An overview of fault-tolerant digital system architecture

by STEPHEN Y. H. SU and RICHARD J. SPILLMAN
Utah State University
Logan, Utah

ABSTRACT

With the increasing use of computing systems in such crucial areas as medicine and space, there has come a great need for computers that remain operational in spite of hardware failures. This paper provides a brief overview of several approaches to fault-tolerant computing. Five hardware redundancy techniques are reviewed: static, dynamic, hybrid, self-purging and the reconfiguration scheme. In addition, the advantages and disadvantages of error correcting codes and software fault-tolerant systems are outlined as well as bi-duplexed systems, alternating logic, fail-soft and shared logic systems. It is suggested that perhaps the best fault-tolerant system employ a combination of hardware redundant techniques and software protection.

INTRODUCTION

Since the early 1940's when the first relay computers were developed, the question of how to insure reliable computer operation has been an important one. Today, when computers are used in critical space missions, millions of miles from their human operators, and in biomedical systems where a human life depends on their correct operation, even a small computing error could result in the loss of a life or millions of dollars of equipment and years of research. Under such conditions, the design of computing systems which can operate correctly in spite of hardware or software failures is important. Such systems are called fault-tolerant systems. Specifically, fault-tolerant computing has been defined as the ability to execute specified algorithms correctly regardless of hardware and/or software failures.[1]

The first step towards a fault-tolerant system is to build as much fault-tolerance into the system as possible.[2] Fault-intolerance is the procedure whereby the reliability of the system is increased by avoiding the causes of system failures. This is achieved before the final system is constructed, in the design phase. Only the most reliable components are selected, software is completely tested before it is released, and fault detection and ease of repair considerations are introduced at the beginning of the design process and considered at every succeeding step. Fault-intolerance, however, can only postpone the occurrence of faults, it can not eliminate them entirely. Hence, the second step towards a fault-tolerant system requires protecting the system from faults.

There are several approaches to fault protection including hardware redundancy where extra components are introduced into the system, error correcting codes such as parity checkers, or software fault-tolerant systems where special software procedures are used to recover from an error. In this paper, all three approaches will be briefly reviewed and their advantages and disadvantages outlined. Specifically, sections of this paper will examine five redundancy techniques: masking, standby, hybrid, self-purging, and reconfiguration; will review error correcting codes; software fault-tolerant procedures will be outlined; a number of new approaches to fault-tolerance will be reviewed; and will draw some conclusions and offer some suggestions for future research.

REDUNDANCY TECHNIQUES

The effects of hardware errors can be overcome through the use of protective redundancy.[3] Hardware protective redundancy is defined as the use of additional components which allow the system to continue to operate correctly in the presence of hardware faults. The cost of the extra components was, at one time, a strong argument against the use of redundancy. However, since the advent of LSI and MSI and the reducing cost of digital hardware, redundancy has become an important means of implementing fault-tolerant systems. There are three classifications for the conventional redundancy techniques: static, dynamic, and hybrid. In addition to these three, two other approaches are discussed separately in this paper, self-purging redundancy and reconfiguration scheme redundancy which will introduce different ideas from the previous three.

Static redundancy

Static redundancy involves the use of extra components such that "the effect of a faulty circuit, component, subsys-

tem, signal, or program is masked instantaneously by permanently connected and concurrently operating circuits."[4] It is also called masking or massive redundancy.

The simplest static redundancy technique is triple modular redundancy (TMR). This technique was first studied by John von Neumann.[5] It is implemented using three identical modules operating in parallel as shown in Figure 1. The output of each module passes through a majority voting system whose output agrees with the majority of module outputs. TMR could be expanded to include any odd number of redundant modules to produce an NMR (N Modular Redundancy) system, where N is an odd number.

An NMR can tolerate (N−1)/2 module failures. The three major advantages of static redundancy are:

(1) The corrective action is immediate, the faulty module never effects the circuit.
(2) There is no need for fault detection procedures.
(3) The conversion of a non-redundant system to a static redundant one is easily undertaken. Simply construct (for a TMR system) two new copies of each non-redundant module.

It is important to remember the primary assumption behind the use of static redundancy. That is, a failure in one module is independent of the other modules. This assumption is not valid within an LSI or MSI package and hence, static redundancy is ineffective at the logic gate level within LSI or MSI.

