FAULT TOLERANT COMPUTING: ISSUES, EXAMPLES, AND METHODOLOGY

David A. Rennels

June 1987
CSD-870024

FAULT-TOLERANT COMPUTING: ISSUES, EXAMPLES, AND METHODOLOGY

LECTURE NOTES
ADVANCED FAULT-TOLERANT COMPUTING WORKSHOP
BANGALORE, INDIA
JULY 20-25, 1987

David A. Rennels
University of California
Los Angeles, CA 90024, USA

Abstract

These lectures are intended to discuss a selection of fault-tolerant computers, with emphasis on issues and tradeoffs in the design of unmaintained systems which must provide dependable real-time operation over long-periods of time. Current research issues are discussed along with the problems of developing and validating new high performance systems. The notes are compiled from previously published survey and research papers presented by this author. The information is modified and rearranged for tutorial purposes and augmented with additional presentation slides.

These lectures are presented in four parts outlined below:

1. *Requirements and Approaches in Existing Fault-Tolerant Systems*

   The first part begins with a brief history of fault-tolerant computing. Classes of fault-tolerance applications are identified along with their corresponding requirements. Existing fault-tolerant computers are discussed, emphasizing the basic concepts employed in the design of these systems. Tradeoffs and alternatives are examined which are available to the system designer in attempting to meet applications requirements.

2. *Real-Time Systems for Long-life Unmaintained Applications*

   The second part discusses computers for long-life unmaintained applications, focusing on previously published research into fault- tolerant designs for planetary spacecraft. The architectures of the JPL STAR (Self-Testing and Repairing) Computer, the Unified Data System, and the Fault-Tolerant Building Block Computer are discussed.

3. *Fault-Tolerance Issues in Next Generation Real-Time Systems*

   The third part of these lectures discuss, from the prospective of this author, problems of implementing fault tolerance in the next generation of high performance highly parallel computers. Emphasis is placed on long-life unmaintained systems which have real-time recovery requirements. Several current research areas are discussed including: i) application of Fault Tolerance to very high performance parallel architectures, and ii) self- checking, self-exercising logic.

4. *Methodology of Implementing Fault-Tolerant Systems*

   The final part of these lectures deals with the methodology of implementing fault-tolerant systems. This is a difficult problem from both the standpoints of management and engineering. It involves i) specifying requirements, ii) evaluating contractor performance, iii) validating the design.

1. REQUIREMENTS AND APPROACHES IN FAULT-TOLERANT SYSTEMS

1.1 HISTORY OF FAULT TOLERANT COMPUTERS

Until the 1980's most work in fault-tolerance was directed at providing recovery from physical faults, manifested either as transient errors, intermittent faults or component failures. Many of the design techniques for hardware fault-tolerance were developed for early relay and vacuum-tube machines. Since they were prone to

error, techniques such as error detection codes, instruction retry, and diagnostics were used (usually in an ad-hoc fashion) to help these machines compute correctly most of the time. It was not viewed as practical to produce a computer which continued computation in the presence of major errors and faults which are hard to handle, e.g. failures in control, transient faults, etc. There was one notable exception in Prague, where a team led by Prof. Antonin Svoboda developed a computer (SAPO) with comprehensive fault detection and recovery capabilities. Due to very poor quality of available components, (and, as he often described it, threats from the ruling authorities) he found it necessary to build three processors and vote their results in order to circumvent frequent relay errors to achieve acceptable performance [OBLO 62].

With the development of transistor machines, component reliability was greatly improved. Automated fault recovery was not viewed as being cost-effective since redundant hardware and fault recovery mechanisms would add considerably to the initial cost of the system. Emphasis was placed on minimizing downtime when a fault occurred by including fault detection mechanisms in the design and providing comprehensive diagnostics to allow a repairman to quickly isolate faulty logic and replace it.

Research into circuit testing and fault diagnosis is probably the oldest active area in the fault tolerance field. By the mid 1960s effective algorithms had been implemented for testing combinational circuits, (Roth's D algorithm being perhaps the best known [ROTH 67]). The use of microprogramming allowed diagnostic programs to be written in microcode, giving much improved access to internal logic for more effective testing. By the mid 1960s extensive fault-detection logic, retries of single instructions, and microdiagnostics were in use in many (e.g. IBM) mainframes.

In the 1960s several applications arose for which the potential cost of computer failure was very high, and automatic fault recovery was needed so that computations could continue uninterrupted in the presence of faults. Among these applications were computer-controlled spacecraft (which cost several hundred million dollars each) and computer controlled telephone switching systems. These were probably the first requirements for fully fault tolerant systems, and they caused a new advance in the state of the art. In addition to the previously developed functions of fault detection and diagnosis, these systems were required to automate the recovery process.

It was understood from the outset that there were only a few basic conceptual approaches to implementing hardware fault tolerance. Redundancy could be implemented statically in a structural fashion where neighboring units took over the function of failed units much like an extra column in a building. Alternatively, an active redundancy approach could be taken in which faulty units were detected and spares substituted in their place. Static redundancy took the form of either "quadded" components connected in series-parallel (groups of four components could operate correctly with one failed) or triplicated modules with majority voting to ignore a single faulty unit in the three (called TMR-Triple Modular Redundancy). Active redundancy required the ability to explicitly detect errors in a unit and a fault- tolerant recovery mechanism to replace a faulty unit with a spare and to re-establish computations.

Two early fault-tolerant space computers were built and flown in the Orbiting Astronomical Observatory (OAO) satellite and the Apollo guidance system, and a fault tolerant Electronic Switching System computer (ESS) was developed for telephone switching. The OAO computer used static quad redundancy shown schematically in Figure 1a. It was one of the last computers built with discrete transistors, and its approach to fault tolerance was unique to that technology. Each transistor was replaced by four in a series parallel circuit and was designed in such a way that if any transistor failed, the others could continue to provide the proper logic function [KUEH 69]. This approach could not be continued with integrated circuits, because independence of failure could no longer be guaranteed. With all four transistors on the same piece of silicon, a fault could damage all of them.

The next machine, the Apollo guidance computer, used static redundancy for the processor and active redundancy for the memories (design techniques which are still useful with modern technology). There were three processors running the same programs, and their results were voted to mask out an error by any single processor. Two memories were used and data was written to both of them and encoded in an error detecting code (see Figure 1b). If one of the memories failed, its data would be incorrectly coded, and the processors would then use data from the other memory [ANDE67].

The initial ESS designs used a different approach to active redundancy. Two computers executed the same programs and their outputs were compared (Figure 1c). If one failed they would disagree, and diagnostics were executed to find which machine was faulty [TOYW 78]. Later designs use two processors each of which is specially designed to detect its own internal faults so that the faulty machine can usually identify itself when the computers disagree.

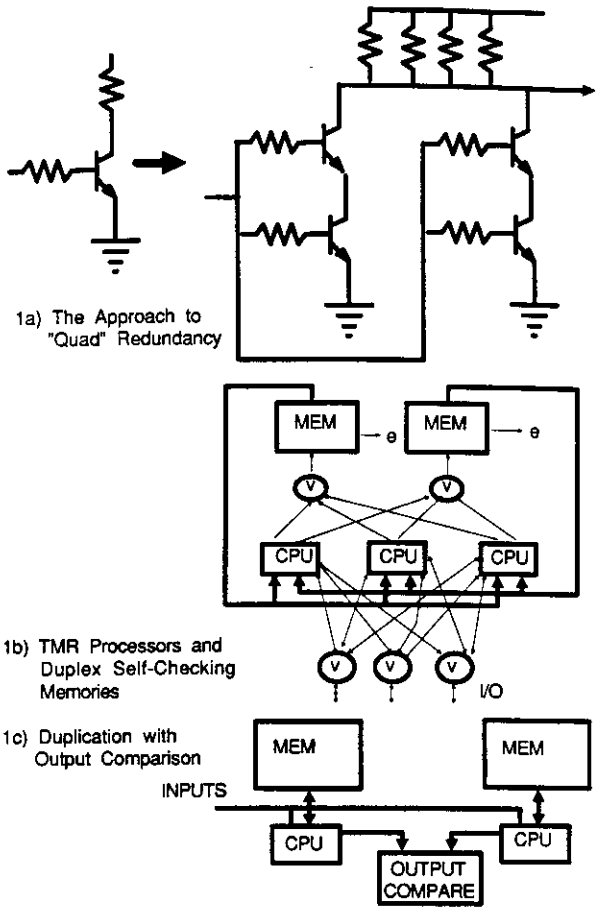


Figure 1: Early Fault-Tolerance Approaches

The JPL-STAR Computer (shown in Figure 2) was developed for long unmaintained life during deep space missions during this period [AVIZ 71a]. It used a fourth approach of active redundancy. The computer was subdivided into functional units: Memory Modules (MM), an Arithmetic Processor (ARP), a COntrol Processor (COP), an I/O Processor (IOP) and a Test And Repair Processor (TARP). Each unit was designed to detect and signal its own internal faults concurrent with regular program execution. The triplicated Test and Repair Processor was a simple module which served as a hard-core recovery unit. It was charged with replacing a faulty unit with a spare and reinitializing the system so that computations could continue. The TARP introduced a new hybrid form of dynamic and static redundancy. Three TARPs operated simultaneously and their outputs were voted. Backup spare TARPs were also provided and two agreeing TARPs could replace a disagreeing TARP with a spare. This architecture was finely subdivided into small functional units, and in most cases, only one unit of each type was powered at a time. This "standby replacement" system was chosen because of very limited power availability on the spacecraft and the need for very long unattended life.

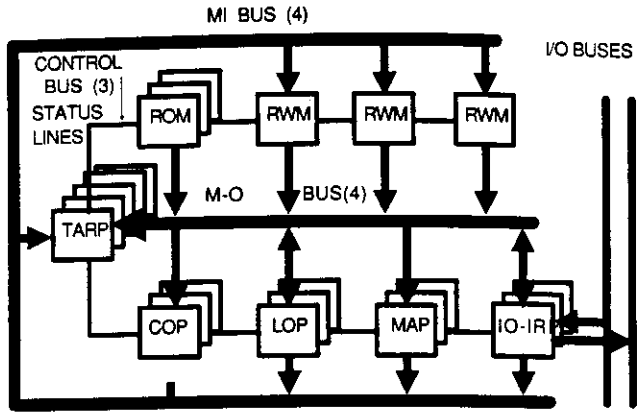


Figure 2: The JPL STAR Computer

By the end of the 1960s nearly all of the basic forms of fault- tolerant architecture to be found in later designs had been built and experimented with (e.g. triplication with voting, duplication and comparison, self-checking units, and backup sparing). These concepts were refined and adapted to more modern hardware and software technology in subsequent computers.

In the 1970s, several fault-tolerant machines were developed for commercial aircraft control. Two very advanced research machines were developed to the same specifications by the same NASA sponsor, and were built and tested as prototypes (the Fault-Tolerant Multiprocessor (FTMP) and Software Implemented Fault-Tolerance (SIFT). Simplified block diagrams of the two architectures are shown in Figures 3 and 4. Both systems execute three copies of a program in different hardware and vote the results to mask faults but they do it in very different ways. All processors in the FTMP are clock synchronized and voting is done by hardware. Processors in SIFT use independent clocks, and voting and synchronization are carried out by software.

In the FTMP structure, a set of processors and memories are connected to five redundant buses through special redundant bus guardian circuits. Processors and memory modules can be dynamically assigned to be a member of a group of three processors and three memories which will run the same computation (designated a triad). This is done by commanding their associated bus guardians to communicate over specially assigned buses. The guardian circuits in the processors vote the three copies of data arriving from their assigned memories, and conversely the memories' guardians vote on information from their assigned processors. If a bus, processor, or memory fails there will still be two valid copies of information at each voter, and the fault will be masked, allowing the triad to continue. When such a failure occurs, a different triad can sense the condition and reconfigure the affected triad by sending commands to bus guardians to assign a new processor, memory, or bus to the affected triad. In this system the common clock had to be specially protected since its failure would disable everything. A special redundant hardware clock design was developed which is immune to single-point failures [HOPK 78].

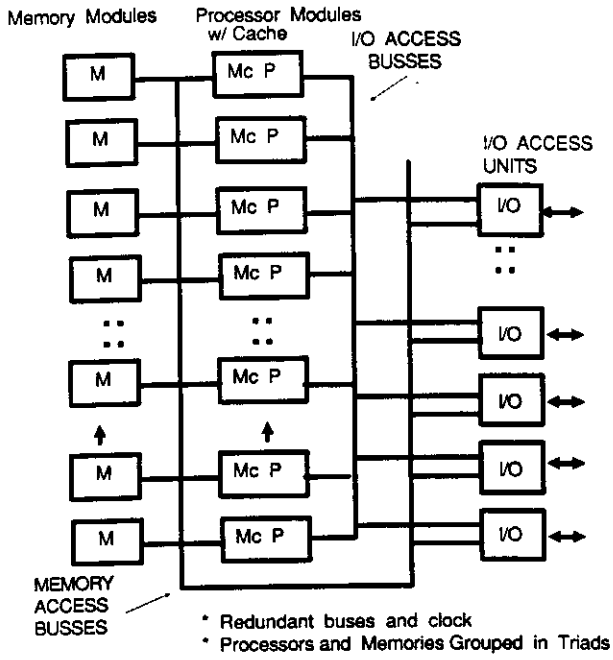


Figure 3: The FTMP Structure

The SIFT computers are totally connected. Each computer can broadcast a message over a serial line to dedicated buffers in all the other computers. The computers operate with unsynchronized hardware clocks, and synchronization occurs by a software voting process. Each computer contains a synchronous software executive and software voting procedures. Periodically, the computers exchange messages containing their views of the time and develop a consensus as to its value. As user processes are scheduled in a time-synchronous fashion, they are executed at approximately (but not exactly) the same time. They send their results to the other processors where a software voting procedure is invoked to mask faults. If a computer fails and generates disagreeing outputs, the other two ignore it [WENS 78].

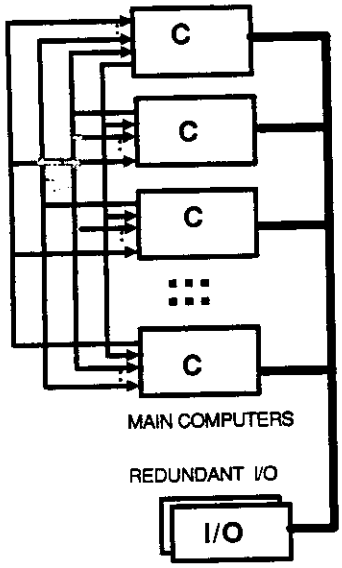


Figure 4: The SIFT Structure

In practice the FTMP architecture has two major advantages over SIFT in dedicated real-time control applications. It runs faster than SIFT because its voting is done by hardware. The SIFT computers use a significant percentage of their processing time running the software voting and synchronization programs. More importantly, the fault-tolerance features of FTMP are nearly software transparent. Nearly any software executive can be run on FTMP with fault recovery procedures written to run under it. (Remember the triads will continue to operate under fault conditions until a reconfiguration procedure is invoked.) SIFT, on the other hand, is constrained to using its custom synchronous executive which implements the fault-tolerance features.

In this authors opinion, FTMP may have been more practical, but SIFT has probably made an equally important research contribution. The SIFT program has done pioneering work in proving the correctness of software, the use of formal specifications, developing new software voting and synchronization concepts, and allowing for the use of selective redundancy. In the SIFT system the processors can schedule different (non-redundant) programs in three machines some of the time while running highly protected triplicated programs at other times. This type of approach has considerable merit in commercial systems where resources can be assigned individually to non-critical tasks, then brought into a redundant configuration for more important uses (such as periodically waking up a fault-tolerant monitoring and system recovery process).

A highly redundant computing system was used on the Space Shuttle [SKLA 76]. This system has five redundant computers. During critical mission phases, four execute identical programs and the control outputs of the four are voted in the control actuators. The fifth is used as a non-redundant backup and contains totally different programs in the hope that if an uncorrectable design or software error occurs in the four primary machines, the fifth can be switched-in to replace them. Two sets of programs have been written by different contractors (the four are programmed by IBM and the fifth by Rockwell). Incompatibilities between the two sets of programs have caused check-out problems and a delayed launch, but the system now appears to be working satisfactorily.

Also during the 1970s more advanced fault tolerant computers were developed for communication switching, and the use of fault-tolerant computing expanded into process control for critical transportation applications [IHAR 78].

Recently, two parallel developments have accelerated application of fault-tolerant computing. The first is the increased degree to which the public depends upon computing systems, greatly multiplying the inconvenience and public awareness of an occasional computer fault. A failure of computing for automated bank tellers for a large city one or two times a year, or temporary outages of credit card checks can

inconvenience thousands, as can failure of just one computerized ignition system on a freeway at rush hour. The second is the enormous decrease in the cost of computer hardware. The primary cost associated with implementing hardware- fault tolerance is associated with redundancy that is added to provide fault-detection and recovery. Although the relative hardware cost of a highly fault-tolerant computer may be several times that of a non-fault tolerant machine, hardware prices have dropped an even greater relative amount making fault-tolerance cost- effective for a larger number of applications.

A number of new companies have been formed to enter the commercial marketplace with fault-tolerance. The best known is Tandem Computers which has grown to a large size within a very short time by supplying this technology to the transaction industry [KATZ 82, BARL 82]. Other companies (August Systems, Stratus, Synapse, etc.) have since been formed to enter this expanding marketplace and have met with varying success.

A new and exciting area is the development of fault-tolerant local area networks. A large amount of redundancy is naturally available in collections of identical machines, and the individual machines have the built-in intelligence (with properly written programs) to provide sophisticated recovery algorithms at very low cost when other machines fail. It is instructive to briefly discuss two examples, Locus and the Draper Laboratories AIPS systems.

LOCUS is a network operating system which exploits the hardware redundancy in a local network to provide very high availability. It is running on a distributed VAX network at UCLA and provides a network-transparent UNIX environment. The user can log onto any machine in the system, except when restricted by administrative fiat, and see an identical environment (passwords, files, mail, etc.) Files can be duplicated at two different sites, and, as programs execute, data is paged out to those sites. If the user's machine crashes, he or she connects to a different machine and continues using the system [POPE 81]. This system is based on the use of commercial hardware with little or no hardware modifications.

AIPS is an extension of the FTMP concept to distributed systems as shown in Figure 5. A group of processing sites are connected through switching nodes to a redundant intercommunication structure which behaves like a triply redundant bus. Redundant links are provided, and messages can be circuit switched over different paths to provide physical damage tolerance. Each processing site may be a Fault Tolerant Multiprocessor (FTMP), a triplicated (TMR) Fault Tolerant Processor FTP, a duplicated pair of processors, or a single non-redundant machine. Each site has a local clock which synchronizes computers at that site, but clocks are not synchronized between sites. Hardware voting is done throughout the system. Within a site containing triplicated (TMR) processors, voting is straightforward because the processors are clock-synchronized and are executing identical programs. Voting of triplicated data sent between processing sites with different local clocks requires a hardware synchronization operation, but the data skew can probably be kept small and hardware voting is still feasible. This design recognizes the need for selective redundancy. In a complex system, not all processes are sufficiently critical to justify triplicating their processors. Thus duplex and single processors can also be included [AIPS 84]. This system is a candidate for use on the Space Station.

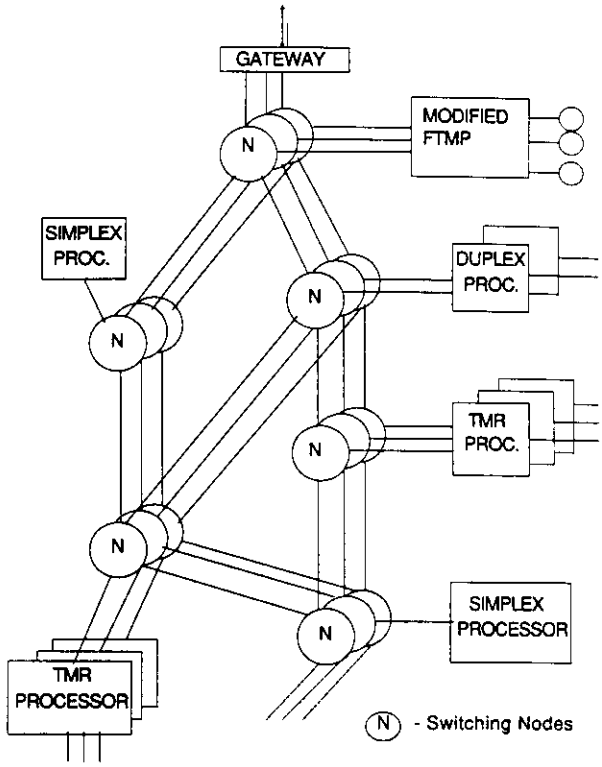


Figure 5: A Simplified Diagram of the AIPS System

1.2 FAULT-TOLERANCE PERFORMANCE

Ideally, all fault-tolerant computing systems should recover from all possible faults with no errors or delay. Practical systems can only achieve limited capabilities. We cannot enumerate all possible faults, and economic tradeoffs further limit what capabilities are designed into a system. In order to comparatively evaluate fault-tolerant computers it is necessary to establish a way to compare the requirements to which they are designed and the performance they achieve. All fault-tolerant systems have an objective of improved availability, i.e. to minimize downtime. In addition, three principal characteristics which define a space of requirements for fault-tolerant design as shown in Figure 6. In the figure they are shown as three axes which define (at least in an informal way) a space in which fault-tolerant designs can be compared.

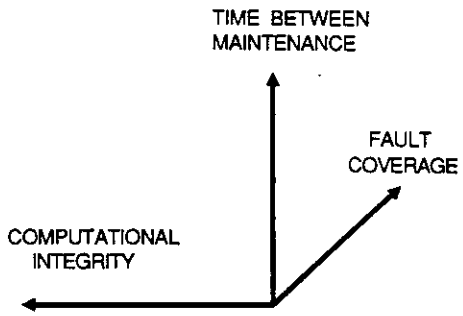


Figure 6: Fault-Tolerance Characteristics

Fault Coverage

The first axis is fault coverage. Coverage has a qualitative component: i) which fault-types are covered, and a quantitative component ii) how well the protection mechanisms work in recovering from each fault-type. For every fault-type (i) we define a quantitative coverage parameter c(i) which is the conditional probability that the system will recover properly, given that a fault of that type occurs. We can define an overall average system coverage as an average of the coverages for the associated fault- types weighted by their probability of occurrence.

For any fault-tolerant design it is necessary to enumerate the fault-types that are being protected against. One group is physical faults, which should include transient errors (sometimes called single event upsets) and permanent faults associated with the component technology used. There exist a number of standard models for predicting the rate of occurrence of random faults which cover existing technologies and introduce approximations to take into account chip complexity and, to some extent, new device types. For new technologies the system developer must estimate failure types and rates by extrapolating from past experience and/or developing failure physics models of new devices.

Another class of faults which may be specified are externally induced errors caused from the surrounding environment such as electrical noise, radiation, or physical damage. A third class of faults are caused by design errors (logic errors, software errors).

Coverage has been shown to be a very sensitive parameter in achieving fault tolerance over long periods of time [BOUR 69]. This can be easily seen since if N faults occur, the probability of correct operation will be bounded by c**N, no matter how much redundant hardware is employed in the system.

As a system requirement, the level of coverage reflects the criticality of the application to which the computer is applied and what levels of performance people are willing to accept. The SIFT and FTMP computers were designed to a requirement that there be less that one chance in a billion of computer failure during any a ten hour aircraft flight (between ground maintenance). Since there is a reasonable chance of a fault occurring within that time, coverage was required to be a nearly perfect (.999999..). Other applications, such as spacecraft control may require a smaller value of coverage (e.g. >.98-.99) if a computer outage is not as critical. The surrounding system may be designed with interlocks which would prevent computing errors or loss of computations from destroying the spacecraft, and remote ground intervention may be possible to manually reconfigure the on-board computer to a working configuration should that be necessary on rare occasion. The users of some general purpose systems might be satisfied with reducing the average crash rate from once a month to once a year allowing a coverage of .93.

The following paragraphs address approaches which can be taken to achieve coverage against random hardware errors and faults.

Triplicated Processing for High Coverage:

Extremely high levels of coverage against random faults in hardware are very difficult to achieve. To design for coverage of 0.999999 would require knowing with near absolute certainty which faults will occur in practice. This is clearly impossible, so the designers use the only alternative available. Three computers are operated simultaneously and their outputs voted. Here we do not have to know how a computer will fail, it is assumed that no two computers will fail at the same time, and if one fails, the correct result will appear in the output vote.

But even using a conceptually simple approach of voting replicated units, the fault-tolerant implementation is difficult. A number of complex issues arise in the design of the "hard Core" portion of these fault tolerant systems. That hard core consists of common elements which can result in system failure, e.g. voters, synchronizing clocks, and communication paths between the machines. The FTMP and SIFT programs worked out highly redundant clocks, and voters which are inherently fault tolerant. FTMP did this in hardware while SIFT uses distributed software algorithms to provide the same functions. Both systems use redundant interconnections between computers with mechanisms to circumvent faulty paths [HOPK 78, WENS 78].

Duplicated Processing with Concurrent Fault Detection

There are less hardware expensive ways to provide recovery from hardware faults -- at a correspondingly lower value of coverage. It was shown that a conventional computer could be made self-checking at a cost of about an additional 10% in hardware. With self-checking design (discussed later in these notes) each computer has internal error detection circuits and can signal an internal error as soon as it occurs. Clearly two machines of this type running the same computation could provide fault-tolerance with high computational integrity using two conventional machines

and comparing their outputs. If they disagree, the computer signaling an error would be ignored and the other one would complete the computation. This would be considerably less expensive than running three machines and voting. The coverage is lower because the internal checking circuits may not detect all errors, and if two processors disagree, and neither signals a fault, the one in error may be selected to continue the computations.

Simplex Processing with Backup Spares

An even less expensive approach is to use standby redundancy -- to operate each program in a single computer which has concurrent fault-detection. Backup spare computers are available to take over should a permanent fault occur in the primary machine. This results in a delay or loss of some computations if an error or fault occurs and also results in much more complex recovery algorithms (described below). This approach can be expected to have lower coverage than either the use of massive voted redundancy or the use of duplicated self-checking processors.

Computational Integrity

The next axis is computational integrity. This has several components: i) accuracy of computation, ii) time delays during recovery when computation is not available, and iii) definition and protection of critical state information. The following addresses approaches which can be taken to achieve computational integrity against random hardware errors and faults.

In order to achieve complete computational integrity (without time delays for fault recovery) it is necessary to carry out at least two self-checked computations or three non-self-checked computations concurrently. (This includes replication of all storage.) If a fault occurs in one machine carrying out the computation, another can continue it without delay. This is hardware-expensive and has in the past been reserved for computers used in critical applications such as airframe control (e.g. the SIFT, FTMP, and Space Shuttle computers discussed above.)

Only recently, has the price of hardware dropped to the point where this level of redundancy can be introduced into commercial machines. The Stratus Computer system uses a pair of self-checking processors, running the same software concurrently. At a price of about $100 per chip, this system includes eight 68000 processors running as four self-checking pairs.

For many applications, computational integrity requirements can be relaxed, allowing computing to stop for a time during error and fault recovery, and allowing some computations to be lost. These relaxed requirements allow the use of less expensive standby redundancy. Each computation is run in a single machine which is backed up by other computers running different computations or serving as unpowered spares. The machine's state is periodically saved as rollback points. If a transient fault occurs, the computer returns to the last rollback point to restart its computations. For a permanent fault, provisions may be added for a different computer to obtain the rollback state and continue the interrupted computations. In either case the computation is delayed.

In the JPL-STAR computer (which as previously described used standby redundancy) the recovery delay could take hundreds of milliseconds. This may not be a problem in some commercial applications, but for real-time systems the worst-case recovery time must be accounted for in the design of the surrounding system. For example, a spacecraft computer might be scheduled to start a sequence of periodic commands to a subsystem during the time that a fault recovery was in progress (or even worse it might fail when some, but not all commands have been sent). After the recovery the applications programs would be required to account for the fact that a delay occurred, and the command generation program would have to send the command later. This might cause other commands to be modified to related subsystems, etc.

A standby redundant system cannot guarantee computational integrity if inputs are lost or if outputs are forgotten during a program rollback. It is seldom recognized that special hardware must be included to capture any input which occurred between the time that a rollback point was established and a fault induced a rollback to that point. Clearly if inputs are ignored the computational results will be modified by the occurrence of a fault. Similarly, outputs can not be repeated when a rollback is attempted.

To summarize, a high degree of computational integrity can be maintained when expensive fault masking techniques are employed (TMR/Hybrid or duplex self checking). Time delays are introduced with the less expensive standby redundancy and, if careful design is not employed, computational integrity cannot be guaranteed. Probably the best solution is the selective use of redundancy. If non-critical programs are run on single machines, while occasional critical programs are triplicated, a cost effective solution can be obtained. A system is currently being developed at UCLA which will cause critical computations to be run in triplicate on a distributed computing facility, while the majority of (non-critical) programs will be run in a standby redundant fashion [AVIZ 84].

Time Between Scheduled Maintenance

The third axis of Figure 6 is Time Between Scheduled Maintenance. Maintenance in this context is human intervention to replace faulty components. A fault-tolerant machine will recover from a fault by invoking redundant hardware, but the faulty hardware must be replaced to restore the system since the system will ultimately fail when spare hardware is exhausted. There are a number of applications for which maintenance is impossible (in this case time between

scheduled maintenance is the total life of the system), and others for which maintenance is inconvenient or expensive (resulting in months to years during which the system should operate without manual repair). For example, many space computers are unrepairable. Many military systems have severe logistic problems for repair, as do systems at remote places. This requires special design of long-life systems with sufficient redundancy to survive an extended period of time with a specified level of confidence. Often these systems are also constrained by limited power availability and a requirement to be fitted into a small volume.

