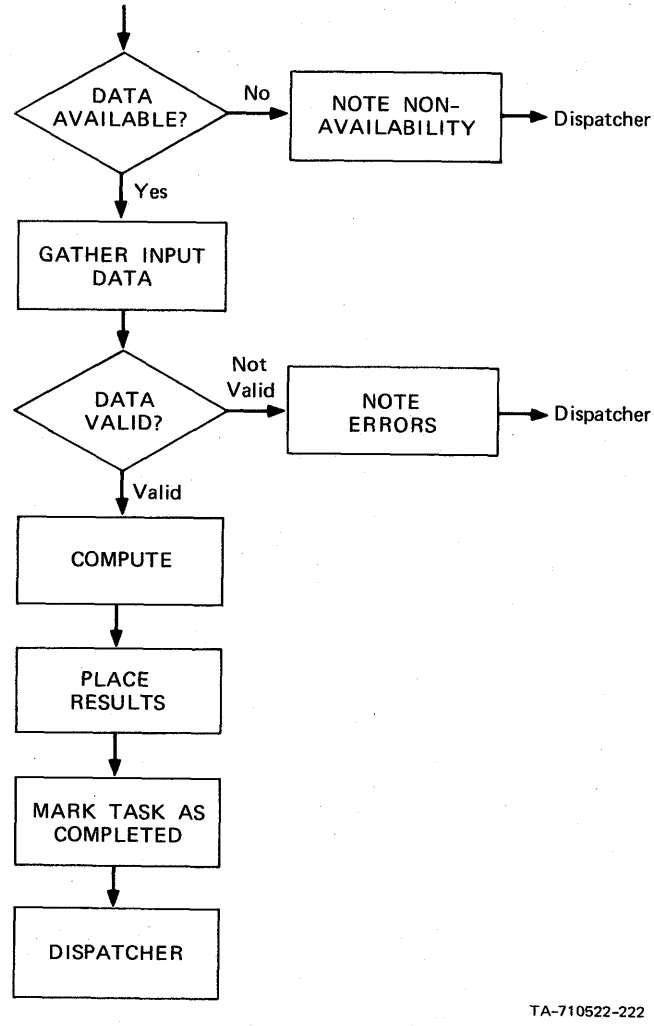

Figure 3—Typical task flow

itself similar to that shown in Figure 3, except at the end, when the control is transferred to the task that is at the head of the queue.
- The dispatcher in each processor checks from time to time to see if the system executive has changed the queue of tasks for that processor. A single bit (per processor) is set in the system executive tables to indicate a change of the queue. If this bit is not set, the dispatcher waits some time (e.g., 1 msec.) before querying it again, thereby preventing continuous interrogation and consequent heavy intermodule bus traffic.

The above scheme has a degree of fault tolerance without special hardware requirements on the memory or processor units. In particular, an erroneous calculation carried out by a module does not destroy the validity of the total system, because results are rejected by the next calculation.

The general strategy outlined above places certain constraints on hardware and software components. These constraints are discussed below.

INPUT/OUTPUT

The input/output subsystem must be designed and operated with the same fault tolerance as the central processing complex. Different modes of operation are possible, depending on the devices that are connected to and controlled by the system. The favored principle is to use replication wherever possible. Varying capabilities of fault tolerance in the central computing system can be achieved by using varying replication and by voting at all times when valid data are required (e.g., at the start of a task). The results of a calculation will exist in several (usually three) copies and eventually a vote must be taken. A vote that is required to allow another task calculation is carried out in multiple modules; however, if the vote is for output, then the output system or output unit must conduct the final vote.

