Approaches to computer reliability—Then and now*

by ALGIRDAS AVIŽIENIS
University of California
Los Angeles, California

To the ingenious designers and programmers
of the first generation
Who made their machines work in spite of
most contrary components
And inspired their successors to continue
striving for reliable computing

ABSTRACT

Approaches to the attainment of reliable computer operation are considered in this paper. The goal is to assure correct execution of programs using less than perfect components. The discussion includes design methodology, fault classification, redundancy techniques, reliability modeling and prediction, and examples of fault-tolerant computers. The last section identifies some relationships between reliability methods for hardware and for software.

HISTORICAL PERSPECTIVE

The problem of reliability has confronted both the designers and the users of computing systems since the building of the first computers in the 1940's. First-generation digital computers used large numbers of vacuum tubes, relays, and other electromechanical devices which were notably failure-prone. Various methods of failure detection and recovery were incorporated in the hardware of these machines. For example, duplicate arithmetic-logic units were used in the EDVAC and UNIVAC computers; various error-detecting codes were used in many others, including RAYDAC, IBM 650, NORC, etc.[1]

The advent of transistor technology in the second generation led to a very large improvement in component reliability. This improvement, in turn, led to a deemphasis of failure detection techniques in the hardware. The remaining exceptions were parity checking and related techniques in storage and I/O equipment. All failures, however, were not eliminated, and the absence of hardware checking led to the rapid development and extensive use of diagnostic programs. The diagnostics were employed to perform periodic checkouts of computers and to assist maintenance specialists by identifying failed parts during repairs. Widespread use of diagnostics began in the late 1950's and has continued into the present, with microdiagnosis largely superseding diagnosis in the middle to late 1960's.

Diagnosis-aided manual repair, however, proved in many cases to be an insufficient solution because of at least three reasons: (1) the unacceptability of the delays and interruptions of real-time programs caused by manual repair action; (2) the inaccessibility of some systems to manual repair; and (3) the excessively high cost of lost time and of maintenance in many installations.

Since the early 1960's the scope of computer applications has steadily expanded, encompassing numerous areas of critical importance. These applications include real-time control of communication and transportation systems, manned space flights, automated factories and power plants. At the present, use of computers is being considered for the monitoring of critically ill patients in hospitals. The reliability requirements for computers in such applications far exceed the requirements established for the computing systems of the 1950's and 1960's. The expected great benefits of computer use are balanced against the potentially disastrous costs of their failure.

Another relevant development of the past decade has been the wide distribution of computing systems throughout the entire planet and their use in space. Instead of being concentrated in a limited number of population centers, computers are now performing important, and even critical tasks in many locations that are remote from the service and repair facilities and personnel. Computers have been employed in space

* This work was supported by the National Science Foundation, Grant No. DCR 72-03633 A03

vehicles orbiting the Earth and the Moon, or traveling to the other planets of the solar system. In these applications fully automatic detection of faults, program restart, and self-repair are either absolute requirements, or economic necessities in order to provide reliable computing at an acceptable cost or risk to the user.

In the most general sense, reliable computing means "the correct execution of a specified set of algorithms", and encompasses all the following elements:

—correctness and completeness of software specifications;
—testing and verification (proofs) of programs;
—elimination of hardware design errors;
—continued correct execution of programs and protection of data in the presence of hardware failures;
—protection of the computing system against error-induced disruption or deliberate invasion of programs and data.

Attempts to meet the requirement for reliable computing, especially when external assistance by maintenance specialists is too slow, too costly, or not available, have utilized the two complementary approaches of *fault-intolerance* and *fault-tolerance.*[2] These approaches are applicable to all parts of the computing system, including its hardware elements, microprograms, system programs, and user programs.

In the "fault-intolerance" approach the reliability of computing is assured by *a priori* elimination of the causes of unreliability, i.e., of faults. This elimination takes place before the normal computing process, and the resources that are allocated to attain reliability are spent on perfecting the system prior to its field use. Since in practice it has not been possible to assure complete *a priori* elimination of all causes of unreliability, the goal of fault-intolerance is to reduce the unreliability (expressed as the probability of system failure over the duration of the specified computing process) or the unavailability to an acceptably low value. To supplement this approach, manual maintenance procedures must be devised which return the system to an operating condition after a failure. The cost of providing readily available maintenance and the cost of the disruption and delay in the computing process also are parts of the overall cost of using the fault-intolerance approach.

In the "fault-tolerance" approach the reliability of computing is assured by the use of protective redundancy. The causes of unreliability are expected to be present and to induce errors during the computing process, but their disrupting effects are automatically counteracted by the redundancy. Reliable computing is made possible despite the remaining program and hardware design errors, hardware failures, and external interference with computer operation. The resources allocated to attain reliability are spent on protective redundancy. The redundant parts of the system (both hardware and software) either take part in the computing process or are present in a standby condition, ready to act automatically to preserve its undisrupted continuation. In contrast, we note that the maintenance procedures in a fault-intolerant system are invoked after the computing process has been disrupted, and the system remains "down" for the duration of the maintenance period.