Long life systems must employ more redundancy than regularly maintained systems, and must use the redundancy that they have in an efficient fashion. This can be in conflict with the need for computational integrity and high coverage which, as we have seen, requires massive redundancy during operation. For example, a long life system which requires three operating processors must provide much more additional redundancy to guarantee that all three processors will survive until end of life. This is infeasible in space applications due to power constraints and the fact that many semiconductor technologies (NMOS, Bipolar) have a higher failure rate when powered. (A long-life system may be designed to operate with only one copy of a computation being run at one time in order to require fewer spares and leave the spare hardware unpowered for longer life). Some coverage and computational integrity is sacrificed to attain longer life with a given amount of redundant hardware.

In order to use redundancy in an efficient fashion, long-life computers are often partitioned more finely than are frequently maintained systems, because this allows longer life to be achieved with a given amount of redundant hardware. For example, the FTBBC computer developed for unmanned spacecraft at the Jet Propulsion Laboratory, allows faulty computers to be replaced with spares. In addition, each individual computer contains internal redundancy which allows individual memory modules, processors, and even memory chips to be replaced with spares to enhance the life of each processor at a small increase in hardware cost [RENN 81b].

1.3 APPLICATIONS CLASSES

Critical Applications

For highly critical applications where the loss of human life or expensive machinery can occur, it is likely that massive voting redundancy will always be used. By running three or more machines and comparing their outputs, it is not necessary to know the hardware fault mechanisms which occur within a module. The only assumption required is that they will fail randomly and independently. If only one machine fails at a time, an output vote will deliver a correct result, and very high coverages can be assumed. The guarantee of no computational delays during fault recovery is often very important. Uncertainty as to interruption of computations can complicate design of the system which the computer may be expected to control - sometimes with unpredictable effects. Thus machines which use fault masking, such as SIFT, and FTMP are the types best suited to these applications.

Long-Life Applications

For systems with long periods of time between scheduled maintanence, standby redundancy is attractive where limited power is available or where the failure rates of unpowered spares can be shown to be lower than the failure rate of powered units [AVIZ 71a]. But this conflicts with the fact that coverage is lower than that available in voted systems, and inadequate fault coverage may limit the expected life of the system. With inadequate coverage, it does no good to add additional spares to extend life because a high probability will exist that the fault-recovery mechanisms will fail.

In distributed systems, the selective use of both voting and standby redundancy may offer the best solution to this problem. A small hard core portion of a system is configured using hybrid redundancy to run critical processes and fault management functions. The remainder of the system uses less expensive standby redundancy depending upon the core to supply high-coverage fault recovery to the whole system.

These long-life systems should display implementations which are finely partitioned (e.g. the use of spare bits in memory, spare microprocessor chips, and redundant I/O circuits) inside of individual computers in order to squeeze the maximum useful life out of a given amount of redundant hardware. This was done in the SCCM [RENN 81b].

Commercial Applications

Until recently, commercial systems compromised fault coverage due to an assumption that customers would not pay for redundant processor units. For example, most current systems are designed to handle easily detectable processor and memory faults, but subtle transients can go by undetected because the processor is essentially non-redundant. In the future, we can expect to see customers become more demanding of fault-tolerance. In this environment we see increasing costs of maintenance calls and interrupted service while the cost of hardware is going down rapidly. The emergence of systems such as Tandem and Stratus systems have shown that fault-tolerance can be achieved at acceptable costs and, as people depend more heavily on these computers for banking and commerce, this market will continue to grow. The availability of cheap single chip processors allows duplication or triplication of processors for detection of all transient faults and very high fault coverage (at least for medium performance machines). This has been recognized and implemented in the Stratus system [FRIE 82].

Institutional Computer Networks

Nearly all large computing systems in the future will be distributed processors due to the simple fact that processors have become remarkably inexpensive. Many current systems use networks of time-shared super minicomputers (e.g. VAXs at universities). Since their hardware is usually off-the-shelf and relatively expensive, hardware fault-detection capabilities are limited and partial fault-tolerance is implemented in software. One of the better systems of this type, LOCUS, provides a distributed UNIX environment with a high degree of transparency. As described previously, a user can log on from any machine and have access to identical services and data. Files can be automatically maintained in the secondary storage of two machines so that if one machine fails, the user can log onto a different processor and resume computations [POPE 81]. These networks rely upon a unified naming and file management scheme which is both redundant and consistent across the network.

The SIFT results point toward an approach which is can be used to supply "hard" fault tolerance in such a network in a cost effective fashion. It should be possible to schedule and run selected programs in three different machines, voting their results. This can allow crash-free and error-free results for critical programs while allowing most of the computations in the network to be scheduled on single machines. A capability of this type is currently under development at UCLA [AVIZ 84].

But a second revolution is beginning in systems of this type. While many institutions pride themselves on the number of time-shared super minicomputers tied together into a network, there is a move afoot toward collections of microcomputer workstations which often give greater processing power to the user than the "slice" he would get from a larger machine -- and at considerably lower cost. For example, it has been estimated that the annual cost of maintenance of the VAX network at UCLA will be higher than the hardware cost of replacing it with workstations of similar performance (although the cost of tranferring software would probably make such a change unfeasible). Some shared facilities will still be needed, and they are expected to evolve toward specialized servers which provide services not available in workstations. Large storage systems, high-performance numerical processors, and data base machines will be needed as the specialized servers within large networks of workstations. These unique nodes will be depended upon by multiple users and therefore need considerable fault-tolerance. Thus research into fault-tolerant mass memory systems, special purpose number crunchers, and data base processors should gain additional impetus.

Ultra High Performance Processing

There exist a class of dedicated system applications (e.g. signal processing) which need special purpose processing functions to be carried out at enormous speeds. Often the desired processing rates are in the range of billions of multiplies per second. In most of these applications, the value of individual measurements is very small due to the large number being taken, and an occasional error in an individual measurement is not very important, but system availability should be high. Here the designer is pressed to use nearly all of the available VLSI chip area to increase speed, and the use of silicon real estate for fault-tolerance is viewed as highly expensive. This is a controversial area because fault-tolerance specialists feel that the VLSI area costs of concurrent fault detection and other fault-tolerance features is justified. Users often do not.

A promising approach is to implement fault-detection locally within the high-performance system and to rely upon external control computers (which are slow and highly fault-tolerant) to effect diagnosis and recovery within the high-performance front-end processors. Proposed fault detection and isolation mechanisms have included (1) insertion of "dummy" calibration data into the input stream and having an external computer identify this calibration data in the processed output to certify correct operation, (2) using a "roving" duplex processor as a checker which duplicates the function of each of several active modules (in a pre-planned sequence and compares outputs to verify correct operation, and (3) using low-cost error detecting codes in all arithmetic. In any case, it appears that a hierarchic fault-tolerance approach is needed where special purpose, very high-performance parallel processors are maintained by simpler and much more fault tolerant external processors. Applicable fault tolerance techniques will be better understood when VLSI layouts for these systems are completed, and a data base of applications is better established.

1.4 HARDWARE FAULT TOLERANCE CONCEPTS

This section is a review of some of the basic concepts and design issues in the implementation of fault-tolerant distributed systems. System development usually proceeds from a set of user specifications (including fault tolerance requirements) to an architecture concept. The architecture is then refined and partitioned into subsystems, each with specific functions to provide. The subsystems are then further partitioned and protective redundancy is added in the form of fault detection and recovery mechanisms.

a) Hierarchic System Partitioning

Complex systems are naturally partitioned at several levels based on functions provided by specific subsystems. A fault-tolerant system displays similar functional partitioning, but in addition it contains redundant components and recovery mechanisms which may be employed in different ways at different levels. It is reasonable to view a fault-tolerant system as a nested set of machines (subsystems) each of which may display varying levels of fault- tolerance. For example, at the highest level, a distributed system may recover from a failed computer by shifting its computations to other machines. At the next lower level, a single computer may be capable of replacing a faulty memory module with a spare and switching to alternative communication channels to circumvent a failed port, but not be able to recover when a short occurs on the local memory-processor bus or when its power supply fails. At a lower level, the memory modules may be capable of replacing defective RAM chips with spares, or the chips may contain redundancy and be capable of tolerating certain failures but not others.

A descriptive model must capture this hierarchic property of systems and the way in which fault-tolerance features are apportioned within the various levels. To do this we will introduce the terminology of embedded partitions. A Redundant Partition (RP) is a set of modules which contain sufficient redundancy that if one fails, acceptable performance can be achieved with those remaining. (The RP may contain spare modules to replace those which fail, or it may redistribute functions when one fails and operate in a degraded fashion.) Recovery from a fault within the RP may be effected within the partition itself, or it may require require action by higher levels within the system. An RP may be made up of heterogeneous modules (e.g. a computer with its communications ports and disks backed up by a similar computer which has different disks and I/O facilities), and a module in one RP will often contain other nested RPs. One example is a redundant set of computers, where each computer contains redundant memory modules, processors, and I/O circuits.) Figure 7 shows a typical system which is partitioned into several levels.

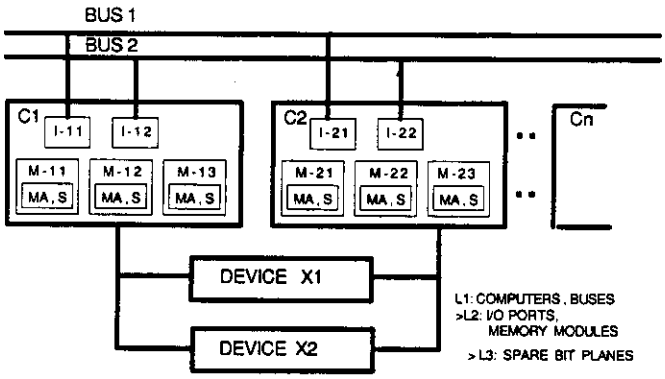


Figure 7: Nested Redundant Partitions

In Figure 7, the highest level partition (RP) is the intercommunications bus and its backup spare. Typically, the computers are grouped into several independent partitions by their dedicated interconnections to external devices. Computers 1 and 2 with devices x1 and x2 form a redundant partition and, if no other computers have devices similar to x1 and x2 and therefore cannot back up their services, they are the only computers in the partition. The computers contain two nested redundant partitions; memory modules, and I/O ports, and the memory modules contain nested partitions of spare bit planes (ma-memory array, s-spare).

Associated with each module is a fault-tolerance interface which goes along with its functional interface. The fault-tolerance interface communicates with modules in its own partition or higher level partitions to identify fault conditions and local recovery actions, and it may request recovery services that cannot be provided locally [RENN 81a, ANDE 81]. For example, if a memory chip fails, its memory interface unit (the next highest level) detects this through a fault-tolerance (FT) interface and may switch in a spare bit plane and reconstruct damaged data automatically. If a memory module fails, its FT interface must notify its computer of the condition, and if the local computer cannot recover from the fault locally, its FT interface must notify other computers. Other computers may be required to diagnose the fault, activate a new memory module, reload and restart the failed computer. More will be said about FT interfaces below.

Figure 8 is an idealization of the fault-tolerance interface needed between modules of a three level distributed system. This type of hierarchic computer architecture is already beginning to appear in spacecraft and probably in industrial automation also. The lowest level (but often the most expensive) redundant partitions are at the electromechanical sensors and actuators for which spares are provided. At the next level, redundant computers are dedicated to specific subsystems (e.g. attitude control on a spacecraft), and at the next level, a set of redundant computers provide high-level executive and control functions.

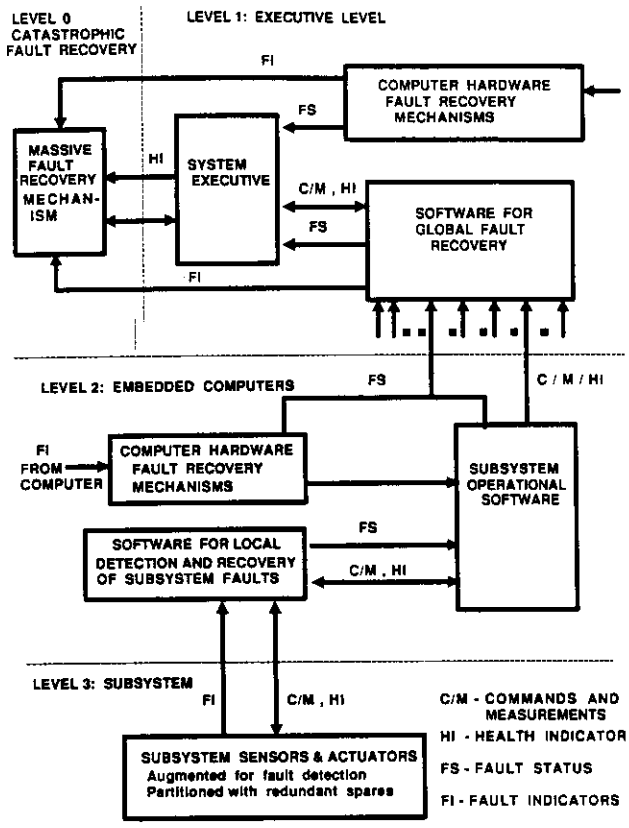


Figure 8: Fault Tolerance Interfaces in a Recovery Hierarchy

The subsystem-embedded computers provide a degree of fault- tolerance for their associated electromechanical subsystems using software which can detect anomalies and effect recovery within the subsystem. The subsystem-embedded computers also contain redundant partitions and be capable of locally recovering from a number of internal computer faults. The system executive computers provide system-level recovery functions. If a subsystem is unable to recover from a local error or fault or if it temporarily ceases to operate while a component is being replaced, system-wide recovery implications can result. For example a spacecraft may have to re- acquire orientation in space, or a command sequence may have to be modified. The system-level computers must also provide fault-tolerance for themselves and be capable of assisting the subsystem computers in local recovery.

Each level can be characterized by its possible response to various internal faults. The two primary categories are faults that the level is designed to handle locally, and faults beyond its range of coverage. If one level recognizes a fault for which it was properly designed, it can notify higher system levels in one of three ways via its FT interface: (1) it can indicate that the fault was recovered locally, (2) request specific help in recovering the fault within its local level, or (3) notify higher levels of a failure which cannot be recovered at its level. Notifying higher levels of faults may be implicit or explicit. If the lower level module provides outputs encoded in an error detection code, or if it is one of a set of duplicated or triplicated modules whose outputs can be easily compared by the next higher level we view this as an implicit error indication. A status signal would be viewed as an explicit indicator.

Much more difficult are faults which were overlooked in the design at the partition at which it occurred. In this case, no explicit notification will be forthcoming when the fault occurs. Here the higher level must employ reasonableness checks and acceptance tests on the functional outputs of the module. This is shown in the figure as "health indicators". Each subsystem should be designed so that its functioning can be checked for reasonableness. For example, data may be specified whose rate of change cannot exceed some specified threshold, calibration data may be included within real measurements, or dummy procedures may be executed to check data for consistency. Control flow checks (locks and keys on procedure entry) and time-out-counters also fall into this category. This heuristic checking can be used to provide a second line of defense against faults which may not be detected at the level where they occurred.

For many dedicated systems reasonable checks of this type can be quite powerful. For example, the attitude control subsystems of many spacecraft execute complicated differential equations to keep the system pointed to the the Sun and a star. If sensors indicate loss of pointing or excessive loss of control gas, a fault in this subsystem is nearly certain to have occurred whether or not it was detected locally within the subsystem.

Reasonableness checks, though useful in detecting that a subsystem is no longer functioning, provide little information about the cause of failure. Fault recovery at this level can require expert systems. A recovery algorithm may involve substituting redundant elements (and possibly activating redundant software modules) in a sequence planned to optimize the chances of restoring correct operation based on educated guesses as to what might be wrong. Research is being conducted into on-board systems which attempt to restore correct operation, but go so far as to re-plan mission operational sequences if only partial operation can be restored.

Several distributed systems using commercial machines have very limited fault detection capability within the computers. Thus the higher level systems functions often employ heuristics of this type for fault detection such as heartbeat checks and checking of time counts in the other machines.

Multi-level models of fault tolerance have not yet been well developed, but this is the way that technology is moving. The state of the art in mathematical models to predict the reliability of fault-tolerant systems are based on the use of Markov models whose number of states become unmanageable when multiple levels of redundancy are employed [NGYW 80]. The development of performance and reliability models for future multi-level fault-tolerant systems is a new and

badly needed area of research. The extension of fault-tolerant computing to supply autonomous repair to complex systems is also a relatively new research area which will have increasing importance in the future [AREN 83].

b) Fault Detection

If a module fails, a resulting error must be detected to enable a recovery mechanism to take corrective action. This is the process of fault detection. Modules at all levels (computers, logic modules, or on-chip redundancy) fall between two basic types. At one extreme are circuits which can detect internal faults concurrently with normal operation -- these we will call "self checking"(SC), and at the other extreme are modules which have no internal fault detection capability which will be designated non-self-checking (NSC). When used in a redundant partition, SC modules can be operated singly, since faults will be detected. On the other hand, NSC modules must be at least duplicated and operate two-at-a-time with outputs compared for fault detection. They must operate three-at-a-time and voted if a faulty module is to be identified quickly (or if transient faults are to be located).

A methodology for designing self-checking logic has been developed, and it has been shown that a self-checking computer can be developed at an approximate 10% increase in hardware complexity [CART 77]. The reason for this relatively low cost is that the majority of a computer's logic is memory which, due to its regular structure, can be designed to detect faults with a few extra bits per word. Irregular logic must often be internally duplicated and compared for concurrent fault detection, but this makes up a small percentage of many modern machines. An important characteristic of this methodology is the fact that self-checking checkers have been developed which signal faults in the checking circuitry as well as in the operational circuits being continually checked [CART 72]. This largely solves the problem of "who checks the checker?". Checking signals are implemented as complementary "morphic" pairs which alternate between values 1,0 and 0,1 when no error exists. Upon detecting an error in the circuits being checked or in the checking circuits, these signals take on values 1,1 or 0,0 indicating a fault has occurred. A reduction circuit was developed so that a number of these complementary pairs from individual checkers can be reduced to a single self-checked pair which serves as a master fault indicator. A useful reference on self-checking circuits is Wakerly [WAKE 78].

One approach to implementing self checking logic on VLSI is to duplicate circuits on the chip, with one copy implementing the required function and the other implementing the same function but with complementary logic functions (every logical "one" in one copy would be a logical "zero" in the other). If these circuits are error free, their corresponding outputs will be complementary pairs, and a self-checking comparison can be performed [SEDM 80].

The JPL Fault-Tolerant Building-Block Computer architecture (described in the next lecture) was designed to use a small set of VLSI building-block circuits to interconnect existing microprocessor and memory chips to form Self-Checking Computer Modules (SCCM). The SCCMs contain redundant communications interfaces to facilitate their use in fault-tolerant distributed systems on spacecraft. Self-checking (morphic) logic design is used throughout the SCCM design. Odd parity checks are split into two halves covering portions of words being checked to generate complementary signals, and random logic is duplicated with the outputs of one member of a duplex pair inverted to also generate complementary output signals. These are reduced (using morphic "and" logic to provide self-checking check circuits. The SCCM is shown in Figure 9. The four building-block types are (1) a Memory Interface building block which implements Hamming Codes and spare chip replacement in memory, (2) an I/O building block, (3) a Bus Interface building block which allows the SCCMs to be connected with similar SCCMs into a network, and (4) a Core building block which compares the outputs of two duplicated processors, checks information on internal buses for proper coding, and collects fault messages from other building blocks. The Core, on detecting a fault can initiate a program rollback to correct transient faults, and disable the SCCM if the fault persists --indicating a permanent fault. A breadboard system was constructed, and faults were inserted into both the operational logic and the check circuits by shorting randomly selected wires to ground [RENN 81b].

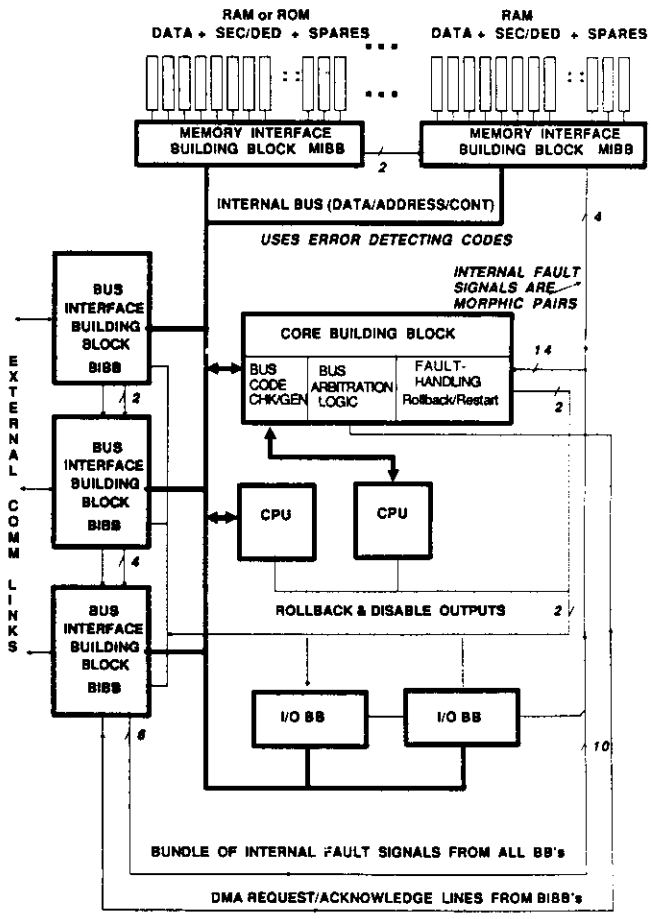


Figure 9: A Self-Checking Computer Module (SCCM)

Possibly a more practical approach was taken by INTEL in some of their chips for the 432 system, which are designed to be duplicated and compared. Every chip contains a set of output comparators, and a control line indicating whether the chip is to serve as an active chip generating outputs, or whether it is to serve as a checker. Two chips are tied together with one designated the active unit and the other designated the checker. Both receive the same inputs and compute the same logic functions but the checker chip does not output. It blocks its outputs and compares them with what the other chip generated, signaling an error if a disagreement occurred [INTE 81]. In a highly competitive commercial environment, few vendors will sacrifice performance of their chips by devoting a large amount of chip area to concurrent error checking. Intel's comparator approach costs little in real estate or performance on the chip, and it gives the user the option to buy two if concurrent fault detection is to be implemented.

In many systems circuit modules may fall somewhere in between SC and NSC and be partially self-checking. Many commercial computers are used in (at least partially) fault tolerant distributed systems which have internal detection of some (e.g. memory parity) faults while other internal faults cannot be detected.

We will characterize the checking of modules in a recovery partition by two parameters -- fault detection coverage, and detection latency time. The fault detection coverage for a given detection procedure is the probability, given that a fault occurs, that it is detected by applying the procedure. The detection latency is the time between the occurrence of the fault and the time that the fault is detected through observations of erroneous results. We simplify this parameter by giving it three values, I (instantaneous), EC (error-concurrent) and NC (non-concurrent). Instantaneous fault detection implies that a fault is detected when a fault first occurs, and is impractical to achieve since many faults do not cause signals to take on incorrect values (i.e. errors) until the system reaches particular states. For example, a stuck-at-zero cell in memory causes no error while the stored word has a zero in the faulty bit position. An error only occurs when a "one" is stored there. Error-concurrent fault detection (concurrent is the term used by Avizienis and others) implies that a fault is detected before a fault-induced logic error travels beyond its point of origin. In many cases this requires that an error be detected within the same clock cycle that it first appears, (although in some designs this can be relaxed to a few cycles), and before damaged data leaves the module. An NSC module, having non-concurrent detection, may take many cycles to detect a fault, and massive error propagation can be expected before the fault is detected.

Self-checking modules exhibit concurrent, but not instantaneous fault detection. An interesting potential area of research is the design of self-checking and self-testing logic which not only detects fault-induced errors, but also flushes out faults within a guaranteed time by exercising, during normal operation, all logic states necessary for faults to cause detectable logic errors. Considerable research has been carried out on test-set generation with hundreds of papers to be found in the literature [BREU 76]. Recent research has focused on generating tests and checking responses directly on VLSI chip, e.g. one approach partitions the chip into small enough logic groups that exhaustive testing can be carried out [BOZO 80]. (Other approaches have been proposed using probabilistic testing, generating pseudorandom test sequences and signature analysis on chip but the coverage of such tests are very hard to determine.)

This area can integrate the separate specialties of VLSI logic synthesis (e.g. state assignment) design for test and fault-tolerant architecture. If chips are made to detect their own internal faults, and can also be made to perform self-tests during normal operation (by interleaving test cycles with operational cycles, or better yet, exercising circuits that are temporarily idle) it may be practical to build circuits which thoroughly test themselves in a few seconds of normal operation. An interesting question is how to mechanize state saving on-chip with little area penalty to allow interleaving of testing and normal operation.

c) Fault Recovery using SC and NSC Modules

There are three common ways of providing fault tolerance within a redundant set of modules: triplication and voting, duplication and comparison; and standby replacement.

To achieve comprehensive fault tolerance using non self- checking modules it is necessary to provide fault detection and recovery by massive redundancy techniques. Three NSC modules may be operated concurrently and their outputs voted to detect and mask out a fault in one. This approach is commonly known as triply modular redundancy (TMR). When spares modules can be used to replace the faulty units in a TMR set, the configuration is designated as hybrid redundancy. TMR and hybrid architectures were previously described in the discussions of the SIFT, FTMP and Apollo systems. Alternatively two NSC modules be operated together and their outputs compared to detect faults. Upon detecting a fault, both are commanded to run diagnostics and the one which succeeds will be deemed operational. The early ESS processors, used duplicated processors. This technique, designated duplex redundancy (DR) sometimes fails for transient faults since both modules are likely to pass the diagnostic and it may be impossible to tell which contains fault-damaged data. When a computer fails in a TMR system, the system reverts to duplex operation, and when a duplex system fails it is often possible to find the good unit of the pair and continue non-fault tolerant operation with a single machine (NMR/Simplex).

If a module is capable of detecting its internal faults, it is possible to operate only one unit and, upon detecting a failure turn it off and replace it with a spare or assign its function to some other module. This approach is known as standby redundancy (SR). This form of

redundancy was used in the JPL STAR and FTBBC computers, because it uses minimal power and is well suited to long unattended life [RENN 78b].

The three approaches are shown symbolically in Figure 10 for the simple case where a single module is redundantly protected.

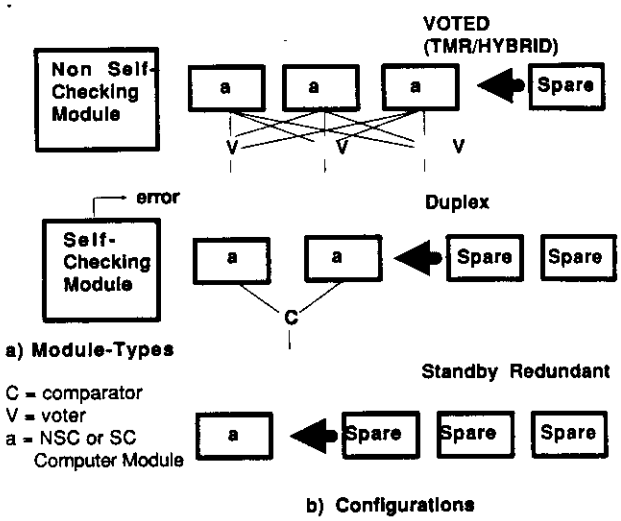


Figure 10: TMR/Hybrid, Duplex, and Standby Redundancy

From Figure 10, it is easy to see that standby redundancy has a major advantage for long life unmaintained systems (if unpowered equipment has a lower failure rate) because more of the total hardware is available as spares. It has a disadvantage of lower fault coverage than the voted configuration, which can detect and recover from any single fault within one module. The Self-Checking module in a SR configuration must make use of internal error detection techniques which may not be capable of detecting certain obscure multiple error conditions such as those induced by clock and power supply noise.

Duplex and standby redundant configurations must make use of program rollback techniques. Since computations may not be recoverable from the point at which a fault was detected, it is necessary to save the system state in the form of rollback points in programs and, after detection and removal of a fault, return to the most recent rollback point to recover computations. The fault- tolerance community developed this technique for use in non-voted computers [ROHR 73]. Especially difficult is the problem of non- repeatable events where a computer may generate an output to external systems and then generate it again after rolling back. A similar problem of this type occurs if a file is advanced by a record, a fault is detected, and it is advanced again during the rollback as previous code is repeated. Solutions to these problems have been developed by the software community under the more general category of automic transactions and stable storage [LAMP 81]. The approach is to make it possible to back out of I/O operations if an error is detected and a rollback or restart initiated. When a series of external events are requested, none are carried out until it is assured that the computer has safely progressed to the next rollback point. The Unified Data System system, for example, buffered all outputs and executed them all at once when the next rollback segment was reached (as indicated by a real time interrupt [RENN 76].

Remapping Processes to Redundant Hardware

In distributed systems TMR, Duplex, and Standby redundancy may be used at different sites. AIPS, discussed in section one allows all three types to be used in different places. TMR/Hybrid redundancy may be used for critical processes, while standby redundancy is used for less critical applications, thus freeing up more processors for increased throughput.

In multiple processor systems the fault recovery process is complicated by the process of remapping the computations of failed processors onto different elements which remain functional (examples are shown in Figure 11). For small systems, this remapping can be relatively simple. For example, the JPL-STAR computer was divided into a set of functional units, each of which had dedicated backup spares which could be switched-in to replace a faulty unit (Figure 11a). SIFT and FTMP are homogeneous structures which allow any spare module of a given type to be substituted for any active module of similar type. If spares are available, the "next" spare is substituted for any module of its type which fails.