Dynamic redundancy

Dynamic redundancy involves only one unit operating at a time with several spares waiting to replace the unit if a fault is detected. Obviously, this system requires both a method of switching the new module into the circuit to replace the faulty unit and a method of detecting the fault in the circuit. There are two possible switching procedures, logic switching and power switching. In logic switching, all the spares are powered up and operating, when a fault is detected the output of the next spare in line is switched into the circuit and the faulty modules output is switched out. This is equivalent to enabling or disabling a gate between the module's output and the system output. Power switching requires that only the operating unit be powered up. When a fault is detected, the faulty units power is switched off and the next spare in line is switched on. The power switching procedure isolates the spares from the circuit and does not waste power on spares which are not performing useful operations.

The fault detection procedure is more complicated and could include either concurrent or periodic techniques.[6] Concurrent fault detection is a continuous fault detection technique which utilizes coding and redundant signals. Periodic fault detection requires testing the circuit at certain points in time with a special diagnostic routine. This involves stopping the normal operation of the circuit to conduct the tests, however.

If a working fault detection system can be constructed, there are several advantages to a dynamic redundant system:

(1) All the spares can be used and the system can tolerate as many faults as there are spares.
(2) The number of spares is easily adjusted to allow for later increases in reliability if needed.
(3) If power switching is used, the spares are isolated from the system so the independent fault assumption applies.

Hybrid redundancy

Hybrid redundancy combines the static and dynamic redundancy approaches. Static redundancy, used to provide fault detection, is combined with a set of spares to replace any faulty module in the static system. The basic hybrid redundancy system is the TMR + spares system (Hybrid (3,S) system) shown in Figure 2.[10]

In the Hybrid (3,S) system, if a fault occurs (the output of one of the gates does not agree with the output of the other two) the faulty gate is switched out and one of the spares is

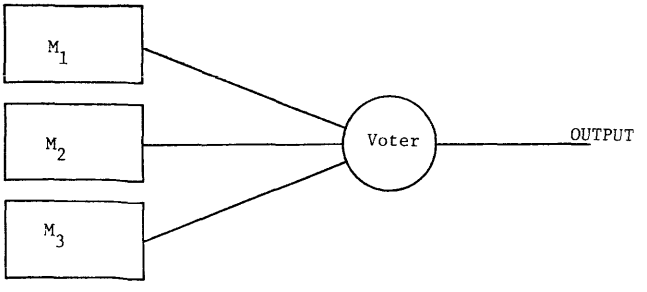

Figure 1—Triple modular redundancy (TMR)

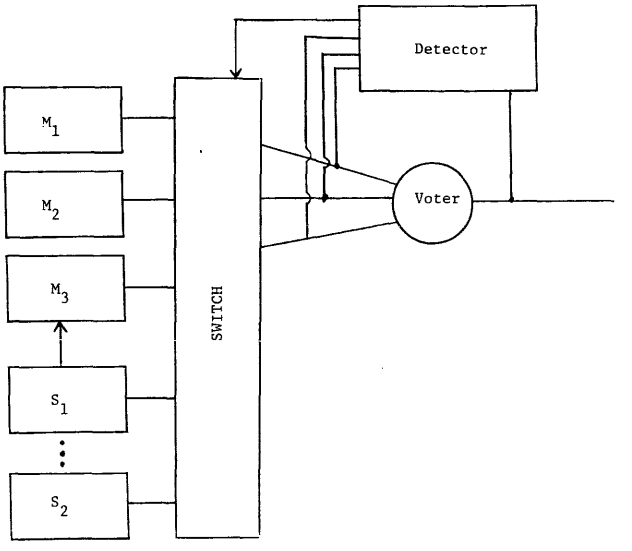

Figure 2—Hybrid (3,N) system

switched in. The system is then reduced to a Hybrid (3,S−1). If all the spares are used up, the system reduces to a Hybrid (3,0) or just a standard TMR.

The hybrid system possess all the advantages of dynamic redundancy without the problem of constructing complicated fault detection procedures. In addition, the voting circuit of a hybrid system masks the effect of any fault. However, the switching system in hybrid redundancy is more complex than either static or dynamic redundancy switching systems. Hence, hybrid redundancy is prone to switching system failure which could bring the entire system down.

Sieworek and McCluskey[8] have suggested the use of an iterative cell switch. They found that an iterative cell switch could save 25 percent and in some cases 80 percent of switch complexity over standard hybrid switching systems. Ogus[26] has also studied fault-tolerant design of iterative cell switches. Use of the iterative cell switch, then, would lead to further improvements in hybrid redundancy reliability.