There are circumstances where the nature of the input/output unit assists fault tolerance through replication, as in the following cases:

- Certain input systems (sensors) can be replicated; each sensor is then individually read (and voted on) by all modules requiring the input.
- Certain output devices can be built in a way that employs a "natural" kind of voting process in the final output medium. For example, a CRT display could be refreshed with each frame derived from a different module. Data on which all modules agreed would be displayed brightly; other data would be more faint. Assuming that faults persist only for short periods, this would result in a temporary flicker for a few frames before the executive removed the malfunctioning module from the calculation. In the application to which the design is aimed, there are other output devices, e.g., flap controls possessing similar "natural" voting capabilities.

In the event that the device is not in one of the above classes, another "final voter" must be designed that inherently possesses the required reliability. This consideration is independent of the architecture chosen for the central computing system.

We note that the architecture described here can operate in a mode whereby the replicated versions of output data (or the replicated data from input sensors) can be processed by any of the processing modules; hence, no modules need be specially designed for this function.

BUS DESIGN

The bus system (B_1, B_2, B_3 as in Figure 1) used for communication between modules must be designed to be fault tolerant. We remind readers that the bus system is used only to allow the processor of one module to read from the memory of a different module. The design need not be such that all bus traffic is checked (as in most other fault-tolerant architectures); however, it should allow a processor the choice of different busses in the event that a bus has failed.

A structure based on a four-port memory module is shown in Figure 1. In this structure, each module would have connection between its units (processor and memory). The bus structure, B_1, B_2, B_3, would enable a processor to choose different paths in reading data from the memory units of different modules. It would be appropriate to connect the I/O system to this bus structure. In the event that a four-port memory unit such as shown in Figure 1 is not available (or not suitable from other standpoints), then the structure can be achieved by attaching a single-port memory to all busses using conventional techniques.

A processor that needs to read from the memory of a different module must seize control of a bus. Logic associated with a bus must ensure that only one processor has control of a bus at any time. In addition, the bus must be allocated to a processor for only a finite time, thereby preventing a faulty processor

from seizing a bus permanently. An internal clock in each bus can control the period for which the processor in question dominates a bus. A failure in this control logic only causes the loss of that bus. It remains to be shown that no situations can occur where the failure of one unit can cause incorrect action of several other units, i.e., we require a design so that faults remain localized.

The interconnection of the units has only one purpose—to enable any processor to read data from any memory using any bus. The interconnection system does not allow a processor to write into other memories. A separate connection is assumed for a processor to write into its own memory.

In summary, the following sequence of action is carried out in reading data word (w) from memory (m) to processor (p) via bus (b).

(1) Processor p places b, m, and w in registers and signals all busses with a DATA REQUEST.
(2) All nonbusy busses scan all processor DATA REQUEST lines (continuously).
(3) If a data request line is on, and b equals the bus number, the bus goes into BUSY state and stops scanning the processors. The requested bus has now been selected by the processor.
(4) The selected bus transmits m, w, and DATA REQUEST from the processor registers to all memory modules.
(5) All nonbusy memory modules scan all busses for a DATA REQUEST line that is on, and then compares the m on that bus with its own number.
(6) If a match is found, the memory goes into BUSY state and ceases scanning the busses. The w on the bus is placed in the memory address register and a read request issued to the memory. The memory is now selected.
(7) When the word is read by the memory, it is placed on the data lines of the bus and a DATA READY line is turned on.
(8) The DATA READY and data are transmitted to the requesting processor. When the data have been received by the processor, the DATA REQUEST line from that processor is turned off.

(9) Action 8 above will cause the BUSY states (actions 3 and 6) to be dropped, and the bus and memory to resume scanning for other requests.

In the above sequence, each unit that requests action of another unit makes a request (e.g., DATA REQUEST). The granting of the request is made by the requestee. This arrangement, for example, will prevent a processor from requesting all the busses simultaneously as the busses will only respond if the bus request (b) agrees with their bus number. Therefore, it would require failure of all of the busses to completely disable the bus structure.