It is evident that the two approaches are complementary and that the resources allocated to attain the required reliability of computing may be divided between fault-tolerance and fault-intolerance. Experience and analysis both point to the conclusion that a balanced allocation of resources between the two approaches is most likely to yield the highest reliability of computing. An overview of past practice shows that fault-intolerance has been the dominant choice in both hardware and software in the 1950's and 1960's. In recent years the fault-tolerance approach has been making significant inroads in hardware system design; its application in software has remained very limited. The cost of redundancy has been the main argument against the use of fault-tolerance techniques in computer systems. The evolution of component technology into large-scale integration, the decreasing cost of mass-produced hardware elements, the very high cost of software testing, and the increasing reliability requirements all favor increasing use of fault-tolerance in computer systems of the future. The resistance to its wider use frequently originates with the practitioners of the current fault-intolerance and manual maintenance methods.

DESIGN METHODOLOGY FOR RELIABLY OPERATING COMPUTERS

This section and the subsequent sections address the issue of providing undisrupted computing while using less than perfect hardware components. The remaining aspects of reliable computing remain outside the scope of this paper.

Unreliable operation is caused by imperfections in the physical implementation of the computer's logic structure. Reliability theory defines the reliability $R(T)$ of a system as the probability of its correct operation up to the time $t=T$, given that the system was operating correctly at the starting time $t=0$. Computers differ from other systems because in their case "correct operation" means the correct execution of a set of programs and protection of data, rather than the continued functioning of a set of physical components of the system. It is the purpose of this section to present those aspects of computer system design that are specifically directed toward the elimination or tolerance of imperfections (called "faults") in the components of the system. It is to be noted that we discuss correct *execution* of a given set of programs and do not include

the questions of correctness of the programs, completeness of their specification, and of accuracy of the algorithms, which remain separate fields of study.

The architects and the users originate the sets of programs and data, the definitions of required operations, the time limits for program execution, and the storage requirements. The objective of the designer is to raise the reliability (i.e., the probability of correct execution of these programs) or availability to an acceptably high value, given that *operational faults* may occur during execution. Such faults are caused by three classes of physical events that affect the hardware of the system:

—permanent failures of hardware components;
—intermittent malfunctions of components;
—external interference with computer operation.

As discussed previously, two complementary approaches have been employed to attain satisfactory reliability. Fault-intolerance is the approach that aims to reduce the probability of occurrence of the first fault during a specified time interval to an acceptably low value. In the "pure" fault-intolerance approach the system is designed without redundancy, and every component of the system must function correctly in order to assure correct program execution. The procedures which lead to the attainment of reliable "fault-intolerant" systems are:

—the most reliable components are acquired within the existing cost and performance constraints;
—proven techniques are employed for the interconnection of components and assembly of subsystems;
—the system is packaged to screen out the expected forms of external interference;
—quantitative prediction of system reliability is made using known or predicted failure rates for the components and interconnections.

In the "purely" fault-intolerant (i.e., non-redundant) design, the probability of fault-free hardware operation is equated to the probability of correct program execution. Such a design is characterized by the decision to invest *all* the reliability resources into procurement of high-reliability components and refinement of assembly and packaging techniques. An alternative to the "purely" fault-intolerant approach is offered by the use of various forms of redundancy to attain fault-tolerance.[3] This approach increases reliability by the use of design techniques that allow faults to occur without disrupting the continued correct execution of the programs. Fault-tolerance does not entirely eliminate the need for reliable components; instead, it offers the option to allocate part of the reliability resources to the inclusion of redundancy. The goal of a fault-tolerant design is either a reliability (or availability) prediction that cannot be attained by the purely fault-intolerant design, or a reliability (or availability) prediction that matches the purely fault-intolerant design at a lower overall cost of implementation.

A fault-tolerant computer system must possess the following attributes:[4]

—Its description includes a set of components (hardware) and a set of programs (software).
—It is either initially free of design faults or it is protected against their disruptive effects during program execution.
—It executes the set of programs correctly in the presence of operational faults.

The first attribute stresses the fact that the ability of a computer to continue operating correctly in the presence of operational faults depends not only on the properties of the hardware, but also on the nature of the software, including both the system programs and the user programs. For example, the ability to recover from the errors caused by transient faults frequently depends on special restart features incorporated in the system software as well as on proper partitioning and state vector storage of user programs.