When a system is designed to operate in a degraded fashion if no spare elements are available, the remapping process becomes more difficult and sometimes involves complex decisions (figure 11b). It is necessary to select computational services which may be delayed (slowed down) or dropped altogether, and for dedicated control systems entirely new operating modes may have to be developed.

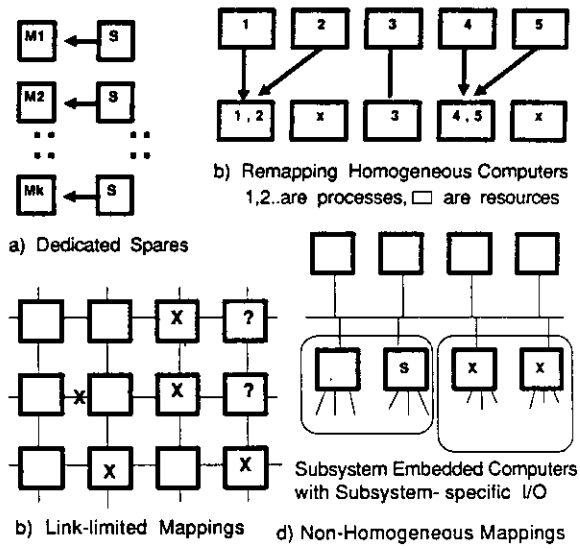


Figure 11: Remapping Processes to Redundant Hardware

This process of remapping becomes most complex when systems contain large numbers of processors. Large networks 1) have limited interconnections between processors and 2) are often heterogeneous. Figure 11c shows a processing network in which the failure of links can make processors unavailable, and the failures of processors can make links unavailable. It is desirable to limit the number of communications links to an individual processor for economic reasons, but the interconnection structure must be as redundant as possible to minimize losses when a failure occurs. Several researchers have applied graph theory to study the general problem of fault diagnosis and the remapping of processes in large connection-limited networks. Hayes has analyzed the problem by specifying processes to be carried out and physical resources as separate graphs, and has considered the problem of designing networks which can tolerate some number of failures and still be capable of carrying out the required computations. This is equivalent to finding an isomorphism between the process and resource graphs in the presence of failures [HAYE 76]. It may be possible to use the automated search techniques of languages such as Prolog to enumerate all alternative assignments of processes and communications in partially failed networks which still satisfy the needed performance requirements. A considerable amount of work has also been done on the problem of diagnosing faults in large collections of machines[PREP 67, RUSS 75]. Kuhl and Reddy have developed techniques by which processors in a large unsynchronized network can diagnose failures using distributed algorithms and come to a mutual agreement on the availability of resources[KUHL 80].

A number of researchers have studied specific network structures in an attempt to optimize this fault-induced remapping process for performance, various technology constraints, and long- life availability [GREY 84, RAGH 82].

In systems with a large number of resources, the concept of reliability is no longer a simple question of does it work or not. Various degraded levels of performance are to be expected and evaluation must be made on the basis of both reliability and performance. A new area of performability modeling has been developed which attempts to evaluate the functionality to be expected from a system, considering the fact that failures will occur at various times during its life [MEYE 81]. This type of modeling can provide comparative evaluation of design approaches in terms of a users objectives -- extra services gained for a fault- tolerance investment.

Finally, to achieve fault tolerance in very large systems, new distributed operating systems concepts will be needed. Functional languages and high-level data flow concepts will apply.

d) The Need for Independence of Failures

In order for most fault-tolerant designs to work properly, random component faults must occur independently in the modules into which a system has been partitioned. There are a variety of techniques for detecting a fault in one module and effecting recovery (either by suppressing its outputs using voting, or by disconnecting it using switching), but few if any existing designs can deal with the occurrence of a fault which affects more than one module simultaneously. This need to localize a fault to only one module leads to very stringent design requirements which may be from a minimum of placing modules of a redundant partition on different chips to encapsulating each module in shielded boxes with ground and power isolation.

A subtle problem of "lurking faults" is encountered. Two faults may occur independently, and at different times, in different modules that may result in simultaneously occurring errors for a particular common data input. This occurrence of simultaneous errors under these circumstances may appear to be simultaneous faults in two modules. It is possible for a fault to occur in logic which is seldom used, and thus not cause a detectable error until the faulty logic is exercised. If an undetected fault occurs in one module and lurks there until a fault in a different module causes an unusual set of computational states which causes the first fault to generate an error, then errors will appear as if the modules had experienced dependent faults. A new requirement occurs to fully exercise and test each module periodically in order to flush out lurking faults and preserve independence. This area was described before as offering interesting research opportunities because three often disjoint disciplines are involved for optimal solutions -- combining architecture, design of fault-detecting logic, and design of circuit testing procedures. Research in this area will be discussed in lecture 3.

1.5 DESIGN FAULTS: A FUNDAMENTAL LIMIT ON FAULT-TOLERANCE?

Anumber of fault tolerant systems have been demonstrated which can deal with random faults, but these systems have little or no capabilities to handle design faults. Examples of faults not covered include:

(1) semiconductor processing faults such as contamination which might make all chips fail after a period of time,

(2) logic design faults in a processor or other hardware --the most subtle ones which cause all redundant copies to make the same mistake when a rarely occuring data or control state is reached are the most difficult to deal with,

(3) software faults - in most systems software errors cause faults more often than hardware faults and, as above, the subtle ones are difficult to test for

since they may only occur when an unusual set of inputs or rare timing relationships exist.

Current systems use extensive testing to attempt to eliminate design faults, and in some cases correctness proofs have been carried out for limited portions of software.

Process Verification

To eliminate processing faults military and space programs spend millions of dollars in detailed inspection of semiconductor parts. There are rigorous and expensive parts screening procedures defined as military standards which must be carried out before parts are qualified for use in many systems [MILSTD]. Both government and privately sponsored research is being conducted into development of test circuits for VLSI wafers and into analysis of circuit failures which have been found. The designer who cannot afford military quality parts is wise to use well established parts whose bugs have already been found through widespread use. Even then it is advisable to get parts from different processing batches for redundant circuit elements so as to avoid correlated failures. An infinite amount of redundancy will be unsuccessful if a system is constructed from components which have a build-in wear out mechanism.

Hardware Testing and Functional Verification

Most current systems make use of extensive tests for hardware to try to uncover any design faults and physical faults that may exist. In the past circuit designers and testors have worked independently. With the enormous complexity of VLSI and the limited number of access pins for testing, it has become necessary to include testability in the initial design. One approach to testability is Level Sensitive Scan Design (LSSD) in which internal flip flops can be tied together to form a serial shift register to scan-in test patterns and scan out results [EICH 77]. Another approach uses multiplexing of internal data paths to improve external access to internal logic [WILL 79]. Some methodologies go so far as designing test pattern generators and checkers on-chip [SIEW 82].

Proof of design correctness is a promising area of research. Proving the correctness of algorithms in hardware is considerably simpler than proving correctness of programs because most hardware operations can be broken into a fixed number of functions (e.g. microprogram segments) each of which usually are carried out in a small number of steps (clock cycles). Software programs, on the contrary are often very long and deal with complex data structures and external timing relationships. Innovative research has been conducted in this area from correctness proofs of microprograms to modeling and proof of fault recovery processes [WENS 78, CHRI 83]. A recent project has developed a methodology for verifying the correctness of layered communications protocols between sets of fault tolerant machines [GUNN 83].

Software Error Protection

Various techniques have been used for protection against software errors in computing systems for many years, especially in time-shared systems. Most of these techniques often can be viewed as building firewalls around executing processes so that they cannot damage other users' programs and data. The use of virtual memory systems, privileged instructions, and communication between user programs and the executive via interrupt mechanisms have provided hardware enforcement of the isolation of processes by constraining users to only those resources granted by the operating system.

Although proving correctness of a large and varying set of user programs is seldom if ever done, there has been designs of proveably correct kernels of the operating system. Although this approach cannot prevent single users from failing due to software errors, it allows other users to be protected and prevents user- induced system crashes [POPE 81].

Approaches to Tolerance of Design Faults

All the techniques above are directed at eliminating design faults. Assuming that the occurrence of these faults can be greatly reduced by extensive testing and partial correctness proofs, there still remains the possibility that some faults remain. There are at least two approaches which have been taken. The first, recovery blocks, uses modular programs, and acceptance checks are run as the program modules execute to determine if they are working properly. If an acceptance check fails, a redundant (but different) program module is executed in place of the one for which an acceptance check failed. This approach is similar to standby redundancy in hardware. [ANDE 81].

A second approach relys upon design diversity and is intended to deal with design faults both in hardware and software. A computing algorithm is specified, independent sets of programmers write different programs, and they are run on different computers. At specified points, voting takes place between the independent programs to determine if a fault has occurred and also to mask out the fault and reinitialize a faulty processor and its program. As long as any two processors with their programs are free of design errors in any voted program block, computations will proceed correctly, and if the faulty processor/program can be reinitialized upon disagreement the triplicated computations will continue. This allows faults to exist in all three copies, so long as no two are faulty in one voted computational block. This type of approach is quite expensive, but is justified in life-critical applications where fault coverage (for both design and random faults) must be extremely high.

1.6 CONCLUSIONS

As several fault-tolerant machines have been developed and evaluated, the art of fault-tolerant design has been systematized, and design methodologies have been widely disseminated (e.g. software voters, redundant hardware clocks, and self-checking logic). Reliability prediction models have been developed which attempt to predict the lifetime and performance levels that can be expected of a fault-tolerant system when it is used many years into the future. Techniques have been developed for proving correctness of portions of the software within these machines.

Future advances of the state of the art are likely to require a multidisciplinary approach. We have many special interest areas relating to fault-tolerant system development (1) VLSI fault physics, (2) logic testing and design for test, (3) system architecture, (4) software correctness proofs, (5) robust operating systems, (6) reliability and performance modeling, and (7) fault-tolerant software through recovery blocks or design diversity. Future research into fault-tolerance will require the integration of all these disciplines if comprehensive and fully optimized designs are to be achieved, yet many practitioners of these areas speak different languages. This remains a challenge to the fault-tolerance community.

One thing is certain. Fault tolerant computing will provide considerable opportunities both for entrepreneurs and researchers for the forseeable future. The field has reached adolescence. Systems and design techniques have been developed, yet its greatest opportunities remain in the future.

2.0 FAULT-TOLERANT ARCHITECTURES FOR LONG-LIFE SPACE COMPUTERS

2.1 BACKGROUND

The primary constraints on spacecraft on-board computing systems are the requirements for long unattended like 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. This is compounded by the fact that radiation-hardened LSI devices must be used which often have a much lower circuit density then equivalent commercial components. Thus it can be safely said that reliability requirements induce the majority of costs for on-board computing. Power, weight, and volume are severely limited, and these physical constraints become especially severe since redundant spare modules must be included. Thus it is very important to find hardware and power-efficient forms of fault-tolerant computer architectures.

This lecture will discuss the R&AD program in fault-tolerant computing at the Jet Propulsion Laboratory of Caltech. This NASA facility has been responsible for spacecraft which have successfully explored the Moon, Mars, Venus, Mercury, Jupiter, and Uranus [SCAM 75]. The JPL fault-tolerance program has had three major research efforts which resulted in the development of experimental breadboards. 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. Avižienis 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 [AVIZ 71a]. A breadboard STAR computer was constructed and tested in 1970-1972.

The *second* part of this program was started in 1973, to develop fault-tolerant distributed processing systems, for spacecraft control and data handling. A breadboard distributed system, designated the Unified Data System (UDS) was implemented and tested in 1977 [RENN 76, RENN 78b].

The *third* part of this program was started in 1976, to develop a fault-tolerant distributed system based on the use of existing microprocessors, memory chips, and a small standard set of VLSI building block circuits. A breadboard of this system, the Fault- Tolerant Building Block Computer (FTBBC) System was completed in 1983 [RENN 78a].

The progression between the STAR and the subsequent distributed computing systems was motivated by changes in the technology of digital circuits. Changes in device technology have made new architectures more attractive, as will be described in the following sections of this paper. But device development is not static, and thus there is reasonable probability that our current approach will be obsolete in 5-10 years and will be superseded by a subsequent architecture.

2.2 THE JPL STAR (Self-Testing and Repairing) COMPUTER

The STAR computer was designed to provide a machine with a 10-year life to be used on long-duration missions to the outer planets [AVIZ 71a]. To achieve this degree of reliability it was clear that a great deal of spare hardware would be required. The design had several major constraints which determined our choice of a standby redundant architecture. The most important was that the only flight-qualified components were bipolar small and medium scale integrated circuit devices which offered relatively low packaging density and relatively high power consumption. Nonvolatile magnetic memory was also required. This colored the STAR architecture in the following ways.

a. Severe *power constraints* ruled out using massive redundancy techniques such as running two systems and comparing their outputs or running three systems and voting. Thus a single computer was to be operational which contained sufficient checking circuitry to detect internal faults. The machine was responsible for automatically replacing defective circuit modules with spares and continuing correct computation.

b. Limitations on *power, weight, and volume* required the use of techniques to provide fault-detection at the lowest possible cost in additional hardware. For example, low-cost arithmetic residue codes were used to detect processor faults [AVIZ 71b].

c. Reliability models indicated that standby redundancy (i.e., single active units with backup spares) was the optimum approach to achieving *long life*. Leaving the spares unpowered could help reduce power consumption and increase their life expectancy.

d. The use of SSI/MSI devices required a *fine partitioning* of the computer system into replacement modules. The central processing unit (CPU), for example, required over 1,000 chips which represented a significant failure rate. In order to achieve the the requisite system reliability, it was necessary to partition the CPU into four smaller modules and supply a set of spares for each module.

e. Since only one operational computer was provided, it was necessary to build a special "hard core" module to diagnose faults in the computer, automatically introduce spare modules, and to generate the control signals that initiated software recovery [ROHR 73]. This "hard core" unit (designated the Test-And-Repair-Processor (TARP)) was a low-complexity hardware module which used hybrid redundancy for its own fault-tolerance [AVIZ 71a].

The resulting STAR computer architecture is shown in Figure 12. The computer is partitioned into seven module types designated functional units (FU) which are backed up by spares and interconnected by two 4-wire communications buses. The abbreviations designate the following units:

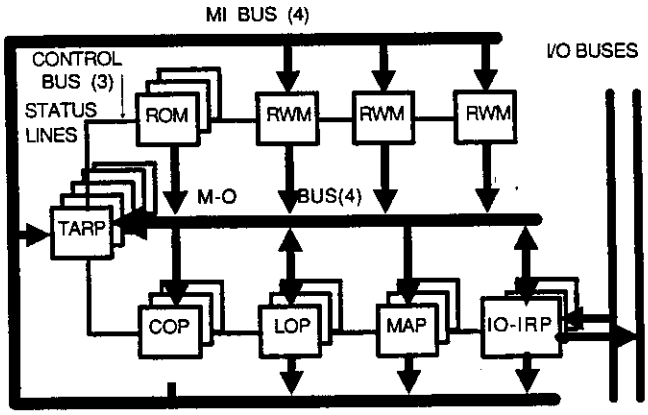


Figure 12: The JPL STAR Computer

COP Control processor; contains the location counter and index registers and performs modification of instruction addresses before execution.

LOP Logic processor; performs logical operations on data words (two copies are powered).

MAP Main arithmetic processor; performs arithmetic operations on data words.

ROM Read-only memory; 16,384 permanently stored words.

RWM Read-Write memory unit with 4096 words of storage (at least two copies powered; 12 units are directly addressable).

IO-IRP Input/output processor; contains I/O buffer registers, and interrupt processor; handles interrupt requests.

TARP Test and repair processor; monitors the operation of the computer and implements recovery (three copies are powered).

Unless otherwise noted, one copy of each FU is powered at a given time. There is a standard interface between each unit and the remainder of the computer. The FU's communicate by means of two 4- wires buses. These buses, designated M-I and M-O, transmit 32-bit words as 8 bytes of 4 bits each. The various processing modules were designed for byte-serial operation to save hardware and thus reduce power consumption and the probability of failure.

The STAR computer is amply documented elsewhere so we will only summarize the salient features of its architecture [AVIZ 71a]. The standard computer is supplemented by one or more spares of each subsystem. The spares are unpowered and are used to replace operating units when faults are discovered. The principal techniques for fault detection and recovery are as follows.

1. All machine words (data and instructions) are encoded in error-detecting codes and fault detection occurs concurrently with the execution of the programs.
2. The computer is divided into a set of replaceable functional units containing their own instruction decoders and sequence generators. This decentralization allows simple fault-location procedures and simplifies interfaces.
3. Fault detection, recovery, and replacement are carried out by special-purpose hardware. In the case of memory damage, software augments the recovery hardware [ROHR 73].
4. Transient faults are identified and their effects are corrected by the repetition of a segment of the current program; permanent faults are eliminated by the replacement of faculty functional units FU's.
5. The replacement is implemented by power switching: units are removed by turning power off and connected by turning power on. The information lines of all units are permanently connected to the buses through isolating circuits; unpowered units have no effect on the bus.
6. The error-detecting codes are supplemented by monitoring circuits which serve to verify the proper synchronization and internal operation of the functional units.
7. The "hard core" TARP is protected by triplication and replacement of failed members of the triplet (hybrid redundancy).

The most unusual module in this computer is the "hard core" TARP. This unit monitors the buses by testing the validity of error-detecting codes. It also monitors status messages from the various FU's. If an improper status message or improperly coded output to a bus is detected from an FU, the TARP diagnoses the unit as faulty. A program rollback is attempted and if the fault persists, the TARP replaces the faulty unit with a spare.

The STAR computer was one of the earliest fault-tolerant machines and thus had a significant impact on subsequent developments in this area. The SAMSO Fault-Tolerant Spacecraft Computer uses many of the architectural techniques pioneered in STAR [BURC 76]. The computer-aided reliability modeling system (CARE) developed as part of the STAR program [AVIZ 71a] produced a model for hybrid redundancy (TMR with spares) and has undergone subsequent development elsewhere. This was followed by a second modeling system, designated RMS, used to model STAR performance [RENN 73a]. A study was undertaken to use the fault-tolerant computer as an automated repairman for the rest of the spacecraft [GILL 72].

Faults were injected into the STAR computer breadboard to determine the effectiveness of the fault-detection and recovery mechanisms. This was done by running a numerical program with known results and clamping randomly selected logic variables to zeros. Successful recovery required a proper fault diagnosis by the TARP and correct (undisturbed) computation when the fault was released. Initial testing uncovered a number of minor design errors which were easily corrected. Subsequent testing demonstrated that approximately 99 percent of all injected faults resulted in proper recovery by the breadboard [RENN 73b].

2.3 DISTRIBUTED SYSTEM APPROACHES

During the early 1970's it became clear that low-power (CMOS) LSI devices and single-chip microprocessors would become available for flight use in the early 1980's, making possible the use of distributed computer systems for future spacecraft. These architectures are well suited to spacecraft computing tasks which support a number of relatively autonomous subsystems.

Spacecraft Distributed System Properties

There are a number of subsystems in current spacecraft which control individual scientific experiments. Examples are subsystems which control radio receivers, power, data storage mechanisms, television cameras, spectrometers, magnetometers, and other instruments. These subsystems vary in complexity but share the property that they contain command interfaces and internal logic sequencers which generate control signals and operate mechanisms to effect the collection of data. An examination of the control logic in these subsystems shows that, for many, it is cost effective to replace the sequencing and interface logic with a microcomputer - either to save chips or to establish standardization in instrument logic designs.

Subsystem-Embedded Computers (Terminal Modules)

The computers which embedded in specific spacecraft instruments and subsystems are designated Terminal Modules (TM) and have the following properties:

1. TMs reside within their associated subsystem. Since they are connected to the subsystem by a large number of subsystem-unique I/O wires, backup spare TMs must also be dedicated and located within the host subsystem. The majority of microcomputers in many spacecraft will be subsystem embedded.
2. Redundant intercommunications buses are required between each TM and the rest of the spacecraft distributed system so as to avoid the possibility of a single-point failure.
3. The intercommunications rate between most embedded computers (TMs) and the rest of the distributed system is relatively low. (It is assumed that the computer network buses will only handle engineering telemetry data and low rate scientific information. A few instruments may generate very high raw data rates, but this information is handled by special telemetry paths, and only lower- rate processed results will be used by the computer system.) Bus communication requirements for most subsystems, are less than a few thousand bits per second, and in most cases, delays of several milliseconds can be tolerated when messages are sent between computers. Most important is the nature of the data transmissions. Since the majority of subsystems operate in a periodic fashion, data movement into and out of various TMs tends also to be periodic. After commands are sent to establish a continuous mode of operation, a fixed pattern of periodic data movements can be established between the various computer modules.

Non-Dedicated High-Level Computers (High Level Modules)

Processing in the various TMs can be controlled and coordinated by nondedicated high-level computer modules (HLMs). A typical spacecraft computer configuration is shown in Figure 13. A set of embedded computers designated terminal modules (TM) are controlled by an HLM designated the Command Processor (CP). This computer acts as a system executive. It receives and stores commands from the Earth and monitors the status of the other computer modules. On the basis of the ground commands and monitored status, it issues local commands to coordinate and control the various other computers in the network. A second HLM is programmed to carry out the function of the Format Processor (FP). Under command of the CP it establishes a periodic pattern of data movements between the memories of the various

computers. It also collects telemetry measurements from the various TM's and HLM's and provides processing services within the network. The high-level modules are nondedicated, and each can be programmed to perform any one of the required computing functions. A common pool of spare (HLM) computers can be employed to protect the high-level computer modules.

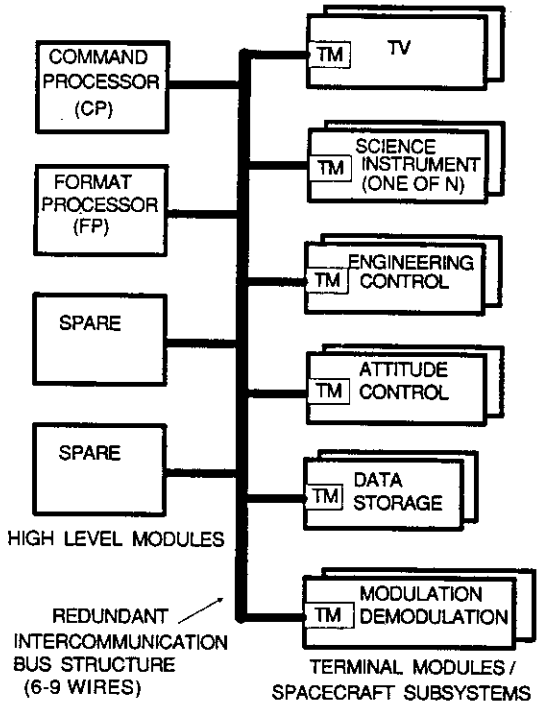


Figure 13: A Typical Spacecraft Configuration

2.4 HARDWARE ARCHITECTURE OF THE UNIFIED DATA SYSTEM (UDS)

The second breadboard system built at JPL was a distributed system designated the Unified Data System (UDS). The UDS architecture consists of a set of microcomputer modules (HLMs and TMs) connected by a redundant set of intercommunications buses as shown in Figure 14.

TM's are located within the various spacecraft subsystems and are responsible for control and data gathering within their associated subsystem. The TM contains a microprocessor (MP), memory (RAM), I/O modules, and several bus adapters BA. The TM interfaces with the other modules in two ways: 1) It receives a single Real-Time Interrupt (RTI) which is common to all modules and which is used for timing and synchronization, and 2) Each TM contains a BA interface to each of several intercommunications buses. Data words can be entered or extracted from the memory of the TM computer using DMA techniques. Each BA can be commanded over its bus to fetch or deposit data into the TM memory. A TM cannot initiate bus communications, but it actively supports DMA transactions into and out of its memory. An external HLM enters commands, data, and timing information into the memory of the TM. The TM delivers information to the system by placing outgoing messages into predetermined locations of its memory, which can then be extracted by the HLM over the bus. The TM can be accessed through several buses simultaneously. The associated BA's provide hardware conflict resolution between competing DMA requests from different buses.

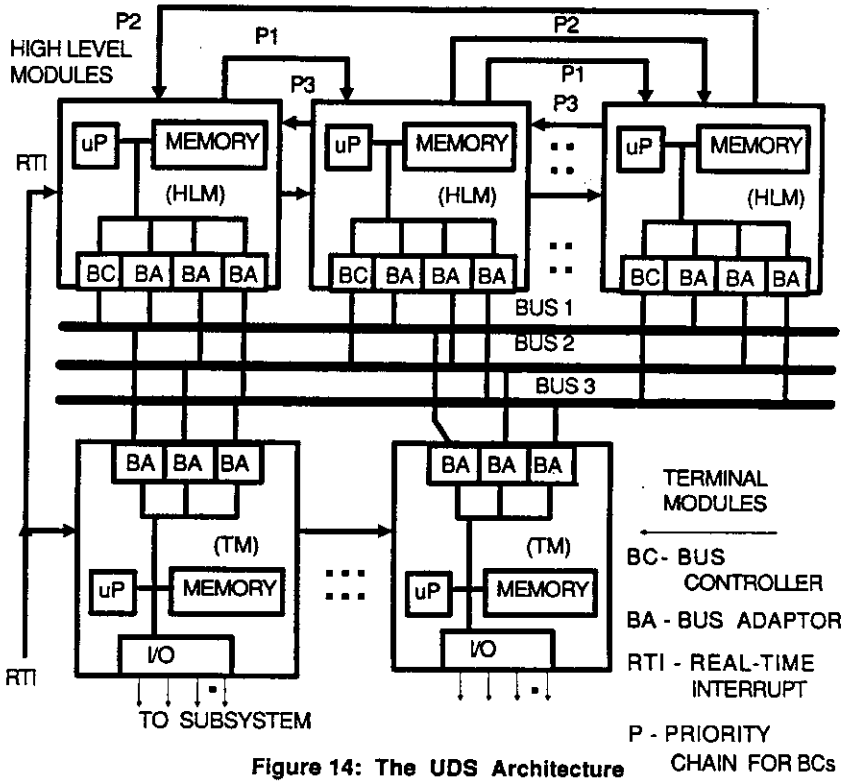


Figure 14: The UDS Architecture

HLM's are responsible for coordinating the processing which is carried out in the remote TM's, for control of intercommunications over the bus system, and for high-level processing such as data compression and decision making. An HLM only communicates with other computer modules and does not contain I/O circuitry other than its connections to the intercommunications bus system. Each HLM consists of a microprocessor (MP), memory (RAM), bus adapters (BA's), and a bus controller (BC). Each BC, which is unique to an HLM, can move blocks of data between the memories of all computer modules connected to its bus (via commands to their BA's). The computer in the HLM activates its BC by presenting it with the address of a bus control table in the HLM memory. This table specifies the source module, destination modules, data names, and the length of the requested information transfer. The BC initiates and controls the specified transmission, monitors status messages to verify a correct transfer of information, and notifies the HLM computer when it is completed. The BC is the mechanism by which the HLM can coordinate the processing in other computer modules by entering commands into their memories and reading out information to monitor ongoing processes.

The *intercommunication bus system* (IBS) consists of several independent serial buses, each of which provides a bandwidth of approximately 1 Mbit. Each bus is connected to one BA in each of the computer modules (HLM's or TM's) to which it is connected. In addition, it is connected to one BC in each of several HLM's connected to the bus. There is a primary bus controller (BC) assigned to each bus whose HLM has complete control over that bus. It relinquishes control over its bus to another HLM under two conditions: 1) its power is turned off, or 2) its processor commands release of the bus to lower priority BC's for a designated time interval. Thus the set of buses may be operated simultaneously with each bus controlled by a different HLM or with individual buses time-shared between several such modules.

Access to each bus by the various HLM's is based on a fixed hardware priority assignment between BC's. A daisy-chain structure is utilized for each bus to establish this priority assignment as shown in Figure 14. Modules of higher priority signal release of a bus via its "daisy-chain," which then activates the hardware necessary to allow bus access within modules of lower priority. Thus spare modules can gain access to a bus whose controlling HLM has failed, or if a bus fails another bus can be shared between two controllers. The individual buses are physically independent; each has its own set of hardware bus access control circuits and a daisy-chain for priority assignments. Therefore, no central bus system controller exists as a potential catastrophic failure mechanism. Similarly, there is no common clock. Each bus uses clock signals generated by the HLM which is in control.

The number of buses within the IBS may be selected to meet mission requirements of data throughput and redundancy. The concept facilitates reconfiguring throughout a mission if failures occur. In the extreme, a single remaining bus can support essential functions of a mission.

The UDS design is oriented toward removing "hard core" items whose failure can cause catastrophic system failure. Intercommunications and clocks have often presented significant problems in this area. These are dealt with the UDS in the following ways. The buses are made independent to avoid any common failure mechanisms. Each computer module uses its own internal clock, and the buses use the clock of whatever module is transmitting. With independent clocks in each computer module there is also a distributed mechanism for protecting against failure of the 2.5-ms common real-time interrupt RTI. If two or more independent RTI signals are generated externally, each computer can decide whether each RTI is correct by comparing these signals with timing derived from its own internal clocks. If an RTI generator fails, the computers will automatically switch to a backup, and if an individual computer clock fails, damage is contained to the faulty module.

System Synchronization and Control Structure

The UDS computers are synchronized by the common 2.5-ms RTI. Various counts of RTI intervals define the uniform time measurement throughout the spacecraft. Analogous to minutes and seconds, the UDS keeps time in frames and lines. A frame (48 sec.) comprises 800 lines (60 ms), and a line comprises 24 RTI intervals. These unusual values of time are chosen for convenience because they correspond to the cycles of instruments on a typical spacecraft. A television picture is read-out every 48 sec. and it consists of 800 lines read-out every 60 ms. Other instruments are synchronized to TV lines, and telemetry sequences tend to repeat on these intervals.