As with the TMR, a hybrid system with a TMR core can not handle multiple faults.[8] For example, if two active modules in a Hybrid (3,S) are faulty, then the voting gate will incorrectly switch the one fault-free gate out and switch a spare in. The voting circuit will then determine that the spare is in the minority and replaces it with another spare. This process will continue until all the spares are used up and the system crashes. Ramamoorthy and Han[9] have suggested a modification of the hybrid system in which each module's output passes through its own error detector circuit as well as the voting circuit. Called a detector redundant system, this circuit could detect an error and determine which modules are in error. Only the faulty modules would be switched out, hence, this circuit would not be subject to the catastrophic consequences of multiple failures that plague standard hybrid redundancy. However, such error detection circuitry would increase the complexity of the control system.

If just single faults are assumed, the Hybrid (3,S) system performs very well. A Hybrid (3,S) with N units (N−3 spares) can tolerate N-2 single faults using power switching whereas a NMR (a static system with N units) can tolerate only (N-1)/2 module failures. A dynamic system with N units can tolerate (N-1) failures, one more than a Hybrid (3,N-3). However, the dynamic system requires a complicated fault detection procedure, which in general lowers its net reliability. In addition, dynamic faults are not masked. So the gain of tolerating one additional module failure is outweighed in most applications by the loss of flexibility in the dynamic system.

Mathur and Avizienis[10] found that in adding another module to a Hybrid (n,S), a hybrid system with a nMR core and S spares (where N=n+S), the largest gain in reliability is made if the new module is added to the set of spares. That is, the reliability of a Hybrid (n,S+2) is greater than the reliability of a Hybrid (n+2,S). Therefore, in a hybrid system, most of the redundancy should be in the spares and n should equal 3.

The optimum value of S in a Hybrid (n,S) can also be calculated. Cochi[11] assumed that the reliability of the switching system used to switch the faulty module out and the spare in is a function of n+S. He found a method for calculating an optimum value for S, the number of spares. It depends upon the reliability of both the switching system and the individual units.

Self-purging redundancy

Self-purging redundancy[7,12] is sometimes considered a form of hybrid redundancy. However, its approach is different enough from hybrid systems to offer some real advantages in certain applications. In self-purging redundancy, all of the modules are initially connected to a threshold voter through a switching circuit as shown in Figure 3. If an error occurs in a module, its switch is turned off, forcing a logic 0 on its output. In effect, the faulty module is removed from the voting.

The major advantage of the self-purging system over the standard hybrid approach is the simplicity of the switching mechanisms. The self-purging switching circuit only has to turn a faulty module off, whereas the hybrid switching system has to turn off the faulty module, locate an unused spare and then turn the spare on. Because of the simplicity of the self-purging switch, it is possible to include the switch as part of the module forming a modified module.[12] In addition, because of the simplicity of the switching mechanism, it is easy to duplicate the switches such that each module's output passes through two switches. The number of inputs to the voter is doubled. The reliability of the switching system, which is the weak point of any redundancy scheme, is thereby increased through this use of redundant switching. Such redundant switching is only possible because of the simplicity of the self-purging switching system.[12]

In systems where multiple failures are likely, most redundancy schemes break down. For example, consider a self-purging system with 5 modules and the threshold of the voter is 3. The output of this system is given by $Z=1_1(1_2(1_3+1_4+1_5)+1_31_4+1_41_5)+1_2(1_31_4+1_31_5+1_41_5)+1_31_41_5$. If two modules fail, say 1_1 and 1_2 then $1_1=1_2=0$ and $Z=1_31_41_5$. So, the majority gate becomes an AND gate. Hence, the system can no longer tolerate any stuck-at-0 fault. If the threshold is 2, then the self-purging redundancy system cannot tolerate a double stuck-at-1 fault.[14] How-

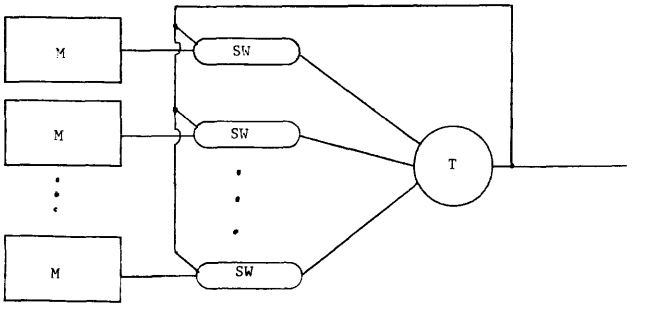

Figure 3—A self-purging system

ever, Losq[13] has developed self-purging systems which have a high probability of recovering from multiple faults, yet Losq's system is not the standard self-purging system shown in Figure 3.