The second attribute requires that design faults should be eliminated from both hardware and software prior to the initiation of the computing process. As an alternative, protective features for the detection and circumvention of design faults in both hardware and software must be incorporated to make the system fault-tolerant. Design faults are caused by errors made during the translation of the original specifications into operational forms, that is, into assemblies of components and of machine language instructions. They are eliminated by validation of the hardware and software designs prior to their operational use. Since complete *a priori* verification cannot yet be assured, computers need protective provisions to detect and circumvent abnormal conditions encountered during operation which may be symptoms of remaining design faults. A completely fault-tolerant operation is attained either when all design faults are eliminated from the system, or when complete protection against remaining design faults is incorporated.

The third attribute of a fault-tolerant computer postulates correct execution of the entire set of programs in the presence of operational faults. Program errors that are caused by faults in the hardware can be avoided or corrected by means of protective redundancy. Protective redundancy may be introduced in three forms:

—additional hardware (hardware redundancy);
—additional software (software redundancy);
—repetition of operations (time redundancy).

These redundant features would not be needed in a fault-free computer; that is, their deletion does not affect computer performance in the absence of operational faults. Given that the faults will occur in the hardware, the redundant features provide a fault-

tolerant computing system which carries out its programs correctly in the presence of operational faults. Partial fault-tolerance (also called "fail-soft operation" or "graceful degradation") occurs when operation continues, but one or more programs are not correctly executed in the specified time.

Research results and design experience lead us to suggest that the introduction of protective redundancy can be accomplished by following a systematic procedure:[4]

(1) Performance requirements are established and system architecture is specified with the initial assumption that operational faults will not occur (the "fault-intolerant" design).
(2) Classes of operational faults that are to be tolerated in the design are identified, and the extent of tolerance is specified for each class of faults.
(3) Cost-effective methods of protective redundancy (time, hardware, software) are chosen to cover every class of faults identified above, and system architecture is modified to incorporate the redundancy.
(4) Analytic or experimental techniques are employed to estimate the extent of fault-tolerance that is provided by the protective redundancy.
(5) Checkout methods are devised to test all redundancy features. Where applicable, fault-tolerance is extended to effect automatic maintenance of peripheral systems that are connected to or controlled by the computer.

Design experience has shown that several iterations of (3) and (4) may be necessary to arrive at a satisfactory fault-tolerant system architecture. The following sections discuss the techniques for the implementation of (2) to (5) and illustrate their use in recent computer systems and in proposed designs.

CLASSES OF OPERATIONAL FAULTS

An operational fault is the deviation of one or more logic variables in the computer hardware from their design-specified values. Faults are caused by failures, which are physical changes in the hardware of the computer. Hardware failures are of three types: "solid" component failures, "intermittent" component malfunctions, and externally caused interference with the operation of the computer. The immediate symptom of any hardware failure is a fault. The fault often causes an error in the program being executed by the computer: either an instruction is not executed correctly, or an incorrect result is computed. Both types of errors may be caused at once by some faults.

A systematic approach to the choice of redundancy techniques in computer design begins with a classification of faults and identification of those classes which are expected to occur in the system being designed. Three useful dimensions for the classification of faults are:[4]

—*duration:* transient (intermittent) vs. permanent (solid);
—*extent:* local (single) vs. distributed (related multiple);
—*value:* determinate ("stuck") vs. indeterminate (variable).

In the design methodology the "permanent vs. transient" classification appears to be most fundamental because the two classes usually need different recovery methods. A program restart is sufficient to correct errors caused by transient faults, while replacement or reconfiguration of hardware is needed to eliminate permanent faults from the system. The classifications according to extent and according to value are applicable to both transient and permanent faults.

The extent of a fault specifies how many logic variables in the hardware are simultaneously affected by the fault which is due to a single failure event. *Local* (single) faults are those that affect only single logic variables, while *distributed* (related multiple) faults are those that affect two or more variables, one module, or an entire system. The physical proximity of logic elements in contemporary MSI and LSI circuitry has made distributed faults much more likely than in the discrete component designs of the past. Distributed faults are also caused by single failures of some critical central elements in a computer system, for example: clocks, power supplies, data buses, switches for computer reconfiguration, etc.

The value of a fault is *determinate* when the logic values affected by the fault assume a constant value ("stuck on 0" or "stuck on 1") during its entire duration. The fault value is *indeterminate* when it varies between "0" and "1", but not in accord with design specifications, during the duration of the fault.

It is important to observe that a precise description of fault extent and fault value can only be made at the source of the fault, that is, at the point at which the hardware failure event has actually taken place. The introduction of one or more faulty logic variables into the computing process will often lead to different or more extensive fault symptoms downstream from the point of failure. For example, a "stuck on 1" local determinate fault on the input to a two-input "Exclusive-Or" gate will cause the output variable of the gate to appear as a local indeterminate fault. Furthermore, if this output is supplied as an input to several other gates, the set of output variables of these gates will appear as a distributed indeterminate fault.