The majority of programs in the various computer modules are self-synchronizing. Each explicitly knows the time counts at which various actions are to be carried out to be synchronized with the rest of the spacecraft. For example, a TV camera control program is designed to start at the beginning of a frame when the line count equals zero. An instrument which carries out 8 cycles during a frame would contain programs which start when the line count is a multiple of 100. Typically, commands which change the operational mode of a subsystem can be received at any time during its current cycle, but they will only be acted upon at the time when the next cycle is to begin. Thus for most commands, there is a wide time window (0.1 sec. to tens of seconds) during which they can be broadcast through the bus system and have identical effects on their destination subsystem.

One HLM in the network serves as a system executive and broadcasts both time counts and commands into designated areas within the memories of the other HLM's and TM's. It reads out data via the bus from these memories needed for control decisions. Under control of this system executive may be several additional HLM's which serve to control collections of TMs or provide data control of data transfer and specialized computing services for the network. In simpler systems, the system executive module directly controls the TM's.

After receiving commands from an HLM, a subordinate HLM or TM starts a set of specified program. These programs utilize the timing information received from the HLM to synchronize with the rest of the spacecraft. For each program it has been precisely specified at which time-counts data is to appear in their memories, at which time-counts control signals are to be generated, and at which time- counts processed data is to be stored in memory for extraction by the HLM.

2.5 THE LOCAL SOFTWARE EXECUTIVE AND SPECIFICATION LANGUAGE

A small local executive program resides in each of the UDS computer modules [RENN 77]. This program, which is identical in all the HLM's and TM's, communicates with user software through a set of special language constructs. Its purpose is to synchronize a set of concurrent programs running within the module. In order to support several concurrent programs which generate precisely timed results and also to support more complex programs which are not easily segmented, software is run in a foreground-background partition. Each processor in the spacecraft system has a well-defined set of foreground program segments to run in each RTI interval. The foreground programs are run in short segments which control and time I/O. They also start and stop unsegmented background programs which perform more elaborate computations which have less stringent timing requirements.

Foreground programs are started by external commands or by other active foreground programs. Commands are entered into a designated area of memory via the bus and have a standard format. The executive periodically checks for the arrival of a command and, if one is received, starts the associated foreground program.

A conceptual diagram of the local executive, which resides in every computer module, is shown in Figure 15. It is built around a scheduling table and is entered at each RTI. Upon entry it suspends the current background process, updates its time counter, and checks for proper exit during the last cycle. It then checks to see if a command has been placed in its memory and, if this is the case, it starts an associated program segment. The executive then checks its scheduling tables to see if any (segmented) foreground programs have requested reactivation at this time. (Activation can occur on the basis of either time or a memory word reaching a specified value.) If so, they are activated sequentially, and each returns to the executive after a few instructions. Upon completing the foreground, the executive returns control to the background program.

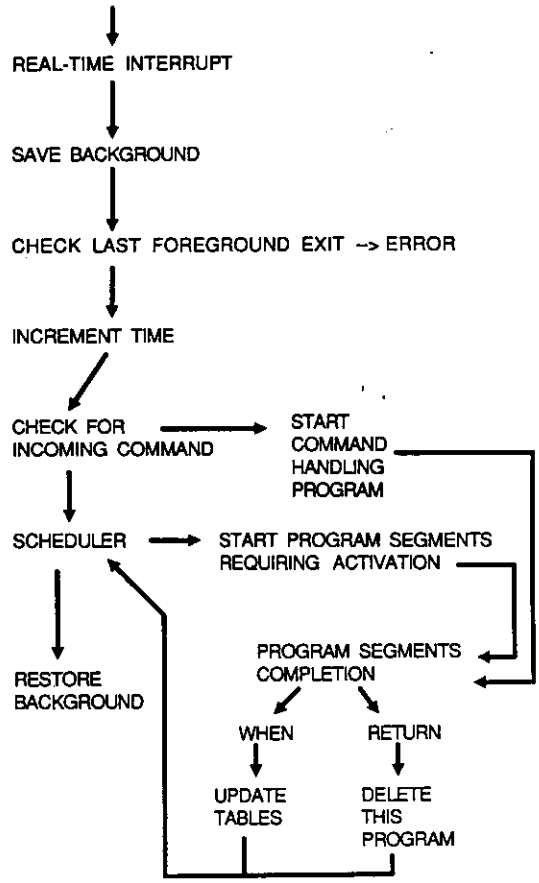


Figure 15: The Software Executive

A UDS program design language has been developed which uses the standard constructs DO, ENDO, CALL, RETURN, IF, ENDIF, and ELSEIF. It is augmented with four special constructs which provide communication with the local executive. These constructs are WHEN, START, STOP and BACKSTART which can be executed from any foreground program [LESH 76]. These special constructs are calls to the local executive to modify its internal scheduling tables to start a program, suspend its execution until a reactivation condition occurs, or to delete a program. START is utilized to activate a new foreground program by placing its entry point in the scheduler. WHEN returns control to the scheduler from the active program and specifies the conditions under which it to be reactivated. By using STOP, a program removes itself from the scheduler, and BACKSTART is used to initiate background programs. An example of part of a UDS program is shown in Table I.

TABLE 1
A Typical Foreground Program

Measure	START TPROG;	(Program to gather elementary measurements concurrently executed)
	OUTPUT L1, L2, L3. . . ;	Initialize the subsystem
	WHEN LINE COUNT = 0;	Suspend execution until start of the next frame.
	DO FOR THE NEXT 50 EVEN LINES (I = 1, 50);	
	OUTPUT SAMPLE COMMAND;	Get next sample of data,
	WHEN RTI = RTI + 1;	Wait for collection at next RTI,
	INPUT DATA AND STORE IN BUFFER (I);	Put it away,
	WHEN LINE COUNT = COUNT + 2;	Wait for next even line,
	ENDO;	
	BACKSTART PROCESS;	Start a background program to process the collected data.
	STOP	

The intercommunications and software approaches employed in the UDS have been developed for near-term applications. There was no attempt to deal with the long term research areas of software fault tolerance. We attempted instead to make software more reliable by simplifying its generation and testing. A number of restrictions have been built into the UDS network to make its operation as predictable as possible, to increase visibility for testing. Timing incompatibilities often represent serious problems when a number of subsystems are connected together. We have attempted to simplify and minimize complex timing requirements at the intercommunications bus interface, where subsystems supplied by diverse groups of people are assembled and expected to work together harmoniously. Finally, if a software fault survives the testing process in TM, we attempt to keep its effects localized by restricting bus access. In most cases a TM software failure will only damage downlink telemetry from the affected subsystem. Software reasonableness checks can be employed in the High-Level Modules to verify that low-level dedicated functions are being properly carried out. We offer little protection against software faults in the HLM's beyond extensive validation and testing.

There is one mechanism for software fault detection in the UDS local executive which is moderately effective. At each RTI, control is forced to the beginning of the local executive. Using two flags the executive records entry and a proper exit from the foreground partition. If control does not return properly to the executive when foreground program segments are executed or if a foreground segment takes too much time, it is probable that the executive will not exit the foreground before the next RTI occurs. At the next RTI, the executive checks the flags to verify proper exit during the last RTI period. If a proper exit did not occur, the executive signals a software fault and executes a user-supplied recovery routine.

2.6 APPROACHES FOR SYSTEM RELIABILITY

The following approaches were taken in the UDS in an attempt to to simplify interfaces between computers and make their operation more predictable in order to simplify testing and fault analysis.

Computer Utilization: It was our intent to supply more computing capacity at each node (memory and processing performance) than was required, and then to use some of the leftover processing speed to simplify system interfaces.

Intercommunications: The hardware investment was made to provide a powerful mechanism for intercommunication between computers. This was done to releave software of the costly overhead functions of moving intercommunications data, freeing it to perform more of the tasks for which the subsystem is intended. Data movements between computer memories are hardware-controlled in a fashion similar to channels in larger computers. Automatic status messages are included with each message to verify proper transmission and to allow rapid error recovery by retransmission.

Error Confinement: The majority of computers in the spacecraft system perform dedicated low-level functions. These embedded TM computers of the UDS are prevented from having the ability to arbitrarily modify the memories of the other computers or to tie up an intercommunications bus. If this were the case, software errors in low-level computers could freely propagate throughout the system. There are many ways to achieve this fault-confinement, and we chose to restrict bus access so that low-level computers cannot initiate communications activity. Centralized bus control is used, and (as will be described in subsequent paragraphs) it is well adapted to the synchronous nature of the spacecraft computing processes.

Minimization of Interrupts: Demand interrupts are minimized in the UDS. Whenever possible, the software is allowed to determine when and in what order it interfaces with the outside world. This tends to slightly increase the number of instructions required and limits

I/O response to millisecond rather than microsecond resolution. But it also leads to more predictable operation, is more easily verified, and allows for software self-defense. If no restrictions are placed on the response of the spacecraft computers to external stimuli, program verification can become quite complex. There can be a very large number of possible timings and orderings of incoming service requests. Fortunately, most electromechanical devices on spacecraft do not require a response to unexpected conditions in less than a few milliseconds. Thus we can restrict the set of possible input states so that software can be more easily verified and have higher reliability.

Control Hierarchy: The spacecraft control structure is hierarchical, and this is reflected in the UDS design. Low-level computers (i.e. TMs) in subsystems are controlled by high-level computers (HLMs), which may in turn be controlled by other HLMs. In order to simplify testing and software verification, we imposed the restriction that each computer in the system can receive commands from only one other computer. Control between computers was effectively limited to a tree structure to provide simplification and a degree of fault-containment.

There are two types of information transfer between computer modules. The first type consists of commands which are sent from the controlling HLM and which specify the algorithms to be performed in the controlled machine (HLM or TM). The second type of information transfer is the movement of data between computers. The controlling HLM also controls the movement of data into and out of the controlled machine, but the data can come from any of the machines under its control. (The controlling computer can also delegate the control of data movements to other high-level computers under its authority in certain circumstances.)

Programs in the TMs interface with the other computers through buffers allocated in their own memories. These programs are invoked by commands from the controlling HLMs which also has responsibility for placing the operands which are needed in their local memories. The programs in the TMs are self-synchronized to process the data when they arrive and to place results in their designated memory buffers for subsequent extraction by the high-level computer.

Synchronous Communications: Control of experiments and engineering subsystems on the spacecraft is tightly synchronized. Subroutines in the associated computers meet strict timing requirements in generating control cycles for their various instruments. This allows correlation of the results of various scientific experiments and provides data at the right time to fit into telemetry formats. Thus for any spacecraft operational mode, the state and periodicity of most instrument and subsystem cycles is well defined, as are the requirements for data transfer between computers. This prevents conflicts and simplifies intercommunications since no conflict arbitration is required. The high-level computer containing a central bus controller establishes a periodic set of data transmissions between computers as is needed for the particular cycles they are performing. (The few nonperiodic functions can be treated in a similar fashion by establishing periodic transmission of message buffers between their associated subroutines.) Bus control is easily verified since it is generated from a single controller (from internal memory tables) and is highly predictable. The cost of forcing intercommunications into periodic data movements is a reduction of bus response time to asynchronous transmissions. A computer must wait for its time-slot before communicating with another machine. This restriction is acceptable in the spacecraft because of the periodic nature of its computations, and thus we have sacrificed performance (concurrency) for increased testability.

Timing Hierarchy: It is often the property of real-time systems that low-level computers provide relatively simple functions such as generation of control signals and the collection of data. These simple tasks must be performed rapidly and with a high degree of timing precision. Intermediate-level tasks, such as spacecraft command processing, are often quite complex and take considerable time to compute. Fortunately, these tasks often have an even wider latitude in timing resolution. This timing hierarchy is analogous to local reflexes and carefully thought out motions in the human control system.

We have attempted to take advantage of a similar timing hierarchy in the spacecraft system and ease timing requirements at progressively higher levels within the system. Simple high-rate signal generation with microsecond precision is removed from software and carried out by hardware I/O devices. Low-level computer programs are not required to provide timing resolution below a few milliseconds. Additionally, the software in the low-level subsystem computers is designed to minimize the timing resolution required of the high- level computers which sends them messages over the intercommunications buses. The goal of this approach is to simplify expensive software and system interfaces.

I/O Timing Granularity to Simplify Software Modification: In the computers on board the spacecraft, several programs must operate concurrently and generate precisely timed control signals for their associated subsystems. For example, the dedicated television computer may control the readout of picture lines, sample telemetry measurements, format picture data for readout, and execute other concurrent functions which must be precisely timed.

It is desirable to be able to change any one of these programs without affecting the input and output timing of the others. This can be achieved to a large extent by imposing granularity on I/O. Inputs are sampled and held for uniform (several millisecond) RTI intervals. During these intervals, segments of several concurrent foreground programs may be executed. Their outputs are collected by I/O hardware and held until the end of the

time interval, and then all outputs are executed at once. The program segments can be executed in any order, and some can be removed without affecting the output timing of the others. Programs can be added as long as the total computation for any interval does not exceed the time available. This approach simplifies simulation since the possible order and timing of inputs are drastically reduced in complexity. Visibility into the system is also improved for testing, since programs are executed in well defined steps during which inputs are held constant. Software can be more easily modified. The cost of this approach is reduced response time to external events. It may require two to three time intervals, on the order of 5-7 ms, for the computer to acquire unexpected data and deliver a response. This is acceptable for the spacecraft application.

A six-computer UDS breadboard was constructed using 3 HLM's and 3 TM's, and it was programmed to carry out a number of spacecraft processing functions. (It includes "flight" television camera and tape recorder subsystems controlled by two of the TM's and two HLMs acting as the Command and Format Processors.) Software tools have been written to take advantage of the predictable, time-synchronized interactions between modules. The breadboard can be started and run to a given spacecraft time-count and then stopped for inspection. The memories of the various modules can be inspected to determine if the correct bus transmissions have taken place and if the foreground programs are in the correct state. This predictability has greatly expedited software debugging and in turn accelerated program development. The minimization of demand interrupts, restrictions on control, synchronous communications, and techniques for fault containment have made the system more predictable and easily debugged.

We are satisfied with the experiences that have occurred with the UDS system. Several major changes have occurred in the telemetry handling of the spacecraft simulation which caused reprogramming of several UDS computers. These changes were made rather easily in a matter of a few days.

The UDS breadboard assumed that concurrent fault detection would be implemented in the various computer modules. Transient errors would be detected and corrected by program rollback within each computer module (TM or HLM) and that if a permanent fault occurred, the computers would halt and signal an error through the bus. The demonstration breadboard did not include these fault detection capabilities because it was made from existing microcomputers which could not easily be modified. The fault-tolerant bus system was a custom design which was implemented along with the executive software and I/O interfaces. The implementation of concurrent fault detection was carried out in the third JPL breadboard project, the Fault-Tolerant Building Block (FTBBC) computer described below.

2.7 THE FAULT-TOLERANT BUILDING BLOCK COMPUTER (FTBBC)

The JPL Fault-Tolerant Building-Block Computer (FTBBC) architecture is designed to use a small set of VLSI building-block circuits to interconnect existing microprocessor and memory chips to form Self-Checking Computer Modules (SCCM). These modules can be implemented as HLMs and TMs in a distributed system and provided the concurrent fault-detection needed to implement fault-tolerance.

Each SCCM is a small computer which is capable of detecting its own malfunctions. It contain I/O and bus interface logic which allows it to be connected to other SCCM's to form fault-tolerant, UDS-type, distributed systems. The SCCM contains commercially available microprocessors, memories, and four types of building-block circuits as shown in Figure 16. The building blocks are: 1) an error detecting (and correcting) Memory Interface Building Block (MI-BB), 2) a programmable Bus Interface Building Block (BI-BB), 3) a Core Building Block (Core-BB), and 4) an I/O Building Block (IO- BB). A typical small SCCM consists of 2 microprocessors, 23 RAM chips, 1 MI-BB, 3 BI-BB's, 2 IO-BB's, and a single Core-BB.

The building-block circuits control and interface the various processor, intercommunication, memory, and I/O functions to the SCCM's internal bus. Each building block is responsible for detecting faults in its associated circuitry and then signaling the fault condition to the Core-BB by means of an internal fault indicator. The MI-BB implements fault detection and correction in the memory and also provides detection of faults in its own internal circuitry. Similarly, the BI-BB and IO-BB provide intercommunications and I/O functions, along with detecting faults within themselves and their associated communications circuitry. The Core-BB checks the processing function by running two CPU's in synchronism and comparing their outputs. It is also responsible for fault collection and fault handling within the SCCM.

The Core-BB receives fault indicators from the other building-block circuits and also checks internal bus information for proper coding. Upon detecting an error, the Core-BB disables the external bus interface and I/O functions, isolating the SCCM from its surrounding environment. The Core-BB can optionally: 1) halt further processing until external intervention, or 2) attempt a rollback or restart of the processor. Repeated errors result in the disabling of the faulty SCCM by its Core-BB. Recovery can be effected by an external SCCM which is programmed to recognize the lack of activity from the faulty SCCM.

time interval, and then all outputs are executed at once. The program segments can be executed in any order, and some can be removed without affecting the output timing of the others. Programs can be added as long as the total computation for any interval does not exceed the time available. This approach simplifies simulation since the possible order and timing of inputs are drastically reduced in complexity. Visibility into the system is also improved for testing, since programs are executed in well defined steps during which inputs are held constant. Software can be more easily modified. The cost of this approach is reduced response time to external events. It may require two to three time intervals, on the order of 5-7 ms, for the computer to acquire unexpected data and deliver a response. This is acceptable for the spacecraft application.

A six-computer UDS breadboard was constructed using 3 HLM's and 3 TM's, and it was programmed to carry out a number of spacecraft processing functions. (It includes "flight" television camera and tape recorder subsystems controlled by two of the TM's and two HLMs acting as the Command and Format Processors.) Software tools have been written to take advantage of the predictable, time-synchronized interactions between modules. The breadboard can be started and run to a given spacecraft time-count and then stopped for inspection. The memories of the various modules can be inspected to determine if the correct bus transmissions have taken place and if the foreground programs are in the correct state. This predictability has greatly expedited software debugging and in turn accelerated program development. The minimization of demand interrupts, restrictions on control, synchronous communications, and techniques for fault containment have made the system more predictable and easily debugged.

We are satisfied with the experiences that have occurred with the UDS system. Several major changes have occurred in the telemetry handling of the spacecraft simulation which caused reprogramming of several UDS computers. These changes were made rather easily in a matter of a few days.

The UDS breadboard assumed that concurrent fault detection would be implemented in the various computer modules. Transient errors would be detected and corrected by program rollback within each computer module (TM or HLM) and that if a permanent fault occurred, the computers would halt and signal an error through the bus. The demonstration breadboard did not include these fault detection capabilities because it was made from existing microcomputers which could not easily be modified. The fault-tolerant bus system was a custom design which was implemented along with the executive software and I/O interfaces. The implementation of concurrent fault detection was carried out in the third JPL breadboard project, the Fault-Tolerant Building Block (FTBBC) computer described below.

2.7 THE FAULT-TOLERANT BUILDING BLOCK COMPUTER (FTBBC)

The JPL Fault-Tolerant Building-Block Computer (FTBBC) architecture is designed to use a small set of VLSI building-block circuits to interconnect existing microprocessor and memory chips to form Self-Checking Computer Modules (SCCM). These modules can be implemented as HLMs and TMs in a distributed system and provided the concurrent fault-detection needed to implement fault-tolerance.

Each SCCM is a small computer which is capable of detecting its own malfunctions. It contain I/O and bus interface logic which allows it to be connected to other SCCM's to form fault-tolerant, UDS-type, distributed systems. The SCCM contains commercially available microprocessors, memories, and four types of building-block circuits as shown in Figure 16. The building blocks are: 1) an error detecting (and correcting) Memory Interface Building Block (MI-BB), 2) a programmable Bus Interface Building Block (BI-BB), 3) a Core Building Block (Core-BB), and 4) an I/O Building Block (IO- BB). A typical small SCCM consists of 2 microprocessors, 23 RAM chips, 1 MI-BB, 3 BI-BB's, 2 IO-BB's, and a single Core-BB.

The building-block circuits control and interface the various processor, intercommunication, memory, and I/O functions to the SCCM's internal bus. Each building block is responsible for detecting faults in its associated circuitry and then signaling the fault condition to the Core-BB by means of an internal fault indicator. The MI-BB implements fault detection and correction in the memory and also provides detection of faults in its own internal circuitry. Similarly, the BI-BB and IO-BB provide intercommunications and I/O functions, along with detecting faults within themselves and their associated communications circuitry. The Core-BB checks the processing function by running two CPU's in synchronism and comparing their outputs. It is also responsible for fault collection and fault handling within the SCCM.

The Core-BB receives fault indicators from the other building-block circuits and also checks internal bus information for proper coding. Upon detecting an error, the Core-BB disables the external bus interface and I/O functions, isolating the SCCM from its surrounding environment. The Core-BB can optionally: 1) halt further processing until external intervention, or 2) attempt a rollback or restart of the processor. Repeated errors result in the disabling of the faulty SCCM by its Core-BB. Recovery can be effected by an external SCCM which is programmed to recognize the lack of activity from the faulty SCCM.

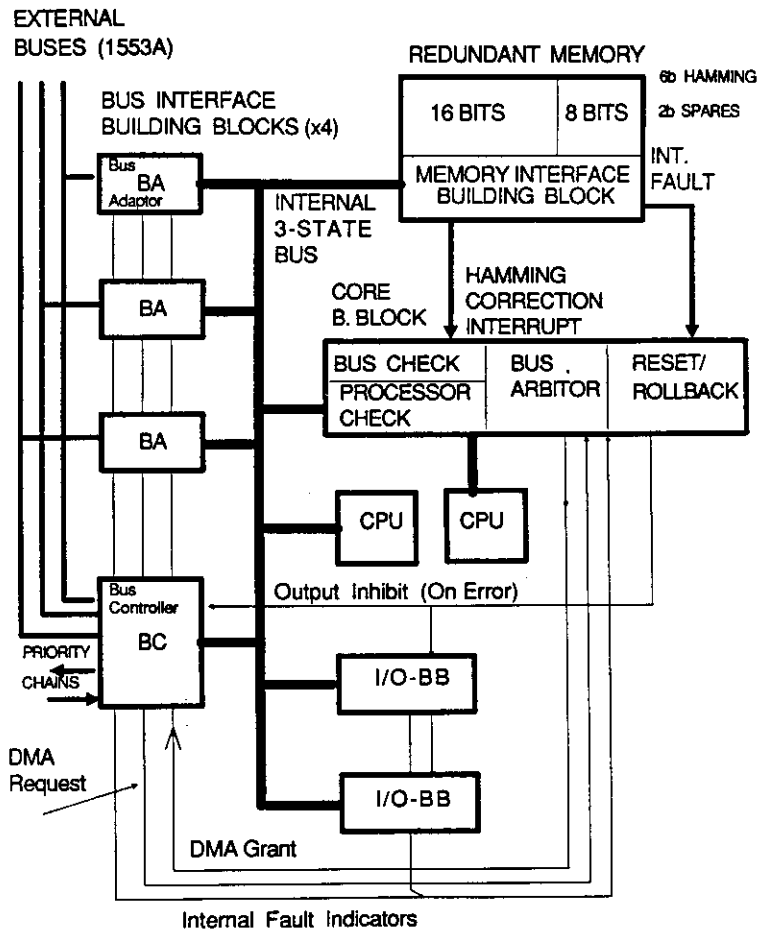


Figure 16: The SCCM

An important attribute of the building blocks in the SCCM is that they are interconnected via the internal processor-memory bus. They are all designed to perform specified functions in response to read or write commands to reserved addresses appearing on the internal bus. The majority of addresses are used for conventional access to RAM, however, the upper 4096 addresses are reserved for I/O functions, external bus transmission requests, reading out error-status information from the building-blocks, and sending reconfiguration commands to the building blocks. For a fetch request to a specific reserved address, the building-block circuit which recognizes the address performs the specified function and delivers a word of information to the internal data bus. Store requests to reserved addresses deliver information over the internal data bus to the selected building block. This is the commonly used technique of "memory- mapped I/O" and it has two major advantages in the building-block SCCM design. *First*, this approach avoids processor-specific I/O operations and thus allows the use of a number of different off- the-shelf microprocessor in the SCCM. *Second*, this approach allows access to the building blocks by both software in the SCCM and from other SCCM's via the external bus system. Using the external bus an external SCCM can command DMA READ and WRITE operations into and out of the memory of the local SCCM. By directing DMA READ and WRITE cycles to reserved addresses, the external SCCM also has access to the building blocks in the local SCCM. The external SCCM can load and read out memory via the bus, and can also sample error status information, command internal reconfiguration, and can even remotely control I/O in a faulty local SCCM.

The following is a brief description of the building-block circuits [RENN 78a].

A. The MI-BB

The MI-BB interfaces a storage array (consisting of a redundant set of memory chips) to the SCCM internal bus. It provides Hamming correction to damaged memory data, replacement of a faulty bit plane with a spare, parity encoding and decoding to the internal bus, and detection of internal faults within its own circuitry.

The MI-BB needs only be capable of detecting errors to satisfy the requirement of an SCCM. However, memory is the source of the largest number of failures within the SCCM, and single-fault repair in this area will greatly improve the reliability of the SCCM. A block diagram of the memory interface building block is shown in Figure 17.

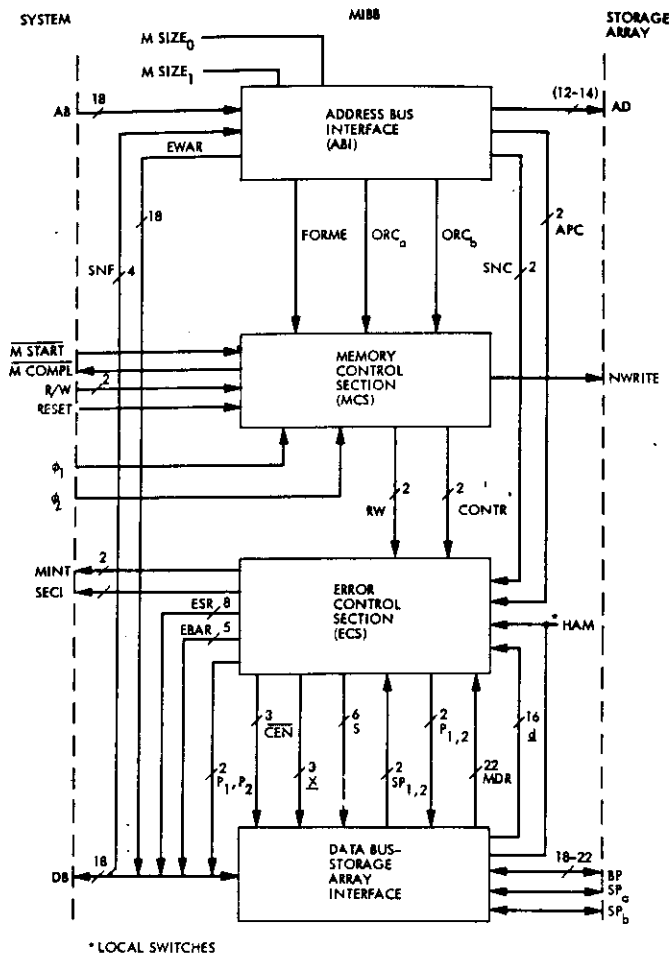


Figure 17: The Memory Interface Building Block (MIBB)

The *access element* (AE) provides the address parity checking and decoding required to select a memory module. It stores and validates the incoming address by means of a self-checking parity checker circuit. If no errors are detected, the low-order 13 bits are sent to the storage array where decoding is performed. (It is assumed that the memory is packaged one-bit per chip so that an on-chip addressing error can be detected using the SEC/DED code.) The decoding of the three high- order address bits is performed in duplex circuits checked by a morphic comparator. More than one memory module (MI-BB plus storage array) may be employed in the SCCM. The three high-order bits are used to select modules within the SCCM storage system. These bits are used as "soft names" and are mapped into a physical module address using a small duplicated associative memory. Soft name assignments are established by commands over the local SCCM bus.

The *error control (EC) element is responsible for* generation of Hamming code check bits and syndromes, byte-parity generation and checking (for the SCCM internal bus), and error analysis. The circuits used in the EC are also self-testing. A single-bit error is corrected by decoding the syndrome generated from the word read from memory, in order to localize the faulty bit. The correction is performed by complementing the faulty bit.

An error analyzer collects various error indications such as single error, double error, and circuit error, and they are recorded in an error status word, which can be transmitted over the bus on system demand.

The *bit plane replacement (BR) element performs the reconfiguration of* the storage array. It contains a multiplexer circuit which can replace any bit plane in the memory with a single standby spare. The bit to be replaced is specified by an external command.

The *data bus interface* (DBI) contains a memory data register and the three-state drivers and receivers used to interface with the SCCM data bus. Bit inversion for Hamming correction is performed in this register.

The *memory control* (MC) element receives commands from the internal control bus which specifies READ and WRITE operations. For addresses less than 61,440 the commands are interpreted as normal memory operations. READ and WRITE instructions with addresses larger than 61,440 are reserved for memory-mapped I/O. A set of these out-of-range addresses is reserved for commands to the MI-BB. Among these commands are: 1) READ ERROR STATUS WORD, 2) READ ERROR POSITION OF FAULTY WORD, 3) READ ADDRESS OF LAST ERROR, 4) RESET, 5) DISABLE CORRECTION, 6) READ REDUNDANT CHECK BITS, 7) REPLACE *i*th BIT PLANE WITH SPARE, and 8) SET SOFT NAME. MC element circuits are duplicated with comparison for fault detection.