Reconfiguration scheme

Su and DuCasse[14] have suggested a hardware reconfiguration technique for tolerating logic failures. Their scheme begins with a basic 5MR (static redundancy with five modules), it will reconfigure into a TMR under single or double failures. If just a single fault occurs in the 5MR, the system will reconfigure to a TMR with a spare. The equation for TMR can be obtained by substituting any variable, I_i by 0 and any I_j by 1 where $j \neq i$. If the ith module is faulty, Su and DuCasse's scheme forces the output of the faulty module to be stuck at 0 and the (i+1)th module's output to be stuck at 1. This creates a TMR plus a spare. If another fault occurs, the faulty module's output is set to 0 and the spare module, which had its output stuck at 1, is brought back into the voting. The remaining system is a pure TMR. If two simultaneous faults occur in the 5MR, then the system will reconfigure directly into a TMR.

The basic 5MR system can tolerate only two failures. A five module Hybrid (3,2) can tolerate three single failures but any multiple failure will be catastrophic. However, the 5MR reconfiguration scheme can tolerate either three module faults occurring sequentially or a single fault following a double fault. Also, if three faults occur sequentially and the system is reduced to a TMR with one faulty module. Assuming that the probability of a stuck-at-1 is the same as the probability of a stuck-at-0 fault, there is a 50 percent chance of the system tolerating a fourth module failure.[14] Hence, in some applications the 5MR reconfiguration scheme offers a greater fault tolerance than even the hybrid system.

Another advantage of the 5MR reconfiguration scheme over hybrid redundancy is the surprising simplicity of the switching system. Su and DuCasse[14] designed a switching mechanism which will perform the necessary reconfiguration function in the 5MR scheme with only five gates and one flip-flop. It has also been shown that the reconfiguration scheme can be generalized to provide a fault-tolerant system for multiple-valued functions.[14]

It is possible to design NMR reconfiguration systems where $N>5$. However, the switching system for a NMR reconfiguration scheme ($N>5$) is no longer as simple as the 5MR reconfiguration switching mechanism. In fact, the complexity of the switching system is a function of N. Hence, as N increases, the reliability of the NMR reconfiguration scheme decreases.

Comparison of redundancy systems

All five redundancy approaches have their own advantages and disadvantages. There always will be some sort of trade-off in selecting one scheme over another. McCluskey, Wakerly and Ogus[6] offer three guidelines for choosing a particular technique:

(1) For extremely short mission times static redundancy is optimal.
(2) For mission times whose duration is of the same order of magnitude as non-redundant system mean life, self-purging redundancy provides the best performance.
(3) For very long mission times, hybrid redundancy is best, unless unpowered modules have the same failure rate as powered modules—in which case self-purging redundancy should be used.

McCluskey, Wakerly and Ogus did not analyze the reconfiguration scheme, so a fourth guideline could be added to the list:

(4) For systems where multiple-faults are likely or where hybrid redundancy is used, the reconfiguration scheme may be optimal.

In the next two sections, error correcting codes and software systems are briefly discussed as alternatives to a hardware fault-tolerant system. In practice, however, it is probably best to use them in conjunction with one of the hardware redundant schemes.

ERROR CORRECTING CODES

Error correcting codes refer to a set of transformations on the digital representation of data and an associated set of checking and correcting algorithms. The transformation could be adding redundant bits or actually changing the form of the data depending upon the nature of the checking and correcting algorithms.[15] These codes provide concurrent fault diagnosis and correction. In a sense, they perform the same masking function of static redundancy with, however, some loss in processor time. There are a number of such codes and reviewing them all is beyond the scope of this paper. Instead a few brief comments on the general nature of error correcting codes will be made.

The encoding process introduces redundant bits, requiring additional hardware in the form of longer word lengths. This also requires additional hardware in the memory, processor, and data transfer systems. In addition, the encoding process is performed by a given algorithm which may be fairly complex. Thus the encoding process increases processor time and hardware requirements.

The checking and correction system also requires additional hardware and computing time. Also there is some concern, as with the switching systems of redundancy schemes, for the reliability of the checking and correction hardware. An error in this hardware would be catastrophic for the fault-tolerant abilities of the system.

Thus, the error correcting code process increases the hardware requirements of the system as well as slowing down the computing speed of the system. The speed of the

system is decreased since it now has to convert data to the coded form and reconvert it on output as well as check the code for errors. These factors must be considered in comparing error correcting codes to hardware redundancy processes. Yet, these codes are very useful in providing concurrent correction along data paths, especially between peripheral units and the main systems.