In addition to the above, it is assumed that each unit has logic associated with it that prevents it from being seized indefinitely. This logic, in effect, says "If I have been BUSY for greater than a time interval Δ, then the particular connection will be broken and scanning will be resumed for other units requiring action." It is possible to incorporate in this logic the capability to ignore requests from the offending requestor in the future, thereby removing that unit from affecting further system operation. The time interval, Δ, will be chosen to be just greater than the greatest time of any correct action request.

The word address (w) that is transmitted to the memory module can be subject to any transformation that is convenient in the design of the processor or

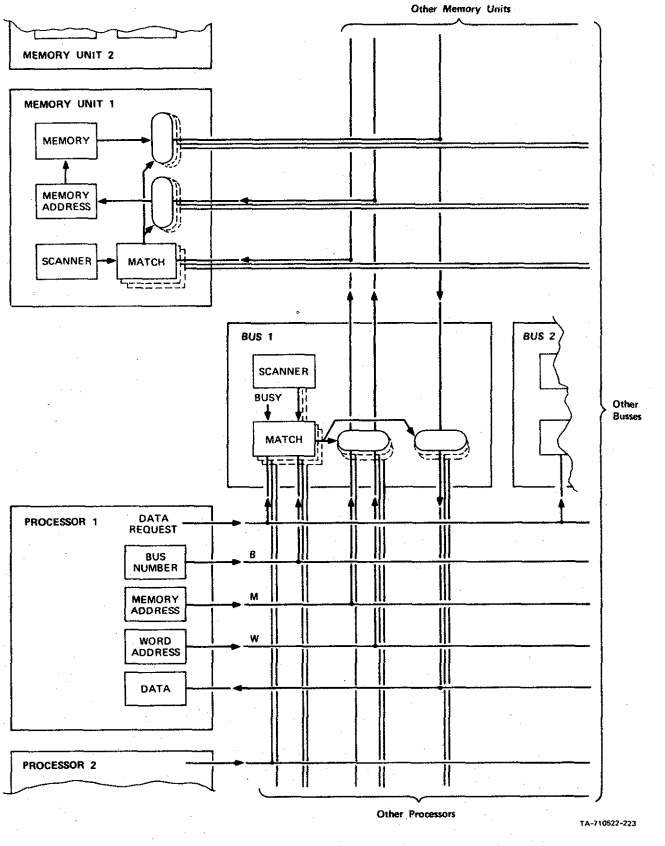

Figure 4—Processor/bus/memory connection

memory, i.e., it can include indirect addressing, indexing, base registers, paging, or any convenient combination of these. In addition, it is possible to incorporate a cache (in the IBM 360/85 sense) in the processor design.

The scheme outlined above obviates the need to provide a BURST MODE type of transmission as each word that is transferred can follow the sequence given. In the event that several words are required, the processor successively requests each word and the bus is seized and the word is delivered. If other processors require the use of the bus during the period of the multiple word transfer, a form of cycle stealing will take place as the bus scans the other units and honors the request before resuming scanning.

A suitable structure for the processor/bus/memory connection is shown in Figure 4.

THE SYSTEM EXECUTIVE

The system executive is concerned only with allocation of resources. All other special functions typically associated with an operating system (e.g., I/O control) would be treated as parts of the application program set.

It is expected that, in steady state, the executive would employ a simple scheduling algorithm to allocate resources. Exceptions to this would occur under two conditions:

(1) Change of task set to be computed
(2) Error conditions—either transient or permanent.

In both of the above cases it becomes necessary to reallocate resources. This task would not have to be carried out with high speed because, in the application considered, condition 1 above will be known in advance; condition 2 can be delayed for a short time because of the fault-tolerant procedure of replication and voting, which is carried out by the processors.