>The complexity of the MI-BB is approximately 2,000 gates. This represents a small failure rate with respect to the memory plane and is readily implemented as a single LSI circuit.

B. The Programmable BI-BB

The BI-BB provides the mechanism by which information is transferred between SCCM's via an intercommunications bus system. This external bus system consists of several redundant buses, each of which employs the MIL STD 1553A format. Each BI-BB can be microprogrammed to play the role of either a bus controller (BC) or remote terminal (designated bus adapter (BA)) for a single 1553A bus. Several BI-BB's are employed in each SCCM so that each computer module can communicate over several buses simultaneously.

The BC and BA functions provided by the BI-BB are much more powerful than those normally implemented for 1553A controllers and terminals. The BI-BB's are capable of moving data directly between the memories of the SCCM's attached to a given data bus with a minimum of software support. The controller and adapters on a given bus operate together in a relatively autonomous fashion similar to the data channels on much larger machines, as described below.

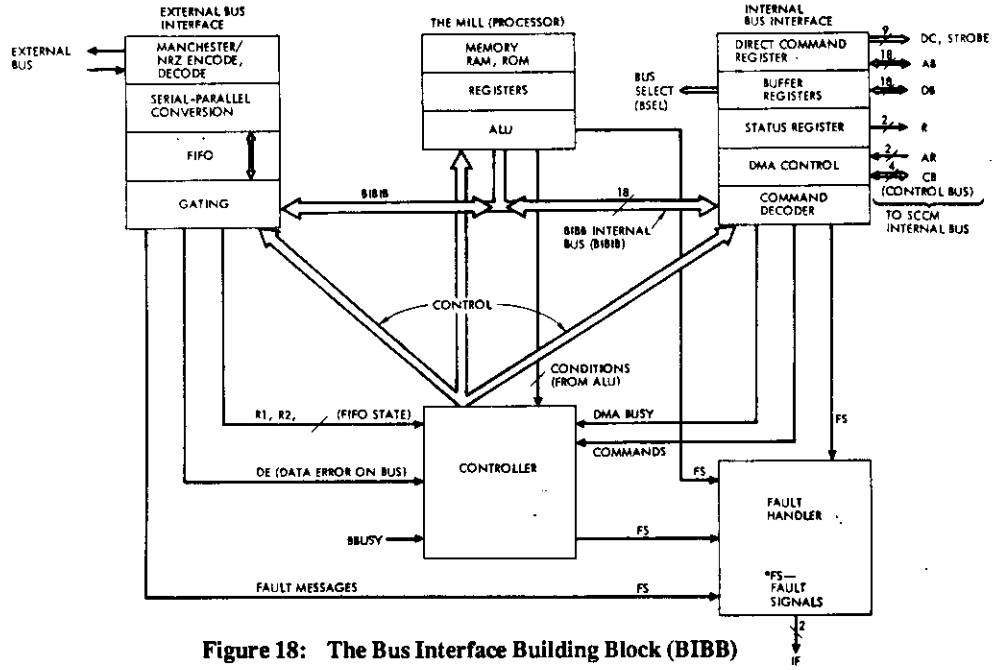


Figure 18: The Bus Interface Building Block (BIBB)

An SCCM which acts as a HLM and controls an intercommunications bus contains a BI-BB which is microprogrammed to be a BC. When it wishes to initiate a data transfer between the memories of the SCCM modules on its bus, it alerts the BC.

The BC reads a control table from its host SCCM's memory which specifies the source and destination of information required for the bus transfer along with the length of the transmission. The BC then broadcasts the appropriate commands over the bus system to "set up" the transmitting and receiving adapter circuits. It monitors the transfer of information, records status messages, and notifies the host computer upon completion of the transfer.

BI-BB's in other SCCM's connected to the same intercommunications bus are microprogrammed as BA's. These BA's serve as remote terminals on the bus. The BC, in setting up a transfer, specifies one adapter as a data source. It then specifies one or more BA's as data acceptors and names the data to be moved.

The Source adapter then finds and extracts the specified information from it host SCCM's memory (using cycle stealing) and places this information on the bus. Simultaneously, the acceptor adapter(s) takes this information off the bus and loads it into its host SCCM's memory via cycle stealing.

An SCCM can obtain several bus adapters to provide an interface to a number of redundant intercommunications buses. Communication can occur simultaneously over as many as three buses with an SCCM without conflict (time delays) seen on any bus. A BA cannot initiate a bus transfer, but only responds to the commands of a BC. Provision is made for sending discrete commands through BA's such as POWER ON, POWER OFF, HALT, INTERRUPT, RECONFIGURE, etc.

A block diagram of the BI-BB is shown in Figure 18. It consists of five major elements, a Manchester/NRZ translator, a microprogram control unit (MCU), a control ROM, a data path element, and a direct memory access (DMA) controller.

The *Manchester/NRZ translator* translates incoming Biphase Manchester commands and data, supplying a bus-synchronized clock, command and data word-sync indicators, NRZ data, and parity and Manchester-error detection signals. It will also accept NRZ data, encode it, and output Manchester data for bus transmission (along with the associated command and data sync signals).

The MCU is a microprogram sequencer. A microprogram location counter is started at one of several fixed addresses by command sync, data sync, or a host processor command (detection of an out- of-range address). The location counter proceeds through sequential addresses or branches on the basis of incoming data, internal flags, or other internal circuit conditions. The microprogram sequencer is programmed to generate a unique set of address sequences for each type of incoming bus command, data sequence, or computer command. This output sequence is then mapped through a control ROM to generate the detailed control signals required to drive the data path, MCU, and DMA control elements.

The *control ROM* maps the microprogram address sequence into control signals for the various circuit elements.

The *data path section* contains 1) registers necessary to buffer addresses and data, 2) ROM to store memory protection bounds, data keys, and table addresses, and 3) an arithmetic logic unit for addressing

computations. This circuit is not unlike existing bit slice processors, with the exception that serial-parallel conversion registers, ROM, and several holding registers are required for the unique bus interface and DMA functions.

The *DMA control circuit* is responsible for obtaining control of the host SCCM's bus and transferring data between the BI-BB and the SCCM's memory.

The fault detection techniques employed in the BI-BB are based on parity coding to protect memory information and duplication with morphic comparison for most of the logic circuitry. This building block has a complexity equivalent to approximately 10,000 gates.

C. The IO-BB

I/O requirements of host systems vary widely in voltage ranges, currents, and timing parameters. The approach best suited to building-block development is to provide a standard set of functions which serve a majority of general applications. The user is required to supply any additional functions unique to his applications.

To be consistent with the SCCM, all building blocks provide memory-mapped I/O. That is, each IO-BB must recognize its identification and the function being requested from an out-of-range address appearing on the host SCCM's internal address bus. Data for output or input is transferred over the data bus in response to a processor write or read to the specified address.

Candidate I/O functions are: 1) 16-bit parallel data in and out, 2)16-bit serial data in and out, 3) a pulse sampling circuit, 4) a pulse counter, 5) a pulse generator, 6) an adjustable frequency generator, 7) an analog multiplexer with A/D converter, and 8) a high-rate DMA channel. The density of VLSI technology is sufficiently high that a number of I/O functions can be supplied on a single chip. The specific function which is required can be activated by connecting pins. This approach can reduce the inventory of different IO-BB's to one or two.

The fault-detection techniques employed in the IO-BB's are straightforward. Where bus information is preserved, parity checking is employed, and other functions are protected by internal duplication with morphic comparison. Input and output short-circuit protection must be provided when two dedicated SCCM's are cross-strapped; i.e., their inputs and outputs are connected together. Otherwise a shorted I/O connection can inactivate a redundant set of SCCM's.

D. The Core-BB

The Core-BB is responsible for 1) detecting CPU faults by synchronizing and comparing two duplex CPU's, 2) collecting fault indications from itself and other building blocks, and 3) disabling its host SCCM upon detection of a permanent fault.

Two options are provided for attempted recovery from transient faults. These are: 1) stop at first fault indication; wait for outside help; and 2) roll back at first fault indication; stop if the fault recurs.

Specific functions of the Core-BB are listed below: 1) compare two CPU's for disagreement; 2) parity encode CPU output for SCCM internal bus transmission; 3) check parity on the internal bus; 4) recognize Core-BB commands which can be sent from an external module via a bus adaptor to a Core out-of-range address (these are commands to halt and inhibit outputs, restart, and enable outputs of the SCCM); 5) allocate the internal tristate bus amongst several DMA requests from the BC, BA, and I/O; 6) detect internal faults within the core module; 7) collect internal-fault indications from all building blocks within the SCCM; 8) disable SCCM output under fault conditions; 9) provide rollback/restart compatibility for optional transient fault recovery; and 10) halt computation on recurring faults.

The Core-BB consists of three elements as shown in Figure 19. The *processor check element* serves three functions: 1) to compare the outputs of two synchronous processors, 2) to encode and check internal bus parity, and 3) to recognize and decode commands sent to the Core-BB through the internal bus. It contains self-checking parity checkers, a duplex command decoder and morphic reduction trees.

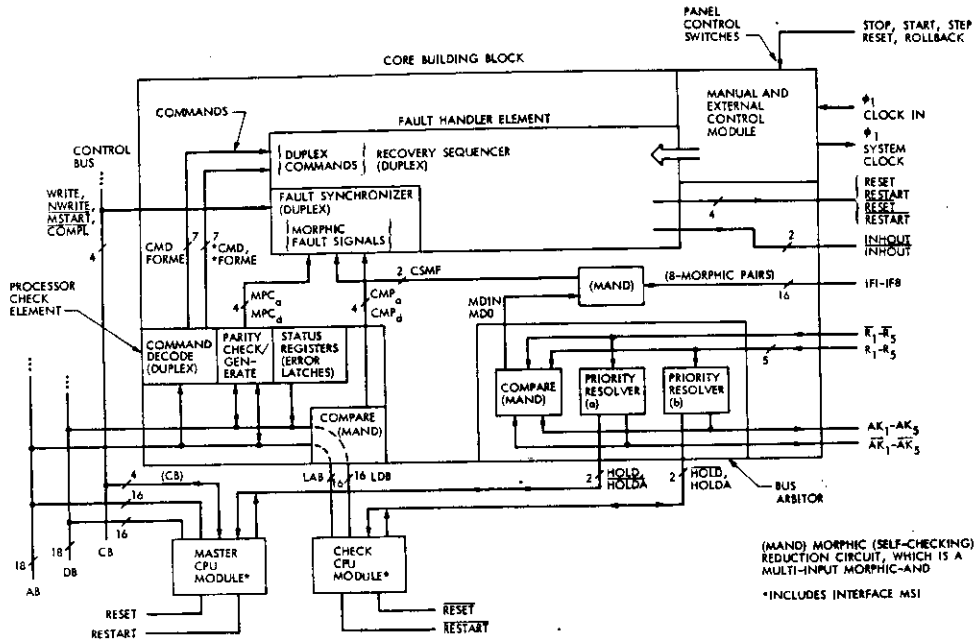


Figure 19: Core Building Block Block Diagram

The *bus arbitration element* accepts morphic bus request signals from the various DMA controllers on other modules. It obtains release of the bus by the processor and grants access to requesting building blocks on the basis of hardware priority. The bus available signal is sent as a morphic indicator to each requesting module. The internal circuits of this element are duplexed and are compared with a self-checking comparator.

The *fault-handler element* accepts morphic fault indicators from the other building blocks and from within the Core-BB. It reduces these to a single 2-wire master fault indicator which indicates a fault somewhere in the computer module. This fault indicator disables the bus controller and output drivers, isolating the SCCM from the rest of the system. Duplex recovery sequencers are employed to implement optional transient recovery sequences. They are checked with a self-checking comparator.

Principal characteristics of the SCCM design are summarized below:

Implementation of Error and Fault-Detection Error and fault detection are carried out in several ways. Regular structures, such as data paths are protected with low-cost error detecting codes. Error correction is also provided in memory using Hamming SEC/DED codes. Irregular logic structures represent a relatively small fraction of the complexity of the SCCM (in terms of active chip area) and are protected by means of duplication with comparison (e.g. microprocessor chips are operated in pairs with output comparison). Similarly, a number of circuits are internally duplicated within the VLSI building-blocks and compared to verify proper operation. Finally, the problem of "Who checks the checker?" has largely been solved with the development of morphic "self-checking" checkers [CART 72]. Morphic circuits are used to reduce both parity checks and comparisons of duplicated circuits into a small number of self- checking fault indicators.

Response to Errors and Faults The SCCM responds to detection of an internal error by disabling its own output capability and attempting a program rollback which will often correct errors which are transient in nature. If the error recurs, it is assumed that a permanent fault has occurred, the SCCM halts, disables its outputs and raises a "flag" (readable through the bus system) that it is faulty. An external SCCM can detect the fault by recognizing an absence of activity from the faulty module or by interrogating the module through any of several paths in the redundant bus system. Reconfiguration commands can be issued through any of several redundant buses.

2.8 PROCEDURES OF FAULT RECOVERY

The SCCMs were designed to be embedded as HLMs and TMs in a UDS-type distributed syste. The same UDS software structure is employed. We first discuss recovery from faults in TMs and then HLM-fault recovery.

Recovery from Faults in TMs

TMs, upon detecting an internal error, attempt a program rollback in an attempt to recover from a transient error condition. If this is successful, the controlling HLM only sees a slight delay in processing and computing continues. If the rollback is unsuccessful and the error persists, the TM shuts down and raises an error flag. Since a TM does not have the ability to initiate

a bus communication, it can only halt and signal an error. Recognition of a failed TM and commands for reconfiguration are the responsibility of a HLM.

Each controlling HLM is responsible for polling the various modules under its control to determine if a module has isolated itself due to a failure. This polling process can be carried out nearly automatically, using the bus system every few (10-100) milliseconds. The HLM is then responsible for issuing the reload and reconfiguration commands necessary to replace a faulty module with a spare or re-initialize a transient-disturbed module.

These commands can be sent to the faulty module through one of its several redundant BA's, which provide the following functions: 1) power or unpower the internal computer, 2) load or readout memory, 3) halt or start the processor at specified locations, 4) sample error status from the building blocks, and 5) send reconfiguration commands to the building blocks. Since there are several BA's in each module which are connected to independent bus systems, there are redundant paths for carrying out reconfiguration. The BA's are powered at all times, and it is usually possible to interrogate a faulty module.

The use of memory-mapped control and I/O in the SCCM is very important in providing the capability for external diagnosis of faulty modules. When an SCCM fails and its internal self-checking logic disables bus control and I/O functions, we still have access to the internal bus of the SCCM through one of its BA's. The BA's normal function allows an external SCCM to command DMA READ and WRITE cycles on the internal bus of the faulty module. If the specified addresses are within the range of RAM, the external SCCM can read or load its memory. If the address is an out-of-range (i.e., reserved) address, the external SCCM can directly command and read status data out of the building blocks of the faulty module.

When an HLM detects a faulty TM, it reads out the fault status information from its building blocks, and can then take one of several strategies. It may attempt to reload and restart the faulty module; or it may immediately substitute a spare if one is available. If no spares are available, the HLM has a final option of directly commanding the faulty subsystem (in a degraded mode) by sending commands directly to its I/O building blocks. The I/O building blocks are reached by sending out-of-range read and write requests through the external bus to the faulty SCCM.

The requirements for fault recovery vary in the different modules in the system. Most HLM's and TM's can tolerate a program interruption of up to a few seconds. When one of these modules develops an internal fault, it is adequate to replace the module with a spare, reload the spare's memory, and restart the appropriate internal programs. If several parameters are needed which indicate the subsystem state before the failure, they can be supplied by the controlling HLM. The HLM periodically samples the required state information and stores it in its memory. This information is supplied when initializing the spare module.

Recovery from Faults in HLMs

The HLM which serves as the system executive, and in a few cases other HLM's, must survive a fault without interruption in computations. In these cases a dedicated "hot" spare is assigned that concurrently performs identical computations to the primary module. At each RTI the two modules perform cross checks via the external bus to see if the other is functioning properly. Failure of either module triggers recovery by the "good" module. Should either active module detect an internal fault and disable its outputs, the other machine continues the ongoing computations. At a later scheduled time, the surviving machine activates a new "hot" spare and diagnoses the faulty module. Normally, only the primary module generates outputs and it gathers data via the bus for both itself and its hot spare. Under special conditions demanding extremely high reliability, two or more "hot" spares can be assigned to back up a critical modules and provide a greater degree of redundant protection.

Controlling HLM's have three major reliability functions to perform: computer redundancy management, module fault diagnosis, and system redundancy management. Computer redundancy management is the simplest function which has already been discussed. It consists of replacing a faulty computer module with a spare by issuing appropriate commands through the busing system. Module fault diagnosis consists of interrogating the faulty module to determine, if possible, the source of failure. In some cases, the module may be reusable by programming around faulty memory locations, or not using faulty circuits (such as a bad BA). An internal reconfiguration can, in some case, be commanded to rectify the fault -e.g., replacing a faulty memory module or replacing a faulty bit in memory.

System and Subsystem Redundancy Management - Automated Repair

System redundancy management must be provided by the System Executive HLM when a TM-controlled subsystem fails or if computers in the system run out of spares. These conditions result in the loss of one or more subsystem functions and require that the system enter some degraded mode of operation. Many systems contain a great deal of redundancy and can function without all of their subsystems. Spacecraft and avionics systems fall into this category because many functional backups are provided at the system level (e.g., redundant experimental mechanisms and redundant navigation equipment).

System redundancy management is implemented as applications software in the HLM's. Its design is highly application dependent and can only be accomplished in conjunction with the design of the host system [GILL 72].

While system redundancy is managed by the HLMs, subsystem redundancy management is normally carried out by software in the subsystem's embedded TMs. The individual TM is responsible for fault detection in its associated electromechanical subsystem. If redundancy is provided in the sensors and actuators of the subsystem, TM software will be responsible for substituting spare mechanisms to repair its host subsystem. If no redundancy is provided or spares are exhausted, the TM software commands its host subsystem to enter a "safe" disabled state and notifies its controlling HLM of the degraded conditions due to the failure of the subsystem. In either case, the TM records which devices in its host subsystem have failed and makes this information available, upon demand, to analyze the failure situation. In some cases, the TM may be able to predict failures in associated equipment by measuring drifts and making margin tests.

2.9 SUMMARY

In response to changing technology, spacecraft computing has progressed from a single expensive computer used as a shared resource to a network of much less expensive microcomputers in various subsystems. This has led to a number of new problems in fault-tolerant system architecture. We have attempted to deal with three classes problems: software and system complexity, hardware fault tolerance, and implications for LSI implementation.

The use of distributed processing in space has been retarded to some extent by the fear, on the part of potential users, of the high degree of complexity that can result. We have dealt with this problem by introducing constraints into the software, I/O, and intercommunications within the system in an attempt to simplify interfaces and augment testability. Synchronous communications, tree-structured control, removal of interrupts, granularity of I/O, and the synchronous foreground/background executive are all aimed at increasing testability, reliability, and ease of use at some expense in processing performance (response time and throughput). The design is tailored to the real-time spacecraft control and data handling problem, but we feel that it applies to a number of other dedicated real-time systems as well. It is unusual because we start with more hardware capability than is needed, and accept somewhat inefficient use of this hardware in order to achieve a more manageable system.

The hardware fault-tolerance approach is directed at minimizing "hardcore," achieving high coverage, and providing simplicity of operation. As previously described, the system uses independent intercommunications buses and independent clocks in the various computer modules to avoid some of these difficult hardcore problems. Each computer module checks itself and can be relied upon to disable itself in case of failure. Fault detection is implemented in hardware and uses self-checking circuitry. Self-checking hardware design is currently of wide interest because it offers high coverage and is relatively inexpensive to implement with LSI technology [SEDM 80, CART 77]. Finally, the recovery mechanisms are simple. For most modules it is adequate to switch to a backup spare, followed by a reload and restart operation. Critical functions are run by a pair of computers, each of which is self-checking. We feel that simplicity of fault-detection and recovery functions is important to achieve widespread use of fault-tolerant computing. The potential user must understand a system before he will buy it, and fault-tolerant computing is too often viewed as an exotic, risky technology by these users.

An important result of this program is the determination that a wide variety of fault-tolerant distributed computer systems can be constructed using commercial microprocessors and memories, and a small set of LSI building-block circuits.

3.0 FAULT-TOLERANCE ISSUES IN THE NEXT GENERATION OF LONG-LIFE, REAL-TIME SYSTEMS

Multicomputer systems are potentially more cost-effective than large supercomputers for problems such as certain classes of scientific computations, discrete event simulation, military command, control, and signal processing. Many of the most advanced experimental computing systems sponsored by government and industry fall into this class (e.g. RP3, AOSP, Butterfly, Hypercube, Ultramax). These machines use tens to hundreds of high speed processing nodes and can be characterized by their interconnection structure. They fall into four general categories: i) multiple bus connected (AOSP), ii) collections of shared memory multiprocessors (Multimax), iii) shuffle and exchange network-connected (Butterfly), and iv) hypercube connected.

Fault-tolerance is a serious problem in large systems of this type (e.g. 64-1024 processors). Due to their high complexity and the use of very high speed circuits, transient errors are expected to occur during long computations, and occasionally a processor will fail. Many of the applications proposed for these systems cannot tolerate computational errors due to hardware faults. With applications that involve real-time control, it is often unacceptable for the system to be down for long periods for recovery and reconfiguration. Often there is a large data base of critical information that must be protected under various fault conditions. Furthermore, in space-borne systems repair is often impossible, so the system must be designed for long-life and be capable of fast self-repair.

None of the current generation of high performance multicomputers are designed for fault-tolerance. In order to achieve fault tolerance, it is necessary to add concurrent fault detection capabilities within each processing node and high-coverage fault-recovery mechanisms. Concurrent error detection is necessary to prevent erroneous outputs from leaving a node, thus preventing the damage from propagating to the rest of the system and damaging critical system state data. Confining the damage to system state is necessary for dependable error recovery.

Over the last several years we have been examining the issues involved in implementing fault-tolerance in these high-performance distributed systems at both the system architecture and node levels. The following is a discussion of areas that we have been studying. The first part addresses high-level system issues in implementing fault-tolerance in highly parallel processors. It is a study of one type of these architectures that is actively being developed -- the binary hypercube [RENN 86a]. The second part addresses lower level design issues of implementing fault-tolerance in the processing nodes. This includes implementation of building-block self-checking computer modules as nodes in these high-speed systems, and the use of self-checking, self-exercising logic design to provide concurrent error detection and to rapidly expose latent errors [RENN 86b].

3.1 FAULT-TOLERANCE ISSUES IN LARGE BINARY HYPERCUBES

This section examines some of the issues involved in implementing fault tolerance in hypercubes for real-time applications. Many of these issues are similar for the other different high-speed multicomputers listed above. The hypercube is attractive because it provides a high degree of connectivity while only having log N ports per processor. This makes it possible to build very large ensembles of machines. Intel currently has a hypercube commercially available, and JPL is building a large very high speed hypercube system. Before addressing fault-tolerance issues, it is necessary to examine the hypercube architectures.

3.1.1 HYPERCUBE DESCRIPTION AND INTERCONNECTION STRUCTURE

In a binary hypercube there are a number $K = 2^n$ of computers, and each is given a unique address (or identification number) which is one of the K, n-bit binary numbers. There is a point-to-point connection between every pair of computers whose identification numbers are separated by a Hamming distance of one. Several binary hypercubes are shown in Figure 20. The size of a hypercube grows as powers of two. Doubling a hypercube consists of creating an identical structure and connecting each corresponding node. This can be easily seen from the following explanation:

Given a hypercube of n elements, a single element has some address: 101001...10. It is connected to n neighbors, each identified by inverting one bit in its address (001001...10, 111001...10, etc.,). If the size of the cube is doubled to 2^{n+1} elements, then we can view the original cube as half of the new cube with original addresses prefixed with a leading zero, and the new part of the cube as the computers which have addresses with a leading one. If we take an element in the original cube and the corresponding element in the new part, they have addresses:

0101001...10 old half
1101001...10 new half

Clearly, corresponding elements in each half have similar connections within its own half and one additional connection to the corresponding element in the other half (inverting the most significant bit).

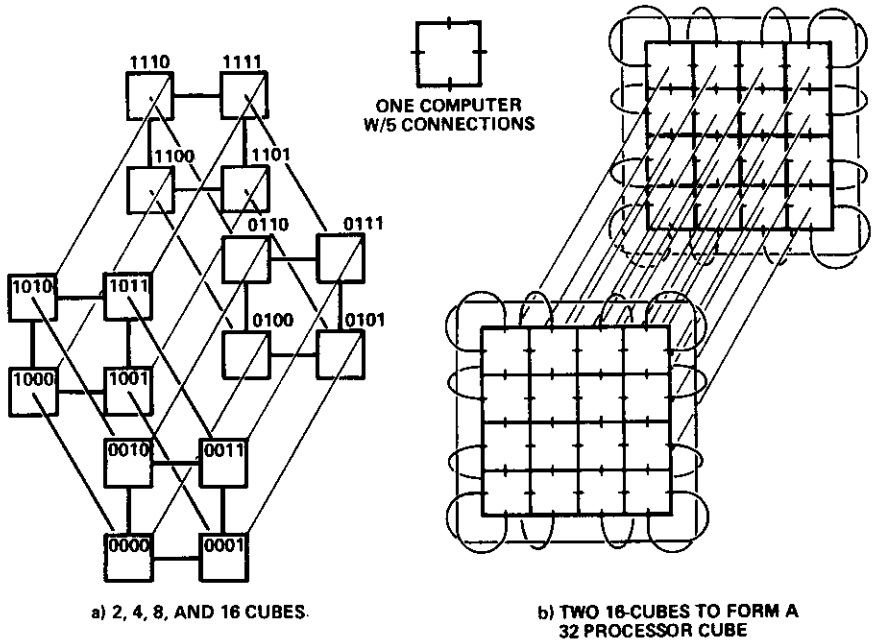


Figure 20: Hypercube Structures

Figure 20a shows the expansion from two through 16 processors, and figure 20b shows a 32-element cube composed as two 16-processor cubes (which correspond to the layout of a Karnaugh map). Larger cubes are harder to visualize, but a large hypercube contains within its interconnection pattern a single two-dimensional array and a set of two dimensional arrays connected as a three dimensional grid. This is one reason why they are so useful for physical simulations.

The JPL Mark III Hypercube

The JPL computing system is connected as a binary hypercube which can be expanded to 1024 processors. A typical processing node contains two M68020 processors. one for I/O control and one operating as the main processor. The main processor and I/O processor each have a fast static RAM serving as a local cache, and both communicate through a 4Mbyte dynamic RAM [JPL 85].

Each (one of log n) I/O connection contains a FIFO buffer to receive incoming messages. The I/O processor can interpret incoming messages, and it can either store them in the shared memory if addressed to the local node or re-route them if addressed elsewhere. The main memory bus supports 12 Mbyte/sec data rates so one can assume a maximum of about 1 Mbyte/sec data rate worst case on the I/O links if each of 8-10 channels are receiving messages simultaneously. Messages are passed between computers as a sequence of 8-bit bytes. The computing node is designed to allow a pipelined floating point chip set to be added onto the internal bus.

3.1.2 POTENTIAL HYPERCUBE FAULT-TOLERANCE REQUIREMENTS

The first information needed to design or evaluate a fault tolerant machine is a specification of the errors and faults that it must tolerate and the requirements for fault-tolerance. For purposes of this discussion fault-tolerant applications will be divided into two general categories: (1) applications where maintenance is available, and (2) applications such as space systems which require very long unattended life.

A. Defining the Fault-Environment

To define the fault types the hypercube designers must identify: 1) randomly occurring physical (permanent and transient) fault-types, 2) potential external disturbances and the resulting computer fault-types, 3) possible design fault-types, and 4) fault-types that may occur due to operational errors. For each fault-type, operating environment, and for the technologies planned for implementation, it is necessary to develop a methodology to estimate: i) rates of occurrence, ii) distribution in space and time, and iii) duration of the various fault-types.

B. Defining Fault-Tolerance Requirements

Fault-tolerance requirements which must be defined include:

Identifying the minimal service under fault-conditions

Identifying critical state information which must be preserved under fault conditions

Establishing the maximum acceptable time delays while fault recovery is taking place

Specifying reliability, the probability that the system can meet its service specification for the duration of the mission. Typical space reliability requirements are 0.95 over a 5-10 year mission

C. Typical Hypercube Fault-Tolerance Requirements

A "typical" set of fault-tolerance requirements are given for this study and are listed below:

1. There is a critical data base which must be protected under all fault conditions, and it is expected to be at least several hundred thousand words distributed through many of the hypercube nodes. This information must be redundantly stored to prevent its loss by a single failure.

2. There is a real-time recovery requirement in the range of milliseconds to several seconds.

3. Very high "coverage" must be provided in error and fault recovery to achieve needed reliability. (Given any error or fault in the specified set, there must be a probability of much greater than 0.99 that it will be recovered in the specified time and without damage to the critical data base.

4. Very high throughput is needed, and this requires efficient use of redundant resources. Redundant elements should be put to use whenever possible, making the general use of massive redundancy (e.g. triplication and voting) unacceptable. For space applications a large amount of redundancy must be carried on-board to assure that the complex system will work for many years (when many permanent faults are expected to have occurred). This requires a finer partitioned design and a much more efficient use of redundancy than on equivalent ground computers. This problem is compounded by the generally lower density of radiation hardened parts required in space. More chips are needed to do the same job.