SOFTWARE FAULT-TOLERANCE

Another approach to fault-tolerant computing is to build the fault-handling capabilities into the software. In other words, the system software operates in such a manner as to allow the system to recover from hardware failures. Such a software system works best against transient faults. There are two advantages to a software approach to fault-tolerance. First, fault-tolerant capabilities can be placed in the system after the hardware has been designed. Second, the fault-tolerant capabilities of the system are easily changed at a later date. No hardware rewiring is required, the software is just rewritten to allow for changes in fault-tolerant needs. On the other hand, there are several disadvantages to software fault-tolerance. Assuring that the software will operate correctly in the presence of a hardware fault is the major one. In addition, software now represents a much larger percentage of total system cost than hardware. The development costs of additional software for fault-tolerance could be excessive. The additional software would also require extra storage and in some cases redundant storage.

The major software fault-tolerant approach is called rollback and recovery.[16] Sometimes called time redundancy, this system involves some sort of fault detection procedure and program restarts after a fault occurs. In its simplest form, rollback and recovery involves just instruction retry. If a fault is detected after an instruction is executed then the instruction is reexecuted. More complex forms of rollback and recovery require checkpointing. A checkpoint is a point in time when the system stops its normal processing of user jobs and saves all relevant information in the current state of the system. A checkpoint may be static in which the checkpoints occur at the same time every day or it may be dynamic in which the checkpoints occur at different times dependent on system load and other factors. If any error is detected, the system rolls back and starts reprocessing everything done since the last checkpoint.

There are several questions which need to be considered before installing a rollback and recovery system. For example, how many checkpoints are needed? If too many checkpoints are used, the availability of the system decreases since it has to stop processing the normal flow of jobs to complete the checkpoint. If the checkpoints are too far apart, then the recovery time will be high since the system has to reprocess everything since the last checkpoint. Chandy[16] has reviewed several rollback and recovery models and has calculated the optimum intercheckpoint time which depends on the use of the system. O'Brien[17] has also studied a number of different checkpoint insertion strategies.

It is important to remember that with software fault-tolerance, a single permanent hardware fault could cause the system to crash. Hence, software fault-tolerance works best in a hardware fault-tolerant environment as a supplemental system used to improve fault-tolerant reliability. In addition, software fault-tolerant systems work especially well in providing individual protection to large data bases.

NEW APPROACHES TO FAULT-TOLERANT DESIGN

In the last few years a large research effort in fault-tolerant design techniques has been going on.[18] This research has led to the development of a number of promising alternatives to the standard fault-tolerant design procedures already mentioned. This section will briefly examine several of these alternatives.

Bi-duplexed systems

Courtois[19] has suggested that safety may, in some cases, be a more important concept than reliability. Safety is the probability of not sending incorrect data. This is especially true in biomedical systems. In such systems it may be better for a computer to stop operating rather than to take an incorrect action or give out incorrect data on a patient. Courtois found that a bi-duplexed (BDR) redundant system has better safety than hybrid systems and it is less complex than hybrid systems. The BDR operates with 4 units whose outputs pass through exclusive-OR gates which stop the operation of the incorrect modules. In Figure 4, the stop signal is generated only if both exclusive-OR gates are 1. Otherwise, the output box selects the output from the modules which have an exclusive-OR output equal to one.

Alternating logic

Systems designed using alternating logic design proposed by Reynolds and Metze[20] possess built-in fault detection capabilities. Hence, such systems, while not in and of themselves fault-tolerant, could easily fit into dynamic redundancy or any of the other redundancy schemes already reviewed. The fault detection is attained through a

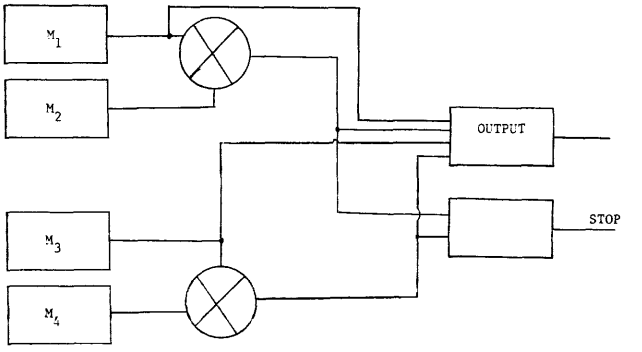

Figure 4—Bi-duplexed system

redundancy in time by successive execution of the function and its dual. Reynolds and Metze have shown that any combinational circuit can be realized using alternating logic design techniques. Since every input to a circuit is an alternating binary sequence of the form (d,$\bar{d}$), the circuit operation is slower. It must process both d and $\bar{d}$ which represent the same information. A fault is detected if the output is not an alternating sequence.