The executive system carries out any required synchronization. For example, the calculation required for advanced attitude and flutter control in certain aircraft must be carried out every four milliseconds. This calculation entails reading some sensors and then computing the new state variables using the old state variables and the input data. All modules assigned to this task have queues in their dispatcher task. The executive places a message in its memory for these processors to update their queues, whereupon the next calculation of this task is carried out by the several

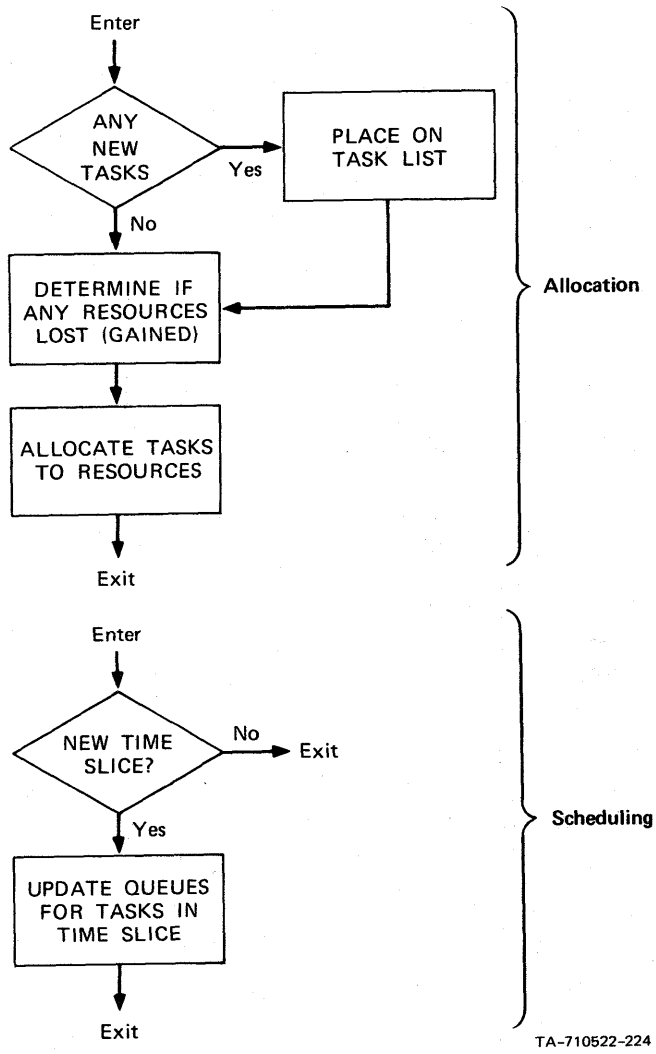

Figure 5—Executive

processors. When tasks are assigned to processors, the executive must designate the other cooperating processors so that all data required may be obtained. For the executive to carry out this synchronization, it must have a time reference that can be read by the processors or that causes an interrupt.

The calculations carried out by the executive are handled in exactly the same manner as other calculations (see Figure 3). A number of processors cooperate on this task, thereby providing fault tolerance when computing the executive. All processors within the system must know the designations of the several processors that are assigned to the executive. These data are held in the memory associated with the dispatcher that requires input data from the executive.

The flow of control for that part of the executive

concerned with allocation and scheduling is shown in Figure 5. This flow will be embedded as the actual calculation as shown in the fourth ("Compute") box of Figure 3. The allocation function is used to determine which tasks are to be computed and by which modules. It will be invoked relatively infrequently as it is required only when allocation changes are to be made. The scheduling function determines the time period during which any calculation is carried out.

FAULT-TOLERANCE PROCEDURES

By suitable design of the executive, the system architecture can carry out different fault-tolerance procedures for different requirements. The assumed fault-detection method is by comparison of multiple copies of data. This comparison is carried out by software imbedded in a system routine—a copy of which is present in all processors.