3.1.3 CONCURRENT ERROR DETECTION AND RAPID FAULT DIAGNOSIS

Since the goal of parallel computers is to apply as many processors to a problem as is possible, standby redundancy (i.e. running each program in one computer in contrast to massive redundancy approaches such duplication or triplication and voting) is needed to maximize the amount of hardware available for parallel computations. For long-life unmaintained applications this is especially important since minimizing the number of modules needed for a working end-of- life configuration also reduces the number of redundant spares needed to achieve a specified reliability. Concurrent error detection is required if rapid, dependable transient fault recovery is to be expected, and if permanent faults are to be detected before damaged information propagates to many places making recovery very difficult.

a) Issues of Designing Concurrent Fault Detection into the Hypercube Nodes

The current hypercube nodes only employ SEC/DED codes in memory, but do not provide comprehensive error detection in other parts. High-coverage, concurrent fault detection can be added by implementing circuits within each computer to provide comprehensive checking. To prevent error-damaged information from upsetting critical state information in memory these checks must be made at nearly every clock step. The techniques for doing this are well understood and typically rely upon SEC/DED codes in memory, error detecting codes on internal buses, and a mixture of coding and/or duplication in the processor and I/O logic [SEDM 80, RENN 81b].

In very high speed machines, this checking can slow down the computer significantly because every cycle must be delayed for checking to be completed. Error checking will result in an unsatisfactory slowdown unless special design techniques are employed. This speed penalty can nearly be eliminated by introducing pipelining into error checking. This is already done in large commercial processors, but it has not been done in the VLSI microprocessor technology for hypercube-class machines. It will require innovative new designs.

Pipelining error checking involves not waiting for error checks and carrying out operations before checking of the operands is complete. If an error is detected, it is necessary to back out of those operations with erroneous data. This is done by providing buffering and control in the processor, memory, and I/O needed to restore the node to the state it was in before the error occurred.

b) Rapid Fault Diagnosis and Uncovering Latent Errors

In some projected environments the hypercube may be expected to survive massive externally induced transient faults and it will be necessary to provide the capability of very rapid self-diagnosis in each node. After a disruption, volatile information will be lost, and some of the nodes may have suffered permanent faults as well. In order to recover within several seconds or less, each node will be required to quickly and thoroughly diagnose itself and report its condition in order to allow reconfiguration to a working set of nodes before restarting the computations.

If the system is expected to operate under an externally induced high rate of transient errors, self-exercising features will be needed to rapidly uncover latent faults and errors (faults and errors which are in circuits not currently being used and will only be detected later when the circuit is called upon). These errors must be corrected quickly before multiple errors build up, jeopardizing recovery.

Both rapid diagnosis and self-exercising capabilities require specialized VLSI design and are an area of current research [RENN 86b] that will be discussed later.

c) Verification of Communications

Current hypercube designs implement inter-computer communications as messages which are sent between buffers in computer modules and relayed between modules by software to their various destinations. This can result in considerable latency between sending a message and its arrival at a destination. In a fault-tolerant environment, it is often necessary for a sending computer to quickly verify that the message was received correctly at its intended destination. If a computer must acknowledge receipt of a message before sending additional ones, latency can be increased several times over. If a message fails to arrive or an acknowledge message is lost due to a transient fault, the message must be retransmitted. A time-out is needed in the sending processor for it to re-try the message, and a sequence number is needed in the receiving processor so that if the acknowledge message was lost, the receiver will not accept the retransmission as a new message. The whole issue of verification of communications must be carefully analyzed in the hypercube to determine if the latencies of error recovery are compatible with the real-time recovery requirements of many applications. Hardware implementations of checking and automatic message acknowledgements are probably essential to achieve acceptably small message latency in a fault-tolerant system and to meet real-time recovery requirements.

3.1.4 PARTITIONING AND ALLOCATION OF REDUNDANCY

Several approaches have been considered for implementing redundancy within the hypercube structure.

A. Use of Natural Redundancy Within a Hypercube (HC)

It is necessary to have spare processors to take over the tasks of those which have failed. One approach is to allocate processors as spares within the existing hypercube network. Figure 21 represents a hypercube which is executing a number of independent processes, (A,B,C,..F) each of which use a relatively small portion of the array. Spares are allocated at the periphery of groups of processors which are running these processes. If a computer or interconnection path fails, a nearby spare is activated and messages are re-routed to the spare. After spare processors are no longer available (or the number of failed communication links result in inadequate bandwidth), the system reverts to a degraded operational mode.

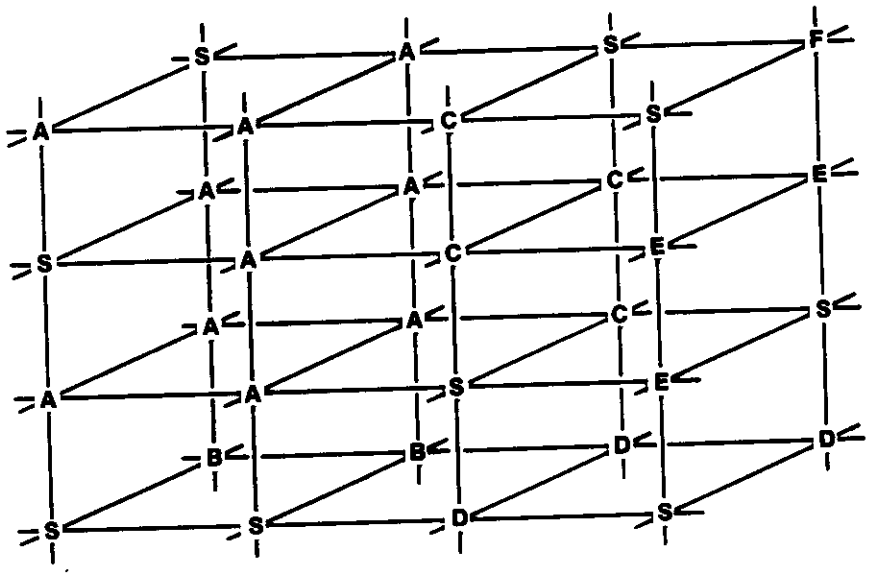


Figure 21: Allocating Redundancy Within a Hypercube

The allocation process can be highly complex in long-life unmaintained systems where a large number of computers fail, and groups of processes are re-located to different areas of the cube to maximize the remaining connectivity and communication efficiency. This approach has the serious limitation that recovery from permanent faults changes the topology of the machine. If a long-life application requires most of the computers or requires the use of most of the available bandwidth and the hypercube's unique interconnection structure (e.g. FFT) to run at high speeds, this approach will not work. It will fail because of the increasing length of re-routing messages around failed processors and bandwidth congestion resulting from sharing the remaining available paths.

B. Sparing to Maintain Topology and Performance in Maintainable Systems

The following approaches have been identified to provide tolerance of a limited number of faults in processors and communications links while maintaining the original performance and connectivity of the processing network (at least until spare hardware is exhausted). They are limited to maintainable systems because sparing for large numbers of faults that might occur in long-life unmaintained systems is both inefficient and overly expensive. Design approaches for long-life systems are discussed in a later section.

Adding Spare Nodes and Links

In this approach spare computers are added and treated as machines in an additional dimension of the hypercube. To be practical, this next dimension is not fully populated, containing but a small number of spare nodes. For example, if the basic hypercube has dimensionality of N (2^N processors), all of the machines have $N+1$ serial ports. The extra port is redundant and is used to reach spares as in the example shown in Figure 22. Each of several spare processors has a set of associated VLSI crossbar circuits attached. When a machine fails, its neighbors address messages through the $N+1$th port to the "spare dimension." The VLSI crossbar circuits are set up to connect the spare machine to the neighbors of the failed machine. Several spares can be implemented as shown in the figure by connecting their crossbar inputs in parallel.

Figure 22 is a example which shows a system with 128 processors, each having seven serial ports and an extra "eighth" port connected to the spares via the crossbar switches. Each port is bidirectional and contains four wires (two request lines a bi-directional data line and a clock line). Each spare processor contains eight VLSI crossbar chips which connect sixteen 4-wire ports to up to four of the seven ports on the spare computer. Each crossbar chip requires a few thousand gates (pass transistors and three-state drivers) and can be implemented with less than 96 pins. (If each crossbar is connected to a P-processor subcube of the hypercube, it is easily shown that for any failure, at most log P ports from a single crossbar will need to be connected to the spare computer. This is because any failed computer in a subcube is connected to exactly log$P$ other computers in that subcube. In the example P = 16 and log$P$ = 4.)

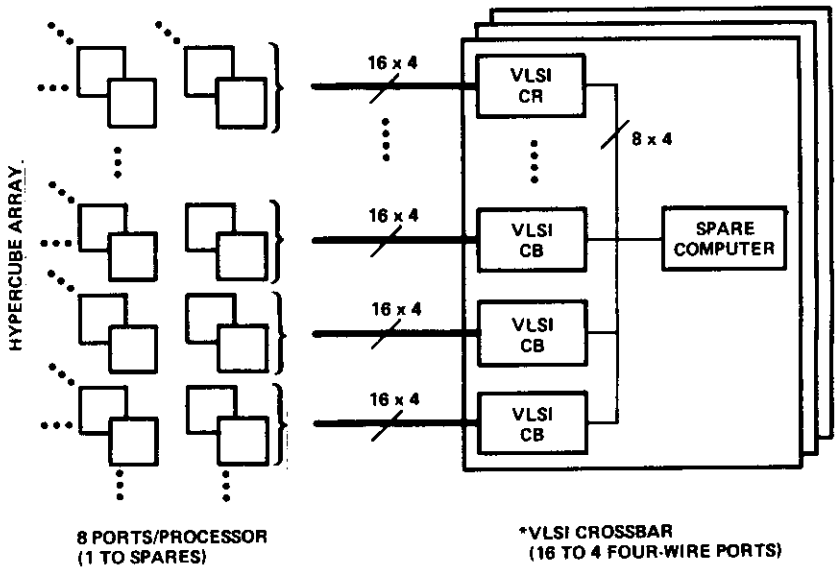


Figure 22: Spares Added as an Incomplete Next Dimension

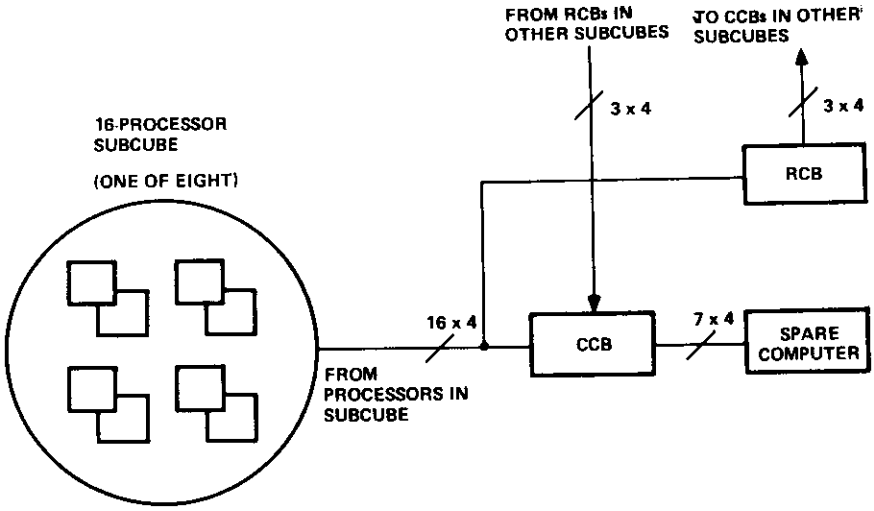


Figure 23: Sparing of a Subcube (One of Eight)

Hierarchic Sparing by Subcubes:

A hypercube with 2^N processors can be viewed as a hierarchy of 2^s sub-cubes, each with 2^m processors (where $s + m = N$). It is possible to provide one or more spares for each subcube reachable, as before, through extra ports in each processor). Figure 23 shows how this is implemented in a 128 processor binary hypercube ($N = 7$) with eight subcubes ($s = 3$) of 16 processors ($m = 4$). If any processor fails in the subcube, the spare will have to be connected to m neighboring processors in that subcube, and one processor in each of s external subcubes to which it is connected.

In the implementation of Figure 23, each spare computer contains two crossbar switches (each implemented as two or more chips). The Connection Crossbar (CCB) has $2^m + s$ inputs and N outputs. It is connected to each of the 2^m processors in its associated subcube (via. their spare port), and it is connected to one output of a Relay Crossbar (RCB) from each of the other s subcubes to which its host subcube is connected. (Remember that although there are 2^s subcubes, any single subcube is only connected to s of them.)

Each subcube contains a relay crossbar which has 2^m inputs and s outputs. The inputs are connected to all computers in the subcube, and each output is routed to one of the s subcubes to which it is connected. Each of the external subcubes can establish one connection to any single processor in the host subcube.

When a computer fails in a subcube, its CCB connects its spare processor to the m processors in the subcube which have connections with the failed processor, and it connects the lines from the adjacent subcubes to the spare processor. The RCBs in the adjacent subcubes are commanded to connect to the processor in their subcube which was connected to the failed processor.

The two approaches described above which use spares in the N+1th dimension, allow spare computers to back up any machine in the hypercube and offer i) a simple reconfiguration algorithm, and ii) low communication delays when a failure occurs. They are recommended for applications which allow periodic maintenance, but are inadequate for long life unmaintained applications where more than half of the processors may fail over a five to ten year period.

The hypercube's point-to-point interconnection structure limits sharing of spare computers or communications paths. In section 3.1.7 we will look for alternative interconnection structures which provide much the same functionality, but allow better utilization of spares.

LONG-LIFE UNMAINTAINED SYSTEMS

Long-life hypercubes will be much more difficult to design. It is necessary to add a much larger amount of redundancy, and the system must be structured so that the redundancy can be used in a very efficient fashion. The system must be finely partitioned so that no single module-type has a high probability of failure, and a typical redundant spare part must be capable of backing up a large number of working parts. Typical space system requirements are a 95% probability of surviving five years or more. In typical space systems it is necessary to more than double the hardware (i.e. start with at least as much redundant spare hardware as active hardware) to provide adequate assurance of full performance at the end of mission.

Current hypercube designs for ground-based applications are ill- suited for long life unmaintained environments. The individual computers have failure rates which are too high for long life applications. In maintained ground systems several megabytes of storage may be implemented in each computing node using large dynamic RAMs. In the long-life unmaintained space environment, special radiation hardened, static devices must be used, resulting in much lower density circuits (and requiring an order of magnitude more parts for the same computing capability). A space computer with several megabytes of memory would currently have a five year reliability of less than a few percent, making sparing for .95 reliability impractical [MIL 217D].

Multi-Level Redundancy

To achieve a long-life with highly complex nodes, each computing node must be made internally redundant. Table 1 was computed as an example using the Bouricious, Carter, Schneider reliability model to show the approximate end of life reliability required of a single computer module in an array in order to achieve an array reliability of 0.95 when 50%, 100% and 200% sparing is employed [BOUR 71]. Coverage is optimistically assumed to be one, and communication links are assumed to be fault-free, making this an upper bound. Clearly, if adequate reliability cannot be achieved using this model, there is no hope since the actual reliability will probably be much less. The ratio of powered and unpowered failure rates are assumed to be one -- which is a reasonable approximation with low power CMOS-SOS logic.

#Active Computers	#Spares/ SCR	#Spares/ SCR	#Spares/ SCR
16	8/.78	16/.57	32/.45
32	16/.78	32/.57	64/.40
64	32/.74	64/.58	

*SCR End-of-life Reliability of each individual computer required for an ensemble reliability of .95

Table 1: Reliability of Single Computer Required for an Array Reliability of 0.95 with 50%, 100% and 200% Redundancy

As a rule of thumb, we can say that each computer of a fault-tolerant hypercube should have at least a 50-60% probability of working at the end of mission, otherwise an inordinate number of spares will be required. Furthermore, if the end-of-life reliability of the individual computers is increased, the number of spare computers required drops dramatically. The critical question is to determine if the individual computer modules can be built to achieve this level of reliability at the end of five or ten years. The answer is that either i) the computers must be very simple to keep the component failure rate acceptably low, or ii) if high performance computers are used, their failure probability will be unacceptably large unless internal redundancy is be employed within each of them.

Due to the fact that high performance computer nodes are needed which will contain a hundred or more chips, it is nearly certain that their failure rates will be too large to attain acceptable reliability without the use of internal redundancy. An upper bound can be obtained by assuming a very optimistic failure rate for a single VLSI chip is on the order of $.2 \times 10{-}6$ (one failure every 600 years). A computer with 150 chips would have a composite failure rate of $3 \times 10{-}5$ per hour, not counting buses, power supplies, clocks etc. The expected five and ten year reliabilities of a non-redundant computer (e-LAMT) are:

5 Year Reliability	10 Year Reliability
.26	.07

We have conducted extensive modeling studies using ARIES which show that only a moderate amount of internal redundancy is required in typical medium to high performance flight computers to greatly improve their end-of-life reliability. A typical high performance machine was modeled, and it was shown that its reliability could be improved from .08 to .75 by adding an additional 50% redundancy to the computer. (A Hamming Code and two spare bit planes were added to memory. The majority of circuit complexity in a computer module is in memory, and, due to its already fine partitioning, one can protect against multiple chip failures with a proportionately low increase in redundant hardware -- typically 10-20% [RENN 81b]). This study examined the tradeoff between employing redundancy at various levels in a multi-computer system, and it was shown that distributing redundancy at multiple levels, chip, functional unit, and computer, results in greatly improved reliability for a given amount of redundancy in a long-life unmaintainable computing system [RENN 82, DEPA 83]).

Thus a long-life hypercube architecture is expected to have computer nodes which are internally partitioned with backup spares provided. Each computer will require spare bit-planes in memory to replace failed chips, and possibly spare memory modules, and CPUs. Then it will be necessary to add a number of spare computers forming a system with several levels of redundant protection.

3.1.5 ERROR AND FAULT RECOVERY

In a complex multicomputer structure, error and fault recovery should be handled in a hierarchic fashion. In general, errors and faults should be handled at a low a level as possible to minimize system-wide disruption while fault recovery is taking place.

Low-level Recovery: Errors and faults which can be corrected quickly within a single node will not seriously disrupt the ensemble of other nodes. This is already done with SEC/DED codes in memory. It is also possible to handle most transient faults faults locally within a processor node using program rollback.

It is also possible to correct some permanent faults locally within a node relatively inexpensively. Examples are substituting spare bit planes in memory and replacing a defective processor and I/O circuits.

Higher-Level Recovery Actions

There are errors and faults which cannot be recovered locally within a node. These must be recovered at a higher level -- either by a group of nodes or as a system-wide recovery action. These fault conditions result in loss of state within a node requiring its re- initialization from the outside or loss of the node entirely due to an unrecoverable fault.

i) Loss of State Within a Node - Critical state must always be protected against single point failures. This usually means that programs and critical memory data must be stored in at least two nodes. Then, if a node fails, its information is still available elsewhere. In the hypercube this means establishing rollback points in each program and saving the state necessary to recover computations both in the operating module and a neighboring module.

ii) Node and Link Failures

When nodes or links fail, a spare must be activated and the programs and state information loaded into it to restart processing.

In all cases where nodes must be reinitialized, there will be a time delay which will affect the operation of the whole hypercube. Typically, real-time applications programs in all the other nodes must be written with alternative procedures to compensate for the time delays. If a node has failed and a spare substituted in its place, other nodes will have to be appraised of the change. If spares are not available, processes will have to be reassigned to provide degraded operation. This type of high-level recovery with system-wide effects is best managed by a single high- level maintenance processor (as is done in most large commercial machines nowadays.

There are several good reasons for controlling fault-recovery from a fault-tolerant computer external to the hypercube processor array. First, whereas the processors in a hypercube are designed for high performance, a processor for recovery management can be relatively simple, have an inherently low failure rate, and be designed for a very high degree of fault tolerance. Second, distributed fault recovery algorithms are quite complex and rely upon obtaining diagnostic information from several sources. If this is placed in the hypercube computers, a considerable overhead is expected in these machines as extra programs and cycles devoted to diagnostic testing. An external centralized recovery processor will remove overhead from the hypercube array and can implement simpler centralized recovery algorithms. This approach is in agreement with the approach taken in most modern complex computing systems in which a maintenance processor is added for fault-recovery management. In space systems, there is usually a separate, very reliable processor for executive housekeeping and executive functions. Such an executive processor is an excellent candidate for recovery management in high performance array processors such as the hypercube.

In the architectures described above, a maintenance processor can be added and connected to the array processors by a redundant set of serial low-speed maintenance buses through which it can collect diagnostic information, command reconfigurations, and initialize spare units when they are activated. If self-checking processors are employed in the hypercube array, the maintenance processor will be explicitly notified when a computer fails, and can execute system diagnostics, compute recovery strategies, command reconfiguration, and reinitialize the system to effect recovery.

3.1.6 FUNDAMENTAL PROBLEMS

Several fundamental problems have been identified in using Hypercube structures for real-time aerospace applications:

Efficient Use of Memory in Heterogeneous Distributed Processing

In attempting to map heterogeneous applications onto homogeneous computer arrays there is often a serious mismatch. Typically, there are a variety of computations to be performed, each requiring different speeds and different memory sizes. Different sensors may provide several high rate data streams whose processing programs require dedicated computers to achieve adequate throughput, but the memory size may be small for each computer. Other processes may require less speed but need an order of magnitude more memory to carry out their computations.

Here we have conflicting requirements. For the purpose of fault-tolerance we would like to have all the computers the same so that spare computers can serve as backup for all machines which have failed -- allowing processes to be reassigned to any other computer when one fails. It is also essential to keep memories as small as possible because they represent the major proportion of the system's failure rate, but it is necessary to size each computer with a maximum size memory to allow efficient sparing.

Improving Connectivity for Sparing

System reliability is optimized if spare computers or communication links can be used to back up a large number of active units. We saw in the examples above that the hierarchic approaches to sparing by adding spares to subcubes only allows the spares to back up nodes within their associated subcube. Many multiple failure conditions can exist in which the existing spares are unavailable.

Clearly, if the topology of the connections between spares and active units is constrained a much lower reliability can be expected. A typical redundant spare part should be capable of backing up a large number of working parts. The hypercube's point-to-point interconnection structure limits sharing of spare computers or communications paths. Thus we should examine alternative interconnection structures which provide much the same functionality, but allow better utilization of spares.

Reducing Communication Latency for Real-Time Recovery

Message passing between nodes is a relatively slow and inefficient mechanism for replicating state information to allow recovery if a memory or processor fails. Errors in messages which are relayed between several nodes may also involve unacceptable latency for error detection and recovery.

Thus we must examine alternative architectures and interconnection structures to see if they can better meet requirements of efficient hardware use, improved connectivity for sparing, and short latency.

3.1.7 ALTERNATIVE ARCHITECTURES

One alternative approach uses interconnection paths with high enough bandwidth to support several communications circuits and thus kill two birds with one stone -- reduce the number of interconnections and make it possible for spares to back up several working units and improve the life extension capabilities of the spares. This appears to be an attractive alternative if high-bandwidth paths are used.

In designing a system with shared communications paths, it is necessary to determine the maximum rate needed by any one "virtual" circuit using the shared path. Given a computer module which runs at 2 mips, we can assume that it will take at least 10 instructions (5 usec) on the average to do anything useful with a data word coming in. If this data is in the form of 32-bit words with 25% overhead for addressing and checking, (and if communications can be largely overlapped with processing) the bandwidth at which the processor saturates will be approximately 8 megabits. If a 100 mbit shared bus is available (optical serial bus or backplane bus) a dozen processors can be supported on a single physical path, giving them interleaved time slots to create virtual point-to-point communications between all of them. (Thus maintaining a close approximation of the original hypercube functionality.) Spare processors on this shared structure can back up all the active units, and redundancy within the communication links can provide improved reliability for very long missions.

An Example of a Potentially Improved Implementation

The preceding discussion justifies the need for employing multi-level redundancy, achieving better memory utilization (balance), and finding an interconnection structure which allows greater sharing of spare resources. One step in this direction is to take advantage of the hierarchy inherent in the binary hypercube structure. The approach is to use of a hierarchy where sets of computers are combined into multiprocessors, and then the multiprocessors are combined into an array. Each of the processors still has a unique address and from the viewpoint of applications software, this logical structure is identical to the original hypercube. The multiprocessor can provide better memory utilization. By sharing memory, different processes with varying memory requirements will better fit into a smaller memory. Also if the processors share code or data, a single copy can be kept rather than the separate copies required in independent computers. (This also allows sharing of power supplies, boards, clocks, etc. and leads to a more efficient use of hardware. Also, since the system is designed for special purpose problems that are partitionable into separate computers, memory contention problems should be easily solved with caches attached to the individual processors.)

Figure 24 shows a group of four processors (plus one spare) packaged as a multiprocessor for use in a 64 processor (16 multiprocessor) hypercube. Each processor has an associated local memory of 8-16 K words and there are four communications ports connected in hypercube fashion to four of the 16 additional multiprocessors. Each port must have four times the speed of the original hypercube interconnects, and becomes more complex because it must recognize and route messages addressed to four processors in the module. In order to prevent latency problems it is advisable to allow messages to be interleaved on a word-for-word basis through each port. Again, notice that the hypercube can still be mapped onto this structure, it only provides additional flexibility in communications and sparing within each multiprocessor. Spare processors, ports, bit-planes in memory, and memory modules can be employed. Port connections are byte-serial (e.g. 4-wires) with spares to circumvent a failed driver or receiver.

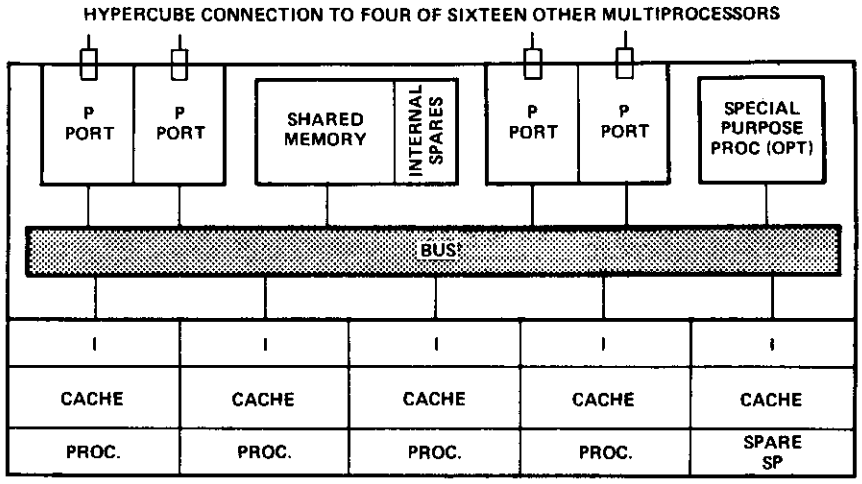


Figure 24: Redundant Multiprocessor for Use in a Large Array

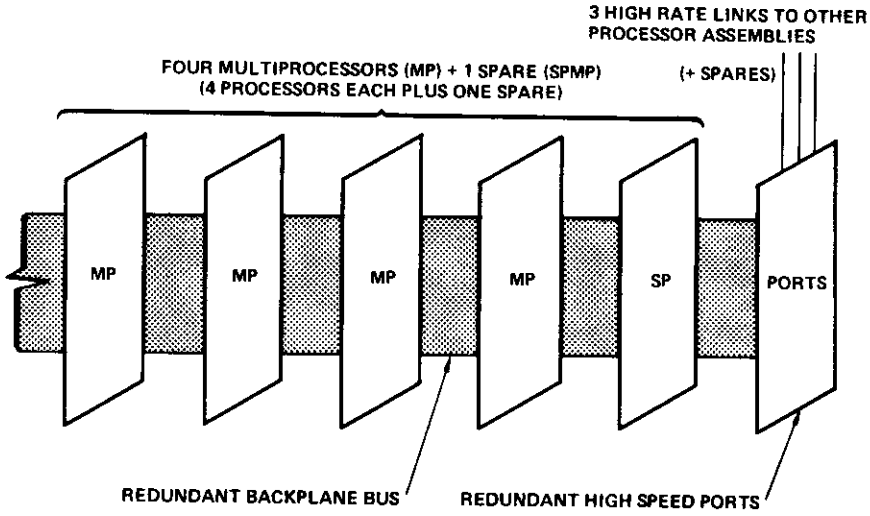


Figure 25: Redundant 16 - Processor Assembly for Use in a 128-Processory Binary Hypercube

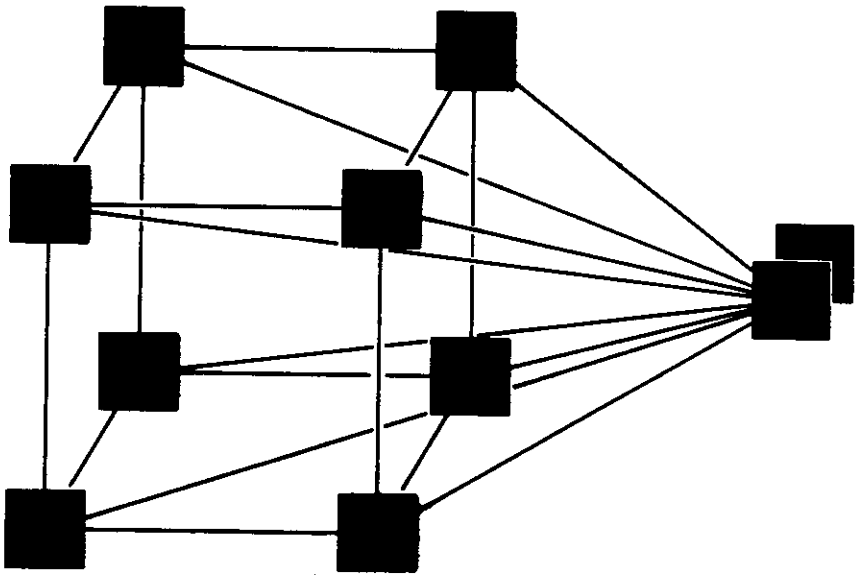


Figure 26: Connection of Processor Assemblies with Spares

This approach can be extended to an additional level of hierarchy to provide better packaging and use of spares and active resources. Figure 25 shows a 16-processor assembly made up of redundantly protected groups of 16 computers (four multiprocessors plus one or more spares). Here we have four multiprocessor boards plus two spare boards connected by a redundant backplane bus. Three ports with individual standby spares connect via. very high rate (128 mbit) links to three of the remaining eight 16-processor assemblies (in hypercube fashion) which make up the 128-computer system. Since the high level cube has a small number (8) of 16-processor assemblies a redundant processor assembly may be added and connected to three of the others as shown in Figure 26. If any assembly fails, message re-routing will never require more than one additional hop.