Fail-soft

A concern for high availability of a computing system prompted the development of fail-soft approaches to fault-tolerance. A fault in a fail-soft (graceful degradation) system results in a reduction in system performance. The system remains available to users. An example of a fail-soft system is the PRIME computer system.[21] PRIME has a distributed architecture with four module types: intelligence modules, interconnection modules, memory modules and storage modules.[22]

The University of California at Irvine also has a fail-soft system called the Distributed Computing System (DCS).[23] The DCS distributes hardware, software, and control of the system over a network. The hardware is in the form of a ring of processors (modules) with a software nucleus in each processor. The rest of the software is spread throughout the system. This DCS can tolerate a module failure and continue operation at a lower performance level, that is with fewer processors. Figure 5 shows a DCS with six processors.

Shared logic

For TMR systems to operate correctly the assumption of fault independence is important. This assumption is violated if any of the modules in a TMR share the same logic gates. Hence, three complete copies of each module must be constructed. Osman and Weiss,[24] however, have developed a means of allowing modules to share the same logic called (n,m,r)-basis realizations. They found that a TMR implemented with logic sharing using (n,m,r)-basis realizations would retain the same fault-tolerant characteristics. Yet, the logic sharing means that three copies of each module do not have to be constructed. This results in a considerable savings in hardware.

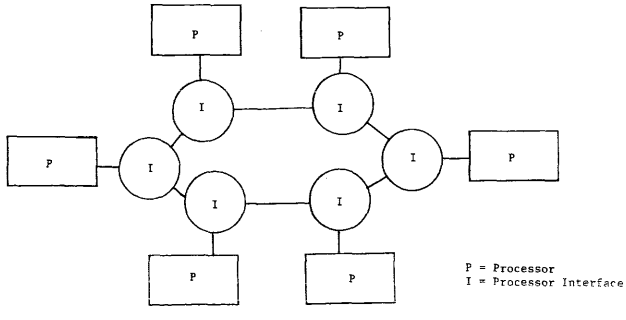

Figure 5—A DCS with six processors

Fail-safe

In the discussion of the bi-duplexed system the concept of safety was introduced in which a system will stop operating rather than generate incorrect data. Fail-safe systems operate under a similar concept. They take into account the fact that an incorrect output of one value may be worse than an incorrect output of another value. That is, the damage caused by outputting an incorrect 1 may be greater than the damage caused by outputting an incorrect 0. In such a case a fail-safe system would produce a 0 output whenever a failure occurs. Hence, an output of 1 would always be correct, an output of 0 may or may not be correct. Such a system is called a 0 fail-safe system.[27] A 1 fail-safe system could be defined in a similar manner. Fail-safe systems are important in biomedical applications where an error resulting in an unnecessary call for a doctor's assistance is not as harmful as not calling for a doctor when the patient has an urgent need for medical treatment. Also, in military systems, an incorrect signal resulting in the launching of a missile can do more damage than the failure to launch a missile when needed. Fail-safe systems for biomedical and military applications would be designed such that if a failure occurs the least harmful output is produced. Hence, a fail-safe system is defined as a system that produces safe-side outputs when failures occur.[28]

Fail-safe systems are subject to problems under multiple fault conditions. A "masked" fault could occur which is a fault that does not affect the output but which could break the fail-safeness of the circuit when another fault occurs.[29] Mukai and Tohma[29] have developed a method of designing asynchronous circuits in which "masked" faults do not occur. In addition, Chuang[30] has also developed a new fail-safe asynchronous design procedure. Chuang's system can handle multiple-input changes which ordinarily could cause a fail-safe machine to produce non-safe-side outputs under certain conditions.

A fail-safe system usually requires twice as much hardware as a non-fail-safe system. This raises some questions on circuit reliability since the more components in a system the higher the chance of a circuit failure. However, Swain[31] has examined this problem and produced a design procedure that will reduce the amount of hardware required to achieve a fail-safe system.

CONCLUSIONS

The need for fault-tolerant computing cannot be understated, especially in light of the role computers now play in medicine, space, and other highly critical areas. This paper has briefly reviewed several methods of achieving fault-

tolerance. They all have their own advantages and disadvantages. Perhaps the best overall approach to a truly fault-tolerant system would be some combination of the techniques reviewed in this paper.

In fact, several computing systems have been proposed or are in the process of being constructed which utilize a combination of the fault-tolerant procedures suggested in this paper. For example, the STAR (Self Testing And Repairing) computer at the Jet Propulsion Laboratory uses the following methods of tolerance:[32]

(1) Dynamic redundancy technique with unpowered spares is used. Replacement is implemented by power switching.
(2) Error correcting codes are used for all data and instruction words with concurrent fault detection.
(3) Fault detection, recovery and replacement are achieved with special purpose hardware.
(4) Software recovery techniques are used in the event of a memory failure.
(5) The processor is protected with hybrid redundancy with three active units.