The first step in the computation of any task is to input the data required to carry out the task. This

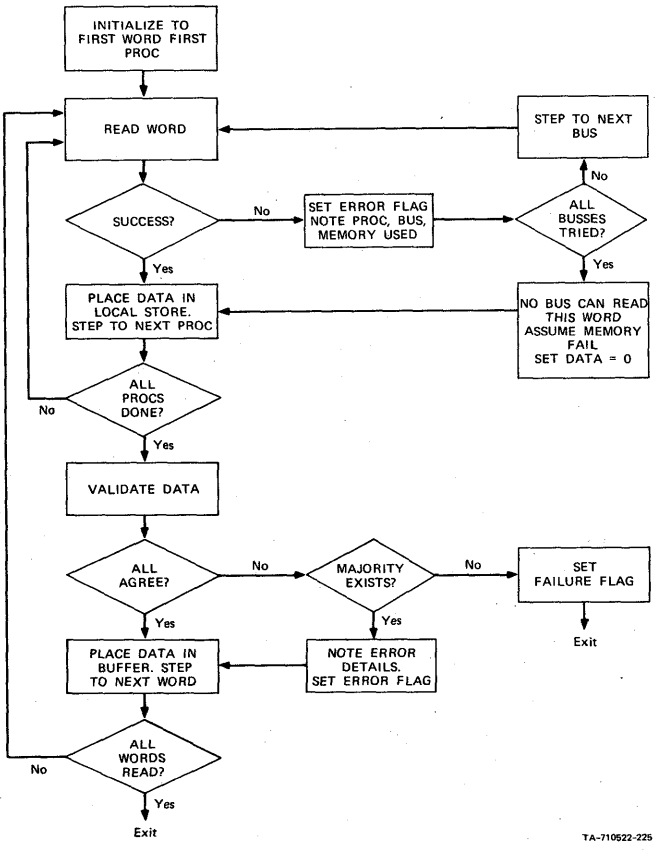

Figure 6—Input communication subroutine

data will exist in the memory of three computing modules. We will use the phrase "Input Data Set" (IDS) to denote the set of words required to carry out the calculation of a task. It is envisioned that all tasks requiring data will obtain it by calling a single subroutine. This subroutine is the only code (outside the executive) that is concerned with detecting errors; correcting them, in some cases; and in all cases, reporting errors to the executive. By the use of a single subroutine for error detection, avoidance, and reporting, the application programmer is relieved of the concern for this aspect of the system. This subroutine is shown in flow chart form in Figure 6. Its functional specification is explained in terms of the input parameters, output parameters, and the actions carried out.

Input parameters

IDS NUMBER	(The identification of which input data set is to be input.)
IDS SIZE	(The number of words to be input.)
BUFFER	(The address of the buffer in which the words are to be placed.)
PROC LIST	(The address of a list of processor numbers from which to input.)

Output parameters

FAILURE FLAG	(A boolean output variable, set = 1 if the input could not be accomplished.)
ERROR FLAG	(A boolean output variable, set = 1 if input was successful but an error was detected.)
ERROR VECTOR	(The specification of the IDS, word position, bus and memory involved in an erroneous input.)

Action

Read an input data set (IDS NUMBER) consisting of IDS SIZE words from the processor memories specified by PROC LIST. If all versions of each word obtained from the different processor memories agree, the data are placed in the memory at address BUFFER, the ERROR and FAILURE FLAG are set to 0, and a return is made to the calling program. If all versions of a word do not agree but a majority agreement exists, the data are placed in the BUFFER, the ERROR FLAG is set to 1, and the details of the

(presumed) erroneous input are placed in memory to be read later by the executive. If no agreement can be found, the ERROR and FAILURE FLAGS are set to 1, the data are not placed in the BUFFER, and a return is made. If no action can be accomplished (e.g., because of a faulty bus system) the units that are faulty are noted, and a return is made. The subroutine will attempt to use different busses for each word transferred. If no response is obtained from an input request, the subroutine steps to the next higher bus.

Fault detection by software voting is compatible with hardware fault-detection techniques such as parity schemes. Such hardware, if it exists in memories, busses or processors, can be used to assist detection and diagnosis of faults. The primary advantage of incorporation of hardware checking is to allow faster checking in the event that an application requires faster correction of fault conditions than can be achieved by software.