This configuration still provides a hypercube structure but is much more general. Data can be routed between processors within a 16-processor assembly entirely by hardware, and in the worst case, messages between any two computers in the 128 computer system must only be forwarded through two processor assemblies. Use of the backplane bus allows distribution of very high rate input data to a 16-processor assembly and the flexible routing of very high rate data through special purpose processors that can optionally be included in the multiprocessor modules. Now we have redundancy at four levels: spare chips within memories, spare modules and processors in multiprocessor modules, spare multiprocessors, and the ability to reassign processes within the overall hypercube. It is this type of multi-level approach to fault-tolerance (all though not necessarily the examples given above) that will provide both the long life and flexibility needed for space applications.

Preliminary reliability models have been made of this hierarchic structure. The approach is to work backwards. Using combinational models, the required reliability of a 16-processor assembly is determined to achieve the system reliability (of 0.95). Then the multiprocessor board reliability is determined to achieve the needed processor assembly reliability. Then sizing and use of internal redundancy within the multiprocessor is chosen to meet the needed level of reliability.

3.2 SELF-CHECKING SELF-EXERCISING DESIGN IN PROCESSOR NODES

This section addresses lower level design issues of implementing fault-tolerance in the processing nodes. It is clear that concurrent error detection is needed in processing nodes of highly parallel multicomputers. Self-checking design techniques can be employed to provide concurrent error detection, as was discussed above in the SCCM. Although they detect errors when they occur, self-checking designs cannot detect fault conditions which exist but have not produced a detectable error. (An example is a word in memory that has not been accessed. Another example is a "stuck-at-one" bit in a register which currently contains a one but will fail when a "zero" is stored there.)

This section describes one result of ongoing research into the design of self-checking computers which are capable of exercising themselves during normal operation in order to uncover latent faults very rapidly. The approach is unusual in that it is carried out from the prospective of the system architect. The generic structures which are commonly used to build self-checking processors and memories are examined along with typical VLSI layouts (e.g. memory arrays and interfaces, CPU data paths, microprogrammed controllers, various self-checking checkers). Techniques are introduced to interleave self-testing with normal program execution cycles at a rate which has minimal impact on circuit layout and speed of ongoing computations (typically a 5-10% slowdown). The results obtained so far indicate that if self-checking features are already in place, additional hardware mechanisms can be added at relatively low cost which enable the computer to test itself during normal operation.

A self-exercising memory design is presented which allows single-bit errors to be detected and corrected within milliseconds. Furthermore, extensive tests for stuck and shorted cells as well as memory disturb tests are carried out many times per second. Modeling results are presented which show this approach to be highly effective in very high-transient environments. Self-testing approaches in a preliminary design for the data path portions of the CPU are briefly discussed.

Self-exercising self-checking design has several important applications. High transient fault rates are expected due to radiation in some space applications where errors must be "flushed out" very rapidly to avoid multiple error buildups which jeopardize fault-recovery. For ultra-reliable applications such as those planned for SIFT and FTMP, the need for rapid removal of latent faults is needed to achieve extremely high levels of coverage that are needed [WENS 78, HOPK 78]. Finally, if a machine can fully exercise itself during the first second or two of normal operation, acceptance testing is built-in. In space applications, acceptance testing is a very large expense which can be reduced by more effective self-tests within flight computers and associated subsystems. The type of design described here is planned for the next generation of the FTBBC computer previously reported.

3.2.1 A SELF-CHECKING SELF-EXERCISING MEMORY ARCHITECTURE

In many computers used for flight and real-time control, the random access memory system is often the most complex component part, containing many more active circuit elements (transistors) then the rest of the computer. Therefore the majority of transient and

permanent faults are expected there, and this is the area where self-exercising self-testing logic can be most effectively used. It is common to employ SEC-DED codes in memory for single error correction. The codes only are applied to words which are read out, so it requires additional software tests in conventional machines to "flush out" latent faults in words not often exercised by application software. These tests can be scheduled as a background processes and may take many minutes to complete.

By taking advantage of the 2-1/2D structure of modern RAMs, and adding a modest amount of additional logic, tests can be implemented in hardware and executed about a thousand times faster. This testing can be interleaved with normal processing with very small speed penalties. Transient faults (sometimes called single-event upsets) can be detected and corrected within milliseconds after they occur. Permanent faults, including stuck-at-one, stuck-at-zero, coupling (e.g. shorts) between neighboring cells, and some pattern sensitive faults can also be quickly detected (within tens of milliseconds) after they occur.

Memory System Organization

This approach applies to memory systems which employ individual memory chips organized as N one-bit words. This is typical of most modern memory chips which contain 16Kx1, 64Kx1, or 256Kx1 bits. It is assumed that the memory system is structured to contain M+P bit words, with M information bits and P parity bits to implement a Hamming SEC/DED code. Each bit position in all words is stored in separate chips so that any single chip failure will at most damage one bit in any word.

A block diagram of such a memory system is shown in Figure 27. An additional Memory Interface Building Block (MIBB) circuit is used for control and to provide Hamming code encoding, decoding and single error correction. If a single error is found in a word being read out, the MIBB corrects the error and stores the corrected value back into the memory. A MIBB of this type has been developed at JPL and reported previously [RENN 78a]. (The JPL design, which has been implemented, can also substitute spare bit planes [groups of chips holding a particular bit in all words] for ones that have failed.)

In the approach presented below, both the memory chips and MIBB are augmented with additional circuitry to provide concurrent self testing and self-checking. This approach takes advantage of the fact that large memory chips externally organized as N one-bit words are internally organized as a square array of memory cells. There are typically SQRT(N) rows and SQRT(N) columns as shown in Figure 28. When a single bit is read from the memory chip, one whole row of SQRT(N) bits is read out of the memory array internal to the chip, but only one of these bits is selected to be output from the chip. For example, in a 16K memory there are 128 rows of 128 bits each, there are 128 sense amplifiers and an internal data register of 128 flip-flops. The high order half of the address bits select a row which is read out internally within the memory, and the other half of the address bits are used to select which bit of the row is to be read out (on READ) or modified for a STORE operation. For a store operation, the selected row is read out into an internal data register, the bit selected to be stored is set to the value being input for storage, and the whole row is written back to the memory array. This structure is essential in dynamic RAM, because it allows all bits of the chip to be read and restored in SQRT(N) read cycles.

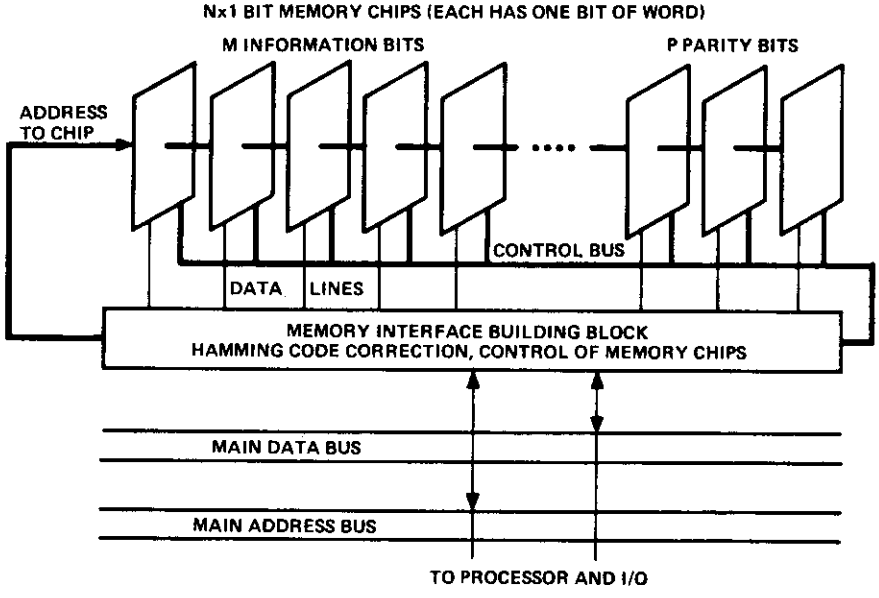


Figure 27: The Memory System

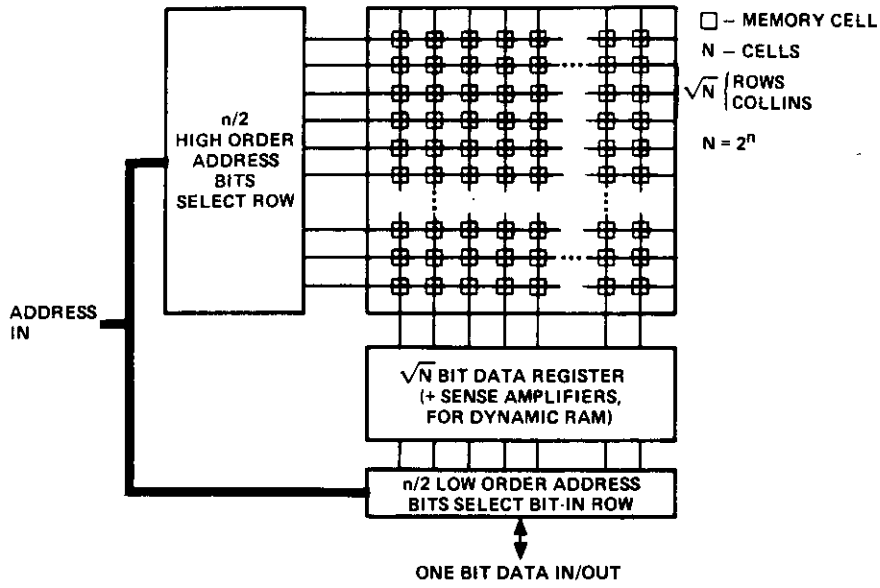


Figure 28: Memory Chip Organized as a Square Array

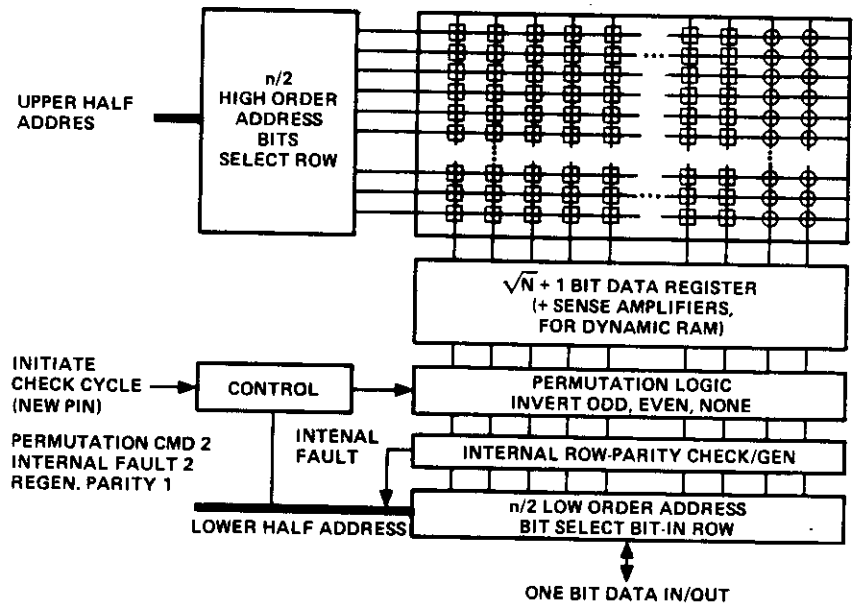


Figure 29: RAM Chip Including Checking and Permutation Logic

Modification to RAM Chips for Self-Exercising Features

The novel design features of the memory chips are described below:

(A) Two parity bits are added to each row of memory cells internal to the memory chips, one parity bit is used to check all odd numbered bits in the row, and the other parity bit checks all even bits in the row. This results in two additional columns in the memory array as is shown in Figure 29, and a two (odd and even- bit) parity checkers attached to the data register on the chip.

(B) The chip can be commanded to perform a CHECK CYCLE (typically taking less than a microsecond) during which a row of the memory array (specified by the most significant half of the address) is read out into the data register, the parity is checked, and the data bits are stored back into the row either unchanged or with one of three permutations:

i) Store Row without Change
ii) Store Row with Odd bit positions inverted
iii) Store Row with Even bit positions inverted
iv) Store Row with All bits inverted.

If a parity error is detected, an internal fault signal is sent to the MIBB, indicating that an error has occurred. During a check cycle, the memory can be commanded to Regenerate Parity (RP). In this case, both parity bits are re-computed over the selected row. (Note that the even and odd positions in a row, each typically consist of an even number of bits. Therefore when a group of (even or odd) bits are inverted, the associated parity bit remains the same).

(C) During normal (processor and I/O initiated) reads, a row (selected by the upper half of the address bits) is read into the data register, and one bit (selected by the lower half of the address bits) is read out. No internal parity checking is done, and this read can be viewed as identical to that done in conventional memory chips. [Note that the SEC/DED code over all chips will detect and correct faults on normal reads, thus internal checking is neither required nor used -- it would slow down the memory.] If DRO technology is used, the row is then stored back into the memory array unchanged.

(D) During normal write operations, the corresponding row is read out, the selected bit modified and the row re-stored back into the memory array. If the bit being stored is different from the bit originally contained in the memory, the associated (odd or even) parity bit is inverted. If the least significant bit of the address is zero, the even-bit parity is updated; if the LSB is one, the odd- bit parity is updated.

It is important that these special features be added to memory chips with minimum changes from their current design. A preliminary design indicates that the parity checks and inversion logic can be added to an existing memory chip with very little increase in area and in a way which does not disturb the layout (pitch) of the data register and memory array. The parity checks use a dual-rail logic design shown in Figure 30 which was used in a CMOS-SOS self- checking BIBB design previously published [SIEV 82]. The additional inverters necessary to permute data are only a slight modification of a typical data register design, adding pass transistors to cross-couple outputs.

It is also important not to significantly increase the number of pins needed on a memory chip. Therefore, the following technique of multiplexing pin functions is proposed which allows these special functions to be provided with only one additional connection pin. The additional pin conveys the command to execute a check cycle. Since the check cycle only needs to have a row in the storage array specified, the upper half of the address pins are used to specify that row address -- and the lower half of the address pins are available for other uses. Since most memory chips of this type have at least 12 address lines, this leaves six or more pins for use in the check cycle. The following use is suggested:

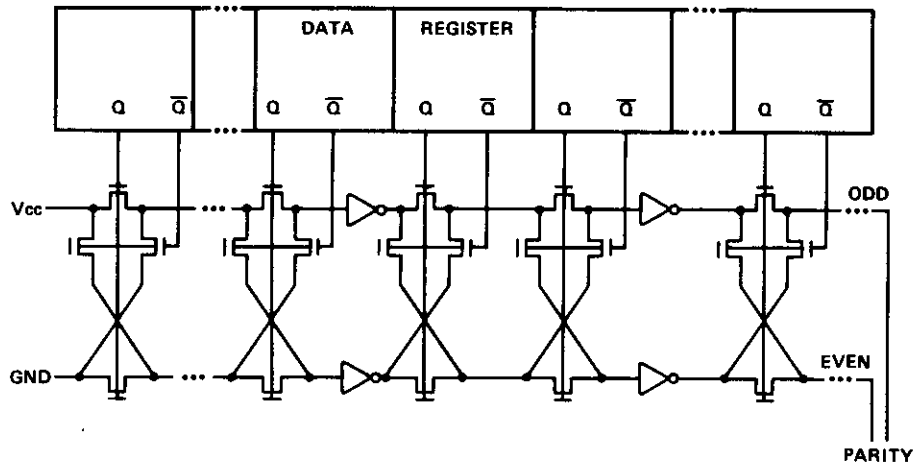


Figure 30: Parity Check Circuits (Simplified)

Low Order Address Bits During Check Cycle

1,2 - Inputs- Specify the permutation to use in storing back the selected row (00-store unchanged, 01-invert the odd bits in the row, 10-invert even bits, 11- invert all bits)

3,4 - Outputs- Are used as a morphic (self-checking) internal fault signal. If no parity error is detected these lines will output 1,0 or 0,1. If a parity error is detected, these lines will output either 1,1 or 0,0. (The two parity checks on odd and even bits of a row are generated morphically and combined with a morphic-and gate.[CART 72].)

5 - Input- Indicates that parity should be recomputed when the row is written back to storage. (This is done to initialize parity in each memory row. If an error occurs in a parity bit, this is the only way to correct it.)

3.2.2 DESCRIPTION OF MEMORY SYSTEM OPERATION

Memory system operation and checking is controlled by the Memory Interface Building Block (MIBB) as described below:

A. Sweep Testing to Detect Faults and Recover Transients

Periodically, after every K (K is a software programmable number of) memory cycles of regular program execution the MIBB signals the processor that the memory system is busy for several hundred nanoseconds while each memory chip is commanded by the MIBB to perform a CHECK CYCLE. A row number (generated by the MIBB) is presented to the chips on the high order address lines, and each memory chip reads out the specified row and performs a parity check to see if it still is correct. A fault is signaled to the MIBB by any chip in which an error is detected. During each check cycle a different row is read out and checked, allowing the whole memory array to be checked in SQRT(N) check cycles. We will designate a sequence of SQRT(N) check cycles which covers the whole memory (starting ar row 0 and ending at the last row) as a memory SWEEP. If scan cycles are performed every 10 microseconds, processing speeds will be slowed down by less than 10% and a memory sweep can typically be completed in 2.5 milliseconds (if 65K RAM chips are used). If the memory is dynamic RAM, check cycles can be carried out as part of the normal refresh cycle, only introducing delays of parity checking.

B. MIBB-Directed Correction of Transient Errors

If a parity error is detected in reading out a column within any chip, an internal error signal is sent to the MIBB controller. This indicates that one or more of SQRT(N) bits in a the row are in error. The High Order (row) Address bits sent to the memory chips (HOA) during the check cycle in which the error was indicated are saved by the MIBB. This is the address of the row in error. From the prospective of the memory system, this means that one or more words in a corresponding page of SQRT(N) words contains an error, but it does not indicate which one(s) contain the error. Therefore the MIBB goes into a second "information recovery mode" before continuing any additional check cycles. This is described below:

Upon receiving an internal error indication from a chip, the MIBB generates a second (special) RP check cycle to the same HOA to regenerate the parity bits, and insure that each parity bit is consistent with the data in the row whether the data is correct or incorrect. (The error could be in the parity bit itself, and this can only be corrected by regenerating the parity bits.) It then initiates a sequence of normal read and write operations to externally correct (using the Hamming Code) every word whose addresses lie in the faulty row. These read operations are requested every 10 to 20 processor cycles and interleaved with normal processing. A sequence of addresses with high order part equal to HOA, and low order part incremented from 000.00 to 111..11 are sent to the memory. At each address a read is performed and, if needed, a Hamming code correct and write cycle are performed. Thus, each word (corresponding to a single bit in the faulty row) is read out from all the memory chips, corrected externally using the Hamming code, and stored back into the memory chips. All single event upsets within a row (SQRT(N)) of a single chip should be corrected within a few milliseconds using this technique, and most multiple errors will be corrected as well (since they are likely to be single errors in different words.)

After this "information recover mode" is completed the MIBB sends an interrupt to the CPU. By interrogating a status register in the MIBB, the software can find out which memory chip signaled the fault and caused the information recovery cycle. The MIBB then continues its original sweep, initiating check cycles for other rows. Repeated information recovery cycles (fault indications) initiated by the same chip indicate a permanent fault, and the CPU may command its replacement with a spare, when bit-plane replacement is implemented in the MIBB [RENN 81b]. The information recovery mode requires SQRT(N) read operations and write operations for those bits in error. Therefore it typically takes slightly longer than a single memory sweep since only one error is expected in most cases. If a memory sweep takes S time when no rows are found with errors, it will require:

$X = S + NS$ when N different rows signal an error.

A typical sweep with a single row in error would result in the following events in a memory system using 16Kx1 Static RAM chips internally organized as a (128x128) array.

Time	Operation	Addresses Checked/Corrected	
10usec.	Check Cycle 0	0-127	No Error
20usec.	Check cycle 1	128-255	No Error
30 usec.	Check Cycle 2	256-383	Fault
40usec.	Special Check Cycle	256-383	Regenerate Parity
50usec	Read 256	Correct	Store 256
70usec	Read 257	Correct	Store 257
2610usec	Read 383	Correct	Store 383 *
2620usec.	Check Cycle 3	384-511	No Error
2630usec.	Check Cycle 4	512-639	No Error
3870usec	Check Cycle 127	16256-16383	No Error

(Correction of errors using dynamic RAM chips is slightly more complicated because it is necessary to continue read restore cycles while data correction is being carried out in order to assure that all data will be restored within a specified decay time. In this case check cycles are continued and interleaved with correction cycles. Since this straightforward, it will not be discussed here.)

C. Permuting Data for Permanent Fault Detection

As mentioned above, logic is provided in each memory chip allowing data read from a row during a check cycle to be stored back either unchanged or in one of three permuted forms: a) odd bits complemented, b) even bits complemented, and c) all (even and odd) bits complemented. We now show how these permutations can be used to provide extensive memory testing.

When the memory is initially loaded, the data is stored in non- permuted form. Initially, the MIBB carries out a memory sweep with special Regenerate Parity (RP) check cycles in order to initialize the parity bits in all rows in the memory chips. This is followed by memory sweeps executing regular check cycles as described above. During the check cycles of each sweep, the MIBB will specify either no change or a permutation for writing rows back to memory. During any single sweep, one permutation may be chosen for all even rows and one permutation (either the same or different) may be chosen for odd rows. One of the four options below is chosen for writing back even rows, and one (possibly different) is chosen for writing back odd rows.

(1) No Change, (2) Invert Odd Bits
(3) Invert Even Bits, (4) Invert All Bits

The following examples are instructive in showing how a series of permutations can be used to test for specific faults.

a) *Stuck at One/Stuck at Zero Cells* - If a cell is stuck at the same value as the data stored in it, there will not be an error until an attempt is made to store the other value (one or zero) that it is incapable of storing. To detect this type of fault, two sweeps are carried out which invert all bits in the memory. During the first sweep, data is checked in the true form while being written back in complement form. In the second sweep, complemented data is readout and checked while it is stored back in memory in true form. Thus each cell is required to take on both one and zero values and the on-chip parity check is used to verify that this took place.

b) *Coupling (e.g. shorts) Between Cells* - this is the situation where a cell is sensitive to the value in a neighboring cell. For example in a case of shorting, a cell might always take on zero when a specific neighbor contains zero, but work correctly when the neighbor contains one. With a series of permutations it is possible to make each cell take on both one and zero and to have all of its neighbors to take on the opposite values than they would have in a non-permuted form. In the example below, the underlined bit has eight neighbors in the memory array. We assume that it is in an even numbered row and an odd bit position within the row. The first sweep inverts all bits in odd rows and inverts even bits in even rows, leading to the second bit pattern in which all neighboring bits are inverted. The second sweep inverts only odd bits in even rows, causing the third pattern (all bits inverted). The third sweep repeats the permutation of the first sweep (inv. odd rows and even bits in even rows), resulting in the fourth pattern below in which the test bit is inverted and the neighbors restored to their original values.

Unpermuted Cells

/	e o e		e o e		e o e		e o e
o	1 0 0		0 1 1		0 1 1		1 0 0
e	0 0 1	→	1 0 0	→	1 1 0	→	0 1 1
o	1 1 0		0 0 1		0 0 1		1 1 0

A fourth sweep repeats the permutation of the second (inverting odd bits in even rows) to return to the unpermuted form. Thus, in four sweeps taking from 10-20 milliseconds, we have performed a form of disturb test on all bits of the memories which lie in even rows and have odd positions within the rows. All neighboring bits have taken on their opposite values, and the bits under test have been inverted while holding their neighbors the same. Thus each pair of (1) the bit under test, and (2) each of its neighbors have taken on all four possible values 1-1, 1-0, 0-1, and 0-0.

The test above only checks bits which lie on odd positions within even rows (1/4 of the memory array). It can be easily seen that three similar sequences of four sweeps exist which will also test all bits on:

(1) even rows, even bit positions
(2) odd rows, even bit positions
(3) odd rows, odd bit positions

Thus a sequence of no more than 16 sweeps will allow a this short test to be performed on all cells in the memory. This typically requires about 40 milliseconds.

D. The Relationship Between Error Checking and Testing Using Data Permutations

Checking for transient errors and single event upsets is unaffected by the way that data may be permuted in the memory. The parity checks are performed in exactly the same way, regardless of how the memory data is permuted. Since the data fields being checked are an even number of bits, the associated parity bits remain unchanged when (the odd or even) data bits are complemented within a row. When a parity error is found, the MIBB performs a series of read and store cycles to correct the words corresponding to the faulty row (as described above). This is done using regular read and write commands. When normal read and write operations are performed, the word read out will either be true or bitwise inverted (depending upon whether that cell position in all memories has been inverted or not). The MIBB must provide correction by inverting complemented words to obtain the correct read-data, and by inverting words words to be stored if the destination cells are expected to hold inverted data. This is explained below.

E. Correction of Permutations for Read/Write Operations

The MIBB maintains a list of permutations in a 4-bit-at-a-time shift register (see Figure 31), and each new permutation (B) is specified as four bits in the following form:

b1: invert odd bits in even rows
b2: invert even bits in even rows
b3: invert odd bits in odd rows
b4: invert even bits in odd rows

Each time that a new sweep is started the shift register is advanced to obtain new permutations for that sweep -- $b1$ and $b2$ are used to control write back when check cycles are performed on even rows. Similarly, $b3$ and $b4$ are used to control write back on odd rows.

The MIBB also maintains two four-bit scoreboards Old State (OS) and New State (NS). As a sweep progresses, the rows which have had check cycles have the new permutation, while those not checked yet have the old permutation. The scoreboards indicate which bits are inverted and which are not in the in the changed and not-yet changed rows of the memory chips. (A zero indicates not inverted and a one indicates inverted). The format of each scoreboard is shown below:

s1: odd bits of even rows are inverted
s2: even bits of even rows are inverted
s3: odd bits of odd rows are inverted
s4: even bits of odd rows are inverted

Each time a new sweep is started, a new permutation is to be generated, and the NS scoreboard is updated simply by taking:

$$S = S \text{ XOR } B.$$

The old value of NS is transferred to the Old State (OS)_ scoreboard.

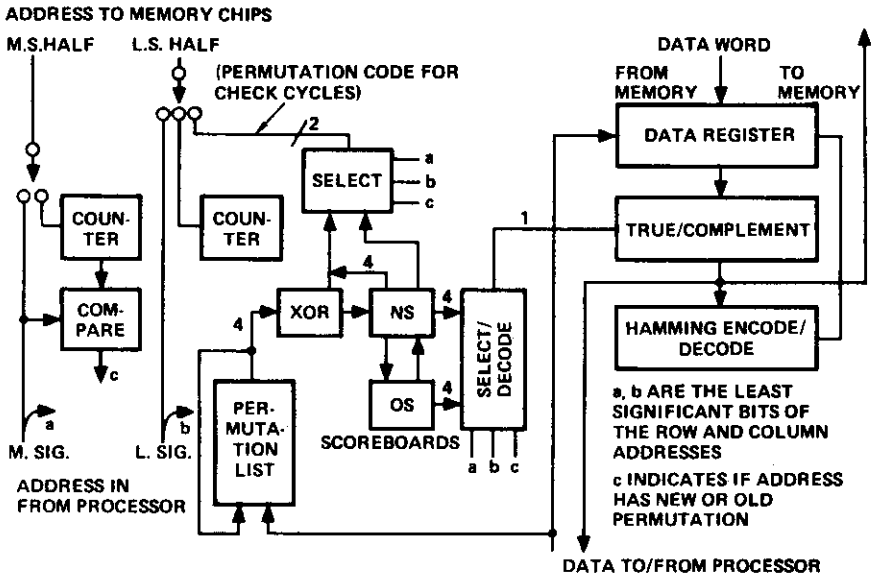


Figure 31: The Memory Interface Building Block

When the processor generates a read or write address, the MIBB checks to see if it should use the old state or new state in determining whether or not to complement data being read out of or stored into memory. The high order bits of the address are compared with the address of the last row for which a check cycle has been performed (compare block in Figure 31). If the address from the program is greater than the current addresses being modified by check cycles, then the permutation has not yet been updated. Therefore the OS scoreboard is used. If the address is lower, the NS scoreboard is used.

When a read or write is performed, the least significant bit of the address indicates whether an odd or even bit is being addressed within a row. The least significant bit of the most significant half of the address indicates whether an odd or even row has been selected. Therefore these two bits are used as an index to the appropriate scoreboard to determine whether or not the data word (being fetched or stored) is to be inverted.