The design of the STAR computer is constantly being updated but it always employes several different fault-tolerant techniques to achieve optimum fault-tolerant performance. Hopkins[33,34] has proposed a fault-tolerant computer for space vehicles which also uses several fault-tolerant techniques. He suggests a multi-processor system for fail-soft behavior. Each processor unit consists of duplicated processors for error detection plus a triplicated scratch pad memory. The memory unit is protected with an error detection and correction code. Several different redundancy techniques are also used in the system.

Wensley, Levitt, and Neumann,[35] after studying several fault-tolerant architectures, concluded that it is economically feasible to build systems using several different fault-tolerant techniques and achieve a mean time between failures greater than 10 years. For the same cost as duplication they also concluded that it was possible to achieve a mean time between failures in excess of 100 years. Yet, they did find significant deficiencies in the state-of-the-art of fault-tolerant design. Hence, additional research into fault-tolerant computing is vital. Several areas need further investigation. For example, hardware redundancy systems will reject a module even if the detected fault was only a transient. Retry procedures need to be developed so that otherwise "good" modules (modules with just transient faults) will not be switched out of redundant circuits. Goldberg[25] has suggested several other areas for future research. Such as the study of systems that can tolerate faulty human operators, or the application of artificial intelligence to achieve highly flexible fault-tolerant systems.

REFERENCES

1. Avizienis, A. "Fault tolerant computing-An overview", *IEEE Trans. Compt.*, vol. 4, January 1971, pp. 5-8.
2. Avizienis, A. "Architecture of fault-tolerant computing systems," *Digest of the Fifth International Symposium on Fault-Tolerant Computing*, Paris, France, June 1975, pp. 3-16.
3. Avizienis, A. "Design of Fault-Tolerant Computers," *AFIPS Conference Proceedings*, Vol. 31, pp. 733-743, AFIPS Press, Montvale, New Jersey, 1967.
4. Carter, W. C. and W. G. Bouricius, "A Survey of Fault-Tolerant Computer Architecture and Its Evaluation," pp. 9-16, *Computer*, Jan/Feb. 1971.
5. von Neumann, J. "Probabilistic logics and the synthesis of reliable organisms from unreliable components," *Automata Studies*, C. E. Shannon and J. McCarthy, eds., pp. 43-98, Princeton University Press, New Jersey, 1956.
6. McCluskey, E. J., J. F. Wakerly and R. C. Ogus, "*Center for reliable computing: Current Research,*" Stanford Digital Systems Lab Technical Report 100, Oct. 1975.
7. Chandy, K. M., C. V. Ramamoorthy and A. Cowan, "A framework for hardware-software tradeoffs in the design of fault-tolerant computers," *AFIPS Conference Proceedings*, Vol. 41, pp. 55-63, AFIPS Press, Montvale, New Jersey, 1972.
8. Sieworek, D. P. and E. J. McCluskey, "An Iterative Cell Switch Design For Hybrid Redundancy," *IEEE Trans. Compt.*, Vol. c-22, March 1973, pp. 290-297.
9. Ramamoorthy, C. V. and Y. Han, "Reliability analysis of systems with concurrent error detection," *IEEE Trans. Compt.*, Vol. c-24, Sept. 1975, pp. 868-878.
10. Mathur, F. P. and A. Avizienis, "Reliability analysis and architecture of a hybrid-redundant digital system: Generalized triple modular redundancy with self-repair," *AFIPS Conference Proceedings*, Vol. 36, pp. 375-383, AFIPS Press, Montvale, New Jersey, 1970.
11. Cochi, B. "Reliability modeling and analysis of hybrid redundancy," *Digest of the Fifth International Symposium on Fault-Tolerant Computing*, pp. 75-80, Paris France, June 1975.
12. Losq, J. "A Highly Efficient Redundancy Scheme: Self-purging redundancy," *IEEE Trans. Compt.*, Vol. c-25, June 1976, pp. 569-578.
13. Losq, J. "Redundancy scheme for optimum multiple fault-tolerance," *Technical Note no. 33*, Digital Systems Laboratory, Stanford University, Stanford California, Jan. 1974.
14. Su, S. Y. H. and E. DuCasse, "A reconfiguration scheme for tolerating multiple failures in digital systems," *Proceedings of International Computer Symposium*, 1975, pp. 216-222.
15. Avizienis, A. "Arithmetic error codes: Cost and effectiveness studies for application in digital system design," *IEEE Trans. Compt.*, Vol. c-20, Nov. 1971, pp. 1322-1331.
16. Chandy, K. M. "A survey of analytic models of rollback and recovery strategies," *Computer*, May 1975, pp. 40-47.
17. O'Brien, F. "Rollback point insertion strategies," *Proceedings of The Sixth International Symposium on Fault-Tolerant Computing*, Pittsburgh, Pennsylvania, June 1976, pp. 138-142.
18. "Special issues on Fault-Tolerant Computing," *IEEE Trans. Compt* Nov. 1971, March 1973, July 1974, May 1975, June 1976.
19. Courtois, B. "On Balancing Safety and Reliability of Hybrid and Biduplexed systems," *Proceedings of the Sixth International Symposium on Fault-Tolerant Computing*, Pittsburgh, Pennsylvania, June 1976, pp. 53-57.
20. Reynolds, D. and G. Metze, "Fault Detection Capabilities of Alternating Logic," *Proceedings of the Sixth International Symposium on Fault-Tolerant Computing*, Pittsburgh, Pennsylvania, June 1976, pp. 157-162.
21. Baskin, H. B., B. R. Borgerson and R. Roberts, "PRIME—A modular architecture for terminal-oriented systems," *Proceedings of the Spring Joint Computer Conf.*, 1972, pp. 431-437.
22. Borgerson, B. and R. Freitus, "A reliability model for gracefully degrading and stand by-sparing systems," *IEEE Trans. Compt.*, c-24, May 1975, pp. 517-525.
23. Rowe, L., M. Hopwood and D. Farber, "Software methods for achieving fail-soft behavior in the distributed computing system," *Record of the 1973 Symposium on Computer Software Reliability*, New York, April 1973, pp. 7-11.
24. Osman, M. and C. Weiss, "Shared logic realizations of dynamically self-checked and fault-tolerant logic," *IEEE Trans. Compt.*, Vol. c-22, March 1973, pp. 298-306.
25. Goldberg, J. "New problems in fault-tolerant computing," *Digest of the*