Ambiguity is avoided when the term "fault" is restricted to the change in logic variable(s) at the point of the actual hardware failure. The fault-caused changes of logic variables which are observed (because of faulty inputs) on the outputs of correctly functioning logic elements are symptoms of the fault and will

be called errors. This distinction establishes a cause-effect sequence as follows:

(1) The *failure* which is a physical event, causes a *fault,* which is a change of logic variable(s) at the point of failure.
(2) The *fault,* in turn, supplies incorrect input(s) to the computing process and causes an *error* to be produced by subsequent operation of failure-free logic circuits.

The preceding discussion makes it evident that the detectability of a fault depends not only on its type, but also on the distance (in terms of computing operations) from the point at which the fault occurs to the point at which checking (fault-detection) is performed. A local determinate fault may cause an extensive error pattern to appear at a point that is several computing steps (in time, space, or both) removed from the fault itself.

METHODS OF PROTECTIVE REDUNDANCY

The key to successful application of protective redundancy is the systematic and balanced selection of suitable methods of its three forms: *hardware* (additional components), *software* (special programs), and *time* (repetition of operations). This section reviews the basic methods of these forms of redundancy.

Hardware redundancy

Hardware redundancy includes the components that have been introduced into the system in order to provide fault-tolerance. As long as faults do not occur, all these components can be deleted without diminishing the computing power of the system. The techniques of introducing hardware redundancy may be divided (on the basis of terminal activity of modules) into two categories: *static* redundancy and *dynamic* redundancy.[5]

The static redundancy method is also known as "masking" redundancy, since the redundant components are employed to mask the effect of hardware failures within a given hardware module, and the terminal activity of the module remains unaffected as long as the protection is effective. The static technique is applicable against both transient and permanent faults. All redundant copies of an element are permanently connected and receive power. Component failures and logic faults are masked by the presence of other copies of the same element. The fault masking occurs instantaneously and automatically; however, if the fault is not susceptible to masking and causes an error, a delayed recovery is not provided.

The original study of the use of static redundancy at the logic element level is due to John von Neumann.[6] He considered transient malfunctions of individual logic gates and showed that arbitrarily high reliability would be attained with high orders of redundancy. Moore and Shannon[7] applied the static redundancy principle to relay contact networks. In practical applications the order of redundancy has to be as low as possible in order to make the cost acceptable to the user. Two forms of static redundancy have been used in practice: replication of individual electronic components in the Orbiting Astronomical Observatory and triple modular redundancy (TMR) with voting in the SATURN IV and V guidance computers.[8] Several other variants of static redundancy have been studied but were not employed in practice because of excessive cost or the need for practically unrealizable special components. It is essential to note that static redundancy is based on the assumption that failures of the individual copies are independent. When related failures take place, the protection by redundancy is lost. For this reason static redundancy (especially at the component level) is not applicable within integrated circuit packages, in which individual components are in close proximity and failure phenomena frequently affect several adjacent components.

In the dynamic redundancy approach fault-caused errors are allowed to appear at the terminals of a module. Fault-tolerance is implemented by two consecutive actions. First, the presence of a fault is detected, then a recovery action either eliminates the fault, or corrects the error. If human assistance is completely eliminated, dynamic redundancy (usually with software support) results in self-repair of a computer system. Limited, that is human- and software-assisted, use of dynamic redundancy techniques in computer hardware has been very extensive.[1,3,5] The most common example is the use of parity to detect errors in data transmission and storage. Important early examples of extensive dynamic redundancy with software and human support are the ESS systems.[9,10] Probably the first operational computer with full self-repair provisions is the JPL-STAR computer.[11]

The application of dynamic redundancy to a computer architecture requires that a number of decisions should be made in the functional design stage. The design choices include: level of modularization, fault-detection hardware, type of recovery action, "hard-core" protection, forms of intermodule communication, validation of inputs, and interfaces with system software.[2] The use of dynamic redundancy has been somewhat inhibited because of the need for an early commitment to it in the hardware design process. In contrast, static redundancy (and software redundancy, as well) can be applied to an existing non-redundant design.

Software redundancy

Software redundancy includes all additional programs, program segments, instructions, and micro-

instructions which would not be needed in a fault-free computer.[2] They provide either fault-detection or recovery in fault-tolerant computer systems, very frequently in conjunction with dynamic hardware redundancy. Three major forms of software redundancy are:

—multiple storage of critical programs and data;
—test and diagnostic programs at various program and microprogram levels;
—fault-tolerance features of the executive program which implement program restarts and interface with the dynamic hardware redundancy.

Combinations of all three forms are found in most modern fault-tolerant computers.

Compared to hardware redundancy, an advantage of software is the ability to superimpose fault-tolerance features after the hardware has been designed. This allows the design of fault-tolerant systems using non-fault-tolerant 'off-the-shelf' hardware. Another advantage is the