3.2.3 SUMMARY OF SELF-EXERCISING MEMORY

This method is a simple but powerful way to provide memory systems which continuously test themselves during normal operation. This allows automatic recovery from very high rates of single event upsets, and exposes latent permanent faults very quickly that might otherwise upset recovery in fault-tolerant computers. This automatic self-testing feature can be used as built-in acceptance testing to determine if the memory system is free of faults. This quick testing is especially important in some applications where a set of computers may be exposed to massive transient disturbances (and some resulting permanent faults) and be expected to recover very rapidly. The first requirement after such an event is to quickly determine if hard failures have resulted in some of the machines. Thus each machine must exercise itself very quickly to determine if it can be used for subsequent computations. This rapid memory checking, and similar hardware-implemented processor testing (currently being studied) is necessary to meet requirements for quick recovery.

The cost is modest in increased circuit complexity, and speed penalties are also small. The above description demonstrates that this system can be built using well known design techniques and requires straightforward changes to existing chip designs.

3.2.4 EXTENSIONS OF SIMILAR TECHNIQUES TO PROCESSOR DESIGN

A typical processor consists of one or more sets of data paths and controllers. Our current focus is on generic datapath structures. The two major parts are a register array and an ALU. The register array can be expected to become much larger in future processor chips using RISC architectures or on-chip caches to better match on-chip and off-chip communication speeds [PATT 80]. This larger array will become more subject to single-event upsets and the resulting latent errors. Interleaved memory testing similar to that described above can be implemented in the processor as well.

For the combinational circuits (e.g. ALU) testing can be carried out by inserting test vectors into the datapaths and selected control levels. As in memory, the philosophy is to insert test vectors into the combinational logic interleaved with normal microcycles. One approach that can be taken is a modification of LSSD scan registers [EICH 77]. At the two inputs to the ALU, a serial-to-parallel shift register is installed. This register can be loaded serially from off-chip and does not affect the datapaths when being loaded (allowing normal computations to continue while it is being loaded). When loading is complete, a cycle is initiated which injects the test vectors onto the data path lines. These test cycles can be initiated by the microprogram and inserted, whenever possible, when the data paths under test are not being used (e.g. during an instruction fetch).

In order to provide for an efficient VLSI layout, the test vectors can be maintained in the microprogram ROM. When the microprogram commands a new test cycle, the next test word is read into a parallel-to-serial register and sent bit-serially to the data path chips via two extra pins (data and serial clock). After 30-40 clock (micro)cycles for loading the test is carried out during the following microcycle.

If the data paths are designed to be self-checking the mechanisms to verify the tests are already built-in. Our approach is to use parity codes for fault-detection in the registers, and to duplicate and compare the ALU. Thus, if the test exposes a fault in one ALU, it will disagree with the other one and a fault will be signaled.

The self-exercising self-checking processor is still under development. A self-checking self-exercising data path section of a processor has been completed, and current work is focusing on the microprogram control section.

4.0 METHODOLOGY OF IMPLEMENTING FAULT-TOLERANT SYSTEMS

When a complex system is designed, it is important that a methodology be used to assure that dependability is engineered into the system from its outset and to assure that the resulting design meets the specified requirements. Fault-tolerance cannot be retrofitted as an ad-on function and work properly.

4.1 THE DEVELOPMENT PROCESS

The development process is typically carried out in five stages as shown in Figure 32. These are:

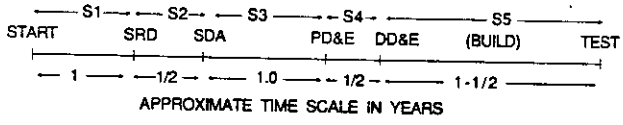
1. System Requirements Definition: fault tolerance requirements are identified, and preliminary planning is carried out for the next Stages.

2. Selection of Design Approach: general fault tolerance strategies are identified, alternate designs are evaluated, and an approach is selected.

3. Preliminary Design: initial design and evaluation.

4. Detailed Design: final design and evaluation; definition of prototype experiments.

5. Build and Test: experimental verification of fault tolerance.



SRD - SYSTEM REQUIREMENTS DEFINITION

SPECIFY: COMPUTATION MODEL AND REQUIREMENTS FOR PERFORMANCE AND FAULT-TOLERANCE

SDA - SELECT DESIGN APPROACH

PERFORM ARCHITECTURE TRADEOFFS (FROM SET OF ALTERNATIVES)
SELECT ARCHITECTURAL APPROACH AND FAULT-TOLERANCE STRATEGY

PD&E - PRELIMINARY DESIGN AND EVALUATION

SPECIFY PRELIMINARY HARDWARE AND SOFTWARE DESIGN
PROVIDE PERFORMANCE AND FAULT-TOLERANCE EVALUATION

DD&E - DETAILED DESIGN AND EVALUATION

PROVIDE A COMPLETED HARDWARE AND EXECUTIVE SOFTWARE DESIGN
PROVIDE REFINED ANALYSIS OF PERFORMANCE AND FAULT-TOLERANCE
PROVIDE A PLAN FOR A FEASIBILITY (BRASSBOARD) DEMONSTRATION

TEST - DEMONSTRATE AND EVALUATE

CONSTRUCT AND DEMONSTRATE BRASSBOARD COMPONENTS
DEMONSTRATE APPLICATIONS PROGRAMS
CONDUCT EXPERIMENTAL EVALUATION OF FAULT-TOLERANCE DURING OPERATION

Figure 32: The Development Process

Stage 1: System Requirements Definition

This first stage lays the groundwork for subsequent architectural tradeoff studies and the design and development of the computing system. The first step is to document the types of computations to be carried out and to derive dependability and fault-tolerance requirements. Objectives are to:

a) Identify computing functions required, study data flow, develop a computational model.

b) Identify fault-tolerance requirements of reliability, lifetime, recovery time, and critical state information.

c) Identify fault types, rates, and their dependence on technology and the environment.

d) Perform initial system partitions, identify alternative architectures, allocate functions to partitions, perform preliminary performance evaluation.

e) Specify initial allocation of reliability requirements to system partitions (to meet system goals).

Stage 2: Selection of Design Approach

The objective of this stage is to conduct tradeoffs between alternative architectures in order to select a design approach which best meets the requirements determined in Stage 1. Each alternative architecture should be partitioned into blocks with applications processing allocated to the partitions. Fault-tolerance mechanisms (detection, sparing, recovery) must be postulated and a reliable estimate made of their effectiveness. The design approached must then be modeled and compared. The selected approach should exhibit integrated fault-tolerance and performance concepts (as opposed to a collection of ad-hoc mechanisms) and the unity, simplicity, and completeness of a well-thought-out design concept.

The definition and evaluation of the selected architecture should include:

(1) a block diagram (partitioning) of the selected architecture,

(2) allocation of applications processing to the partitions,

(3) allocation of redundant sparing,

(4) error detection mechanisms used in each partition, including the various encodings for error detection and correction. Their location and placement and an estimate of their coverage,

(5) error containment strategies,

(6) a description of error and fault recovery mechanisms, (including switching out of faulty units, system reconfiguration and the restoration of error-damaged data) and estimates of their effectiveness (coverage)

(7) software executive strategy and method for preserving critical information during fault recovery.

(8) design features to support initial testing and periodic diagnosis and an estimate of their coverage.

Analysis and simulations should be provided to verify that the architecture can perform the required computations, and preliminary models should be developed for evaluating the reliability of the proposed design. The models should include estimated failure rates and coverages in the various partitions as parameters and should incorporate degraded performance as redundant equipment is exhausted. These preliminary modeling results should provide assurance that the architecture will meet the performance and fault-tolerance requirements identified in stage 1. An analysis should be provided which predicts the sensitivity of the approach to fault rates, coverage estimates, and the allocation of reliability values to modules.

Stage 3: Preliminary Design and Evaluation

The objective of this stage is to perform a detailed architecture definition and more accurately predict its performance and fault- tolerance. This should result in a detailed block diagram of both hardware and executive software components and an accurate behavioral specification and interface definition of each block. At this level of design, an accurate instruction-level (ISP) simulator should be produced and used to develop the system executive software and selected applications programs. The blocks should be defined to sufficiently fine granularity that their internal logic, fault modes, error manifestations, and detection techniques can be accurately predicted. A thorough analysis of all fault-tolerance mechanisms should be carried out. This design will be used to provide much more accurate simulations and estimations of coverage for reliability modeling than was possible at Stage 2.

At this stage it is extremely important to carefully verify the soundness of the design before proceeding to final design and implementation of VLSI and applications software. Therefore extensive analysis of faults and their resulting error patterns, and the actions of the error detection recovery mechanisms is essential to establish accurate parameters for reliability modeling. At this stage plans should be made for the detailed design, implementation, and verification which shall be conducted in the remaining development stages.

Stage 4: Detailed Design and Verification

The objective of this phase is to complete a detailed design of the computer system including design of custom chips, packaging, executive software and a set of applications programs. The fault- tolerance properties of the design must be carefully modeled and verified using accurately determined fault-tolerance parameters of the system. The reliability predictions at this stage (after breadboard experiments to validate them) must inspire a level of confidence necessary to justify construction and use of the computers in their intended application.

Completion of the detailed logic, executive software (including fault-management procedures) and a representative set of applications programs allows a very thorough analysis of the fault- tolerance of the design. The instruction-level behavioral simulator must now be extended to provide logic and circuit level simulations in all of the hardware modules in the system. The coverages and times associated with error detection and fault (and error) recovery mechanisms must be carefully determined by exhaustive circuit analysis and extensive fault simulations. A final reliability model must be produced based on the results of these analyses and simulations. Validation can only be carried out through a computer-aided analysis of the design. Final development and experimentation with a breadboard system can be used to partially verify this analysis, but the limited ability to test VLSI hardware makes it impossible to obtain sufficiently accurate results to validate its operation by experimentation alone.

Stage 5: Build and Test

The objective of this stage is to build and test a feasibility model to experimentally verify its performance and fault-tolerance. The brassboard must be capable of running applications programs and recovering satisfactorily from experimentally inserted faults. A comprehensive set of experimental tests must be conducted to validate the computer-aided fault simulation results of Stage 4 by showing that the simulated fault-detection and recovery procedures accurately reflect what happens in the brassboard. (If not, either errors must be corrected in the brassboard design or the simulation models must be corrected.) When these modeling and analytic results are reconciled, the reliability models based on these results can be used with an increased degree of confidence in their prediction of the dependability properties of the design.

4.2 MODELING AND VALIDATION

Modeling and verification must be carried out at each stage of the development process. A model of the design is created as shown in Figure 33, and is used to predict its reliability. The computer system being designed employs a number of fault-tolerance mechanisms for error detection and fault recovery. These mechanisms are reflected in the model both structurally as states and as parameters which indicate coverages, latencies, and recovery times that determine transitions between states in the model. In the early stages of system design (e.g. stage 2 and 3 above) it is only possible to approximate the structures and parameters used in the model because the design is not yet complete. The designer must estimate on the basis of experience what the parameters (e.g. failure rates, recovery times, coverages) will be. In the early stages, modeling is essential to make tradeoffs between alternative approaches and to determine the performance bounds that must be met in the final design if satisfactory dependability is to be achieved.

As the design process proceeds, the design is increasingly refined, and the model must be correspondingly refined also. Finally, at stage 4, the design is complete, parameters can be carefully determined, and the model is then used for system validation. It is used to predict the ultimate reliability that will be achieved when the system is placed into operation.

Validation of a system involves determining existing fault-tolerance properties of the system (in the form of parameters derived from measurement or analysis) and using mathematical models to predict its future performance after it is deployed. The design is evaluated by 1) enumerating the fault-set, 2) describing how the design deals with each fault-type, and 3) characterizing the fault handling procedures with estimates of coverage, delay times and system-wide effects of faults and errors. Analytic models (e.g. Markov, simulations, timed petri-nets, and combinational models) use the system structure, estimates of failure rates, and estimated coverages and delay times to estimate the system's reliability.

Validation involves two distinct types of expertise, system architecture and mathematical modeling. There are two fundamental issues that must be addressed: i) the fidelity of the model, and ii) the ability to accurately determine the parameters it uses as inputs. Both need substantial improvement beyond the current state- of-the-art to validate the next generation of multicomputers.

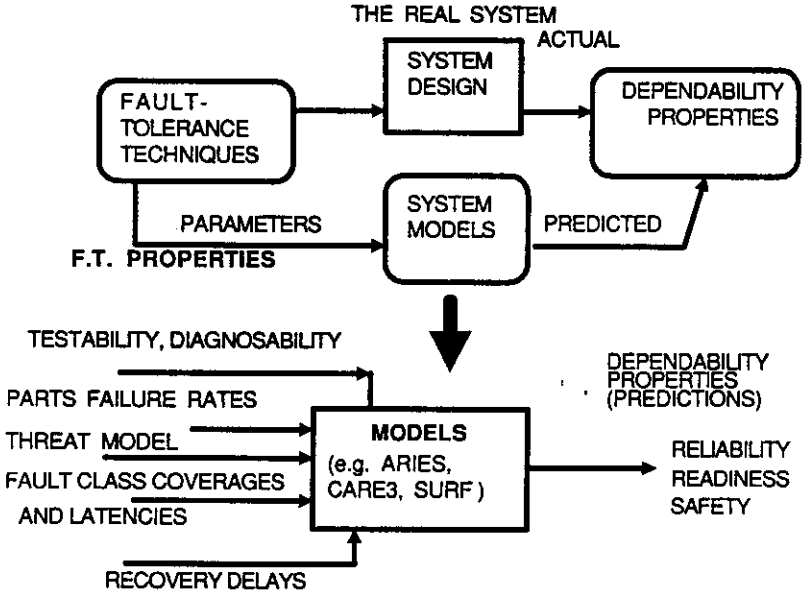


Figure 33: The Modeling and Validation Process

Mathematical Reliability Prediction Models

A number of computer-aided mathematical modeling systems exist which use Markov, Petri-Net, and combinational models to predict performance, and reliability. Some of the best ones are ARIES, SURF, and CARE3. Monte Carlo and other simulation procedures are also applicable. These models have been used to verify the reliability of existing fault tolerant computing systems. They are applicable to the verification of current computing subsystems, but new extensions are needed to existing models to allow them to handle the much greater levels of complexity expected.

A typical reliability model consists of:

(1) the states that the system can be in - A state consists of a unique combination of working and failed hardware and software modules, and the operational mode that the system is in (normal operation or one of several recovery modes).

(2) the probability of making a transition from each state to another state as a function of time - This is based on transition rates derived from fault probabilities and "coverages" determined from the design. The transition from one state to another is initiated by a change of operational mode, completion of a recovery procedure (if the state was a recovery state) or the occurrence of a fault (which can occur either during normal operation or during a recovery state.) There is a coverage parameter (c) associated with each possible transition which indicates the probability that the fault-tolerance mechanisms work as expected, and expected transition rates are multiplied by c. With probability "1-c" the fault may not handled properly and the transition goes to a different state associated with failure of the recovery mechanism. These models can then be simulated (or in some cases solved analytically) to estimate reliability.

There is a need to develop better reliability models because existing models do not adequately predict the future behavior of complex distributed systems. For example, in highly parallel multicomputers redundancy can be employed most effectively at several levels, spare chips in memory, spare memory modules within processors, and spare computers. This results in a large number of dependent failure modes. Existing Markov models do not deal well with systems in which there are a large number of dependent failure modes because the number of states becomes very large, nor do they deal adequately with large systems whose failure rates are time- varying. Simulations are too expensive and cannot be adequately generalized. Thus with the current state of modeling it is necessary to make simplifying assumptions to make the models tractable which compromise the fidelity of the results.

Making advances in modeling is a hard problem whose future is unpredictable because both complex mathematics and new ideas are required.

Matching Models to Reality

In order to use a reliability prediction model, it is necessary to abstract many properties of the system being modeled. A model uses, after all, a gross simplification of the actual system to predict its future. The parameters that it uses are typically single-number averages of complex physical phenomena that may occur individually in millions of different ways. The modeling results are very sensitive to these parameters such as the assumptions of 1) what faults will occur, 2) what their rates of occurrence are, 3) latencies and coverages for fault detection in each class, 4) latencies and coverages for recovery.

There is a tendency for designers to use the inherent complexity and sophistication of mathematical models to justify their results. Often this obscures the fact that: i) fault modes are not accounted for, ii) the model omits important system states, or iii) uses sensitive input parameters which cannot or have not been adequately measured. One can easily see from experience that a simple model applied correctly will often give more honest results than a state- of-the-art model improperly applied. But the casual observer will often choose the more complicated model because of its apparent sophistication.

The only way to properly validate these machines is to develop a stringent discipline in documenting the requirements, assumed fault set, assumptions made in the design, and the details of the design itself. As a computer is developed it proceeds through stages of successive refinement (PMS, ISP, Logic, Electronic), and modeling must be applied at all levels to determine that the design decisions are leading to a correct result. As the design proceeds and is refined, the parameters for modeling can be determined more accurately, and the modeling predictions become more accurate as the model is also refined. It is essential that the system description, and models be precisely specified at each stage of design so that independent evaluation (e.g. by a sponsor) is possible.

There is a need to integrate evaluation/validation tools with CAE/CAD tools, used by designers. These augmented CAE/CAD tools are essential to obtain accurate simulation and analysis of a system design to measure realistic fault-tolerance parameters (fault- coverages, latency) and perform experiments to validate assumptions employed in performability models.

Especially needed are CAE/CAD tools which provide multi-level description of a system and multi-level fault simulations. Commercial design tools only provide fault-simulation at the lowest level. Extensive fault simulation is needed to obtain accurate modeling parameters - several orders of magnitude more processing than is used for test generation in non fault-tolerant systems. This can only be done with a finite amount of computing power by combining low-level modeling of a small portion of a system and less computation intensive behavioral modeling of the rest of the system to observe higher recovery functions. Research is needed to define the common design databases needed to integrate multi-level design tools, verify consistency of specifications at different levels, and provide multi-level fault simulations.

References

[AIPS 84] Advanced Information Processing System (AIPS) System Specification, prepared for Johnson Space Center by the C. S. Draper Laboratory, Cambridge Mass., May 15, 1984. (Distribution Limited)

[ANDE 67] Anderson, J., and Macri, F., "Multiple Redundancy Applications in a Computer," *Proc. 1967 Ann. Symp. Reliability,* Washington D.C., pp. 553-562, Jan 1967.

[ANDE 81] Anderson T, and Lee, P., Fault Tolerant Principles and Practice, Prentice Hall International, 1981.

[AREN 83] Arens, W., and Rennels, D., "A Fault-Tolerant Computer for Autonomous Spacecraft," *Proc. 13th Int. Symp. Fault-Tolerant Computing,* Milan, Italy, June 1983, pp. 467-470.

[AVIZ 71a] Avižienis, A., et. al., "The STAR (Self-Testing-And Repairing Computer: An Investigation into the Theory and Practice of Fault-Tolerant Computing", *IEEE Trans. Computers,* Vol. C20, No. 11, pp. 1312-1321, Nov. 1971.

[AVIZ 71b] Avižienis, A., "Arithmetic error codes: Cost and effectiveness studies for application in digitala system design," *IEEE Trans. Computers,* vol. C-20, pp. 1322-1331, Nov. 1971.

[AVIZ 84] Avižienis A., and Kelly, J., "Fault Tolerance By Design Diversity: Concepts and Experiments," *Computer,* August 1984.

[BARL 82] Barlett, J., "A NonStop Operating System," in *Computer Structures Principles and Examples,* ed. Siewiorek, D. et. al., McGraw Hill, 1982, pp. 480-485.

[BOUR 69] Bouricius, W., et. al., "Reliability Modeling Techniques for Self Repairing Computer Systems," *Proc. 24th National Conf. of the ACM,* 1969.

[BOUR 71] Bouricious, W., et.al., "Reliability Modeling for Fault Tolerant Computers," *IEEE Trans. Computers,* Nov. 1971, pp. 1306-1311.

[BOZO 80] Bozorgui-Nesbat, S., and McCluskey E., "Structured Design for Testability to Eliminate Test Pattern Generation", *Dig. 10th Int. Symp. Fault Tolerant Computing, IEEE Computer Society,"* 1980, pp. 158-163.

[BREU 76] Breuer, M., and Friedman, A., Diagnosis and Design of Reliable Digital Systems, Computer Science Press, Potomic, Maryland 1976.

[BURC 76] D. Burchby et al., "Specification of the fault-tolerant spaceborne computer," in *Proc. 1976 Int. Symp. Fault-Tolerant Computing* (Pittsburgh, PA), pp. 129-133, June 1976.

[CART 72] Carter W., et. al., "Computer Error Control By Testable Morphic Boolean Functions- a way of Removing Hardcore," *Dig. Second Int. Symp. Fault-Tolerant Computing,* Newton Mass., June 1972, pp. 154-159.

[CART 77] Carter, W., et. al., "Cost Effectiveness of Self-Checking Computer Design," *Dig. 7th Int. Symp. Fault-Tolerant Computing,* Los Angeles, pp. 117-123, June 1977.

[CHRI 83] Christian, F., "A Rigorous Approach to Fault Tolerant System Development", *IBM Research Report RJ 4008 (45056),* Sept. 1983.

[EICH 77] Eichelberger, E., and Williams, T., "A Logic Design Structure for LSI Testability," *Proc. 14th Design Automation Conf.,* June 1977, pp. 462-468.

[FREI 82] Freiburghouse, R., "Making Processing Fail Safe," Mini-Micro Systems, May 1982, pp. 255-264.

[GILL 72] Gilley, G. C., "A fault-tolerant spacecraft," in *Dig. 1972 Int. Symp. Fault-Tolerant Computing* (Newton, MA), pp. 105-109, June 1972.

[GREY 84] Grey, B., et. al., "A Fault Tolerant Architecture for Network Storage Systems," *Proc. 14th Int. Symp. Fault-Tolerant Computing,* Orlando, Fla., June 1984.

[GUNN 83] Gunningberg, P., "Voting and Redundancy Management Implemented by Protocols in Distributed Systems," *Dig. Int. Symp. Fault Tolerant Computing,* Milan, June 1983, pp. 182-185.

[HAYE 76] Hayes, J., "A Graph Model for Fault-Tolerant Computing Systems," *IEEE Trans. Computers*, Vol. C-25, No. 9, September 1976, pp. 875-884.

[HOPK 78] Hopkins, A., "FTMP - A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft," *Proc. IEEE*, Vol. 66, No. 10, pp. 1221-1239, October 1978.

[IHAR 78] Ihara, H., et. al., "Fault-Tolerant Computer System with Three Symmetric Computers," *Proc. IEEE*, Vol. 66, No. 10, pp. 1160-1177, October 1978.

[INTE 81] "The Intel 432 System Summary," Intel Corp., Aloha, Oregon, 1981

[JPL 85] "Hypercube Research Project Mark III Core Engineering Notebook," *Report # JPL D-2431*, Jet Propulsion Laboratory, Pasadena, CA, June 3, 1985.

[KATS 82] Katzman J., "The Tandem 16: A Fault-Tolerant Computing System," in *Computer Structures Principles and Examples*, ed. Siewiorek, D. et. al., McGraw Hill, 1982, pp. 470-479.

[KUEN 69] Kuehn, R.J., "Computer Redundancy: Design, Performance, and Future", *IEEE Trans. on Reliability*, Vol. R18, No. 1, Feb 1969, pp. 3-11.

[KUHL 80] Kuhl, J., and Reddy, S., "Distributed Fault Tolerance for Large Multiprocessor Systems," in *Proc. Seventh Annual Symp. on Computer Architecture*, pp. 23-30, May, 1980.

[LAMP 81] Lampson, B., "Automic Transactions," in *Distributed Systems Architecture and Implementation*, An Advanced Course, Springer Verlag, 1981.

[LESH 76] Lesh, H. F., and P. Lecoq, "Software techniques for a distributed real-time processing system," in *Proc. IEEE National Aerospace Electronics Conf.* (Dayton, OH), pp. 290-295, May 1976.

[MEYE 81] Meyer, J., "Closed Form Solutions of Performability," Dig. Eleventh Int. Symp. Fault Tolerant Computing, Portland, Maine, June 1982, pp. 66-71.

[MIL 217D] Military Handbook, "Reliability Prediction of Electronic Equipment," MIL-HDBK-217D, DoD, Washington, D.C. January 1982.

[MILSTD] U.S. Military Standard MIL-M-38510, Class S Parts.

[NGYW 80] Ng, Y, and Avizienis A., "A Unified Reliability Model for Fault-Tolerant Computers," *IEEE Trans. Computers*, Vol. C29, No. 11, Nov. 1980, pp. 1002-1011.

[OBLO 62] Oblonsky, J., "A Self Correcting Computer" in *Digital Information Processors*, W. Hoffman, ed., Interscience Publishers, New York, 1962, pp. 533-542.

[POPE 81] Popek, J., et. al., "LOCUS, A Network Transparent, High Reliability Distributed System," *Proc. 8th SOSP*, Monterrey CA., Dec. 15-17, 1981.

[PREP 67] Preparata, F., Metze, G., and Chien, R., "On the Connection Assignment Problem of Diagnosable Systems," *IEEE Trans. Computers*, Vol. EC-16, No. 6., December, 1967, pp. 848-854.

[RAGH 82] Ragavendra, C., et. al., "Reliability Optimization in the Design of Distributed Systems," *Proc. Third Int. Conf. on Distributed Computing Systems*, Miami, Fla. October, 1982.

[RENN 73a] Rennels, D. A., and Avižienis, A., "RMS: A reliability modeling system for self-repairing computers," in *Dig. 1973 Int. Symp. Fault-Tolerant Computing* (Palo Alto, CA), pp. 131-135, June 1973.

[RENN 73b] Rennels, D. A., "Fault detection and recovery in redundant computer using standby spares," Tech. Rep. UCLA-ENG-7355, Univ. California, Los Angeles, June 1973.

[RENN 76] Rennels D. A., et. al., "The Unified Data System: A Distributed Processing Network for Control and Data Handling on a Spacecraft," *Proc. NAECON*, Dayton, Ohio, pp. 283-289, May 1976.

[RENN 78a] Rennels, D. A., Avižienis, A., and M. Ercegovac, "A study of standard building blocks for the design of fault-tolerant distributed computer systems," in *Proc. 1978 Int. Symp. Fault-Tolerant Computing* (Toulouse, France), June 1978.

[RENN 78b] Rennels D. A., "Architectures for Fault-Tolerant Spacecraft Computers," *Proc. IEEE*, Vol. 66, No. 10, pp. 1255-1268, October 1978.

[RENN 81a] Rennels D. A. et. al., *Fault Tolerant Design Considerations for Future Spacecraft Systems,* UCLA Computer Science Department Report, prepared for the Aerospace Corp., Aerospace Library Call No. A81-04858, October 1981.

[RENN 81b] Rennels, D. A., Avižienis, A., and Ercegovac, M., *Fault-Tolerant Computer Study Final Report,* JPL Publication 80-73, Jet Propulsion Laboratory, Pasadena, California, February 1981.

[RENN 82] Rennels, D. A. et.al., "Fault Tolerant Design Considerations for Future Spacecraft Computer Systems," UCLA Report Prepared for the Aerospace Corporation, El Segundo, CA, Aerospace Library Call Number A81-04858, October 2, 1981.

[RENN 84] Rennels, D. A., "Fault-Tolerant Computing: Concepts and Examples," *IEEE Trans. Computers,* December 1984, pp. 1116-1129.

[RENN 86a] Rennels, D. A., "On Implementing Fault-Tolerance in Binary Hypercubes," *Dig. Int. Symp. Fault-Tolerant Computing,* Vienna, June 1986, pp. 344-349.

[RENN 86b] Rennels, D. A., and Chau, S., "A Self-Exercising Self-Checking Memory Design," *Dig. Int. Symp. Fault-Tolerant Computing,* Vienna, June 1986, pp. 358-363

[ROHR 73] Rohr, J., "STAREX Self-Repair Routines: Software Recovery in the JPL-STAR Computer," *Dig. Second Int. Symp. Fault-Tolerant Computing,* Palo Alto, CA., June 1973.

[ROTH 67] Roth J., et. al., "Programmed Algorithms to Compute Tests to Detect and Distinguish between Failures in Logic Circuits," *IEEE Trans. Elec. Computers,* EC-16, October 1967, pp. 567-580.

[RUSS 75] Russell, J., and Kime, C., "System Fault Diagnosis: Closure and Diagnosability with Repair," *IEEE Trans. Computers,* Vol C-20, No. 11, November 1975, pp. 1078-1088.

[SEDM 80] Sedmack, R., and Liebergot, H., "Fault Tolerance of a General Purpose Computer Implemented by Very Large Scale Integration," *IEEE Trans. Computers,* Vol. C-20, No. 6, June 1980, pp. 492-500.

[SIEV 82] Sievers, M. and Rennels D. A., "An LSI Totally Self-Checking Memory Interface," *Proc. IEEE International Symposium on Circuits and Systems,* Rome, Italy, May, 1982, pp. 1176-1179.

[SIEW 82] Siewiorek D., and Swarz R., *The Theory and Practice of Reliable System Design,* Digital Press, 1982.

[SKLA 76] Sklaroff, J., "Redundancy Management Technique for Space Shuttle Computers," *IBM J. Res. Dev.,* Vol. 20, no. 1, pp. 20-28, Jan. 1976.

[TOYW 78] Toy, W., "Fault-Tolerant Design of Local ESS Processors," *Proc. IEEE,* Vol. 66, No. 10, pp. 1126-1145, October 1978.

[WAKE 78] Wakerly, J., *Error Detecting Codes, Self-Checking Circuits and Applications,* New York: North Holland, 1978.

[WENS 78] Wensley, J., "SIFT: The Design and Analysis of a Fault-Tolerant Computer for Aircraft Control," *Proc. IEEE,* Vol. 66, No. 10, pp. 1240-1255, October 1978.

[WILL 79] Williams, T., and Parker, K., "Testing Logic Networks and Design for Testability," *Computer,* Vol. 12, No. 10, October 1979, pp. 9-22.