An important benefit in using software techniques for fault detection and tolerance is that freedom is retained to change the degree of fault tolerance, either because experience gives data on which better methods can be based, or because the different applications require different degrees of fault tolerance, i.e., some are more critical than others.

If threefold replication is used throughout the system, a single faulty unit will result in one of the replicated processes computing a wrong result. The use of the wrong result in subsequent calculations will be avoided by the fact that other (correct) copies of the data will exist in other modules and when used will, by voting, enable a processor to distinguish the correct data from that which is erroneous.

Consider now the case of double faults existing simultaneously. We must distinguish two cases, uncorrelated and correlated faults. By correlated faults, we mean two faults that cause the computation of two equal but incorrect results. Clearly, two correlated faults cannot be tolerated if the fault-tolerance procedure consists merely of voting among three versions of all results. The probability of such correlated faults will be extremely low and for most applications is acceptable. However, in the system as described, we can achieve greater reliability in the event that the application is so critical that this low probability is still unacceptable. Two strategies are:

(1) Use threefold replication for all critical applications, and in the event of *any* disagreement, do not use the results until yet further processors have carried out a repetition of the calculation, for example use two more processors (making a total of five) and only act if three or more agree.
(2) Use fivefold (or greater) replication* of tasks for all critical applications.

Both of the above strategies will prevent double correlated errors from causing the use of a wrong result in subsequent programs or output. The cost penalty in the above strategies implies that they will only be used for extremely critical applications, where the cost of extra computing equipment is small compared with the penalty for failure, e.g., in aircraft and space missions.

In the case of double uncorrelated faults, we need only consider the case of simultaneous faults. Double faults that occur separated by a time sufficient for the executive to have carried out corrective action after the first fault do not need to be regarded as different than two instances of single faults, which can be tolerated.

Two simultaneous but uncorrelated faults will have the possible effect of producing two different incorrect results from a calculation. These two results will be compared with the one correct result produced by the nonfaulty unit in a threefold replication scheme. Before the result is used in any subsequent calculation (or output), the presence of three differing results will be detected and the executive will initiate greater replication in other processors until sufficient agreement can be found to distinguish the correct from the incorrect result.

In considering the effect of multiple faults in the system, an improvement in reliability is achieved by the fact that the multiple processors are not operating in a lock-step mode. A short term, widespread transient in the system hardware (e.g., power supply or bus system) will not necessarily cause errors in the same application programs in the processors, thereby increasing the probability of being able to detect and correct the errors from the transient.

The executive of the system must itself be fault tolerant. This is achieved by the same techniques as for application programs. Each of the replicated copies of the executive will use data from itself and the other copies. In the event of errors in one of the executives, the other copies will not use the data computed by it, thereby keeping their results valid. The correctly functioning copies will initiate a new copy of the executive in another processor (which will entail

* This requires availability of a sufficient number of the various units (processors, memories, busses).

copying the program to that processor) and will signal the malfunctioning processor to discontinue processing the executive. In addition on inspection of the data in the correct copies of the executive all processors will cease referencing the data in the incorrect copy, thereby preventing a system breakdown in the event that the malfunctioning processor continues processing the executive even though requested to discontinue.

The fault-tolerant procedures outlined above can be summarized as follows:

- Given at least triple replication, all single faults can be tolerated, and all uncorrelated double faults can be detected.
- Given greater resources (memories, busses, and processors), multiple uncorrelated or correlated faults can be tolerated.

It is expected that, in the event of a permanent fault detected, a unit will be relieved of any active part in subsequent calculation. Therefore, the capacity of the system will be reduced; however, until a large fraction of the system is faulty, the fault-tolerance procedures can be continued without jeopardy. It is expected that with the use of LSI circuitry, the different units will be replicated manyfold (e.g., 10 of each unit); for less critical applications, fewer units will suffice. The removal of faulty units will be accomplished by allocating them to null tasks in the case of processors, and by not referencing them in the case of memories. The overall effect of these strategies is to achieve a graceful degradation either of computer capacity or fault tolerance, whichever is desired in the particular application.