Fifth International Symposium on Fault-Tolerant Computing, Paris, France, June 1975, pp. 29-34.
26. Ogus, R. "Fault-tolerance of the iterative cell array switch for hybrid redundancy," *Digest of the Third International Symposium on Fault-Tolerant Computing*, Palo Alto, California, June 1973, pp. 107-112.
27. Mine, H. and Y. Koga, "Basic properties and a construction method for fail-safe logical systems," *IEEE Trans. Compt.*, Vol. EC-16, June 1967, pp. 282-289.
28. Tohma, Y., Y. Ohyama and R. Sakai, "Realization of fail-safe sequential machines by using a k-out-of-n code," *IEEE Trans. Compt.*, Vol. C-20, November 1971, pp. 1270-1275.
29. Mukai, Y. and Y. Tohma, "A masked-fault-free realization of fail-safe asynchronous sequential circuits," *Proceedings of the Sixth International Symposium on Fault-Tolerant Computing*, Pittsburgh, Pennsylvania, June 1976, pp. 69-74.
30. Chuang, H. Y. H. "Fail-safe asynchronous machines with multiple-input changes," *IEEETC*. Vol. C-25, June 1976, pp. 637-642.
31. Swain, D. H. "Fail-safe synchronous sequential machines using modified on-set realizations," *Digest of the Fourth International Symposium on Fault-Tolerant Computing*, pp. (3-7)-(3-12), Urbana, Illinois, June 1974.
32. Avizienis, A., G. C. Gilley, F. P. Mathur, D. A. Rennels, J. A. Rohr, and D. K. Rubin, "The STAR (Self Testing And Repairing) computer: An investigation of the theory and practice of fault-tolerant computer design," *IEEE Trans. Compt.*, Vol. C-20, November 1971, pp. 1312-1321.
33. Hopkins, A. "A fault-tolerant information processing concept for space vehicles," *IEEE Trans. Compt.*, Vol. C-20, November 1971, pp. 1394-1403.
34. Hopkins, A. and T. B. Smith, "The architectural elements of a symmetric fault-tolerant multiprocessor," *IEEE Trans. Compt.*, Vol. C-24, May 1975, pp. 498-504.
35. Wensley, J. H., K. N. Levitt, and P. G. Neumann, "A comparative study of architectures for fault-tolerance," *Digest of the Fourth International Symposium on Fault-Tolerant Computing*, Urbana, Illinois, June 1974, pp. (4-16)-(4-21).