CONCLUSIONS

A system architecture has been presented that achieves great flexibility in fault-tolerance procedures. The salient points of the design objectives that are achieved are:

- Fault tolerance can be varied so that for some tasks it can be arbitrarily high, using suitable replication and reconfiguration strategies; for other tasks, the fault tolerance can be less.
- No special design requirements are placed on the processing units or memories, thereby enabling different designs to achieve different computer power.

The basic feature of the system is that high level fault detection, avoidance, and correction functions are achieved by software procedures rather than by special hardware. The increased computer load caused by the software fault tolerant techniques has not been assessed fully at this time, but it is expected to represent a reasonable cost for the benefits gained.

The system is currently in the design stage with many problems still to be solved. Some of these problems are also present in other fault-tolerant architectures. Typical among these problems is that of finding ways to fragment the memory so that only a part rather than a whole memory unit needs to be reconfigured. Another problem concerns finding methods of checking units of the system that are not regularly used. Reconfiguration software (or hardware in other architectures) will only be invoked infrequently and may itself have been subject to damage during its period of quiescence. In the system as described, this reconfiguration is carried out by a program whose correctness could be verified from time to time by inspection carried out by a program that reads and compares the multiple copies.

REFERENCES

1 A AVIZIENIS
Design methods for fault-tolerant navigation computers
Technical Report 32-1409 Jet Propulsion Laboratory Pasadena California October 1969

2 A AVIZIENIS et al
The STAR (Self-Testing and Repairing) computer—An investigation of the theory and practice of fault-tolerant computer design
IEEE Trans Vol C-20 No 11 pp 1312-1321 November 1971

3 A AVIZIENIS
Arithmetic error codes: Cost and effectiveness studies for application in digital system design
Proceedings of Symposium on Fault Tolerant Computing Pasadena California March 1971

4 W G BOURICIUS et al
Investigations in the design of an automatically repaired computer
Digest of the First Annual IEEE Computer Conference Chicago Illinois September 1967

5 W C CARTER W G BOURICIUS
A survey of fault tolerant computer architecture and its evaluation
Computer Vol 4 No 1 pp 9-16 January 1971

6 W C CARTER et al
Logic design for dynamic and interactive recovery
Proceeding of Symposium on Fault Tolerant Computing Pasadena California March 1971

7 W C CARTER P R SCHNEIDER
Design of dynamically checked computers
Proceedings of IFIPS 1968 Congress Edinburgh Scotland August 1968

8 R S ENTNER
Presentation of advanced avionic digital computer baseline definition

Naval Air Systems Command Washington D C September 1969

9 J GOLDBERG K N LEVITT R A SHORT
Techniques for the realization of ultra-reliable spaceborne computers
Final Report Phase 1 Contract NAS12-33 SRI Project 5580 Stanford Research Institute Menlo Park California September 1966

10 J GOLDBERG et al
Techniques for the realization of ultra-reliable spaceborne computers
Final Report Contract NAS12-33 SRI Project 5580 Stanford Research Institute Menlo Park California June 1969

11 A L HOPKINS JR
A fault tolerant information processing concept for space vehicles
IEEE Trans Vol C-20 No 11 pp 1394-1403 November 1971

12 L J KOCZELA
A three-failure-tolerant computer system
IEEE Trans Vol C-20 No 11 pp 1389-1393 November 1971

13 G Y WANG
An in-house experimental aerospace multiprocessor—EXAM
ERC Memo KC-T-031 NASA Electronics Research Center Cambridge Massachusetts September 1967

14 G Y WANG
System design of a multiprocessor organization
Memorandum RC-T−179 NASA Electronics Research Center Cambridge Massachusetts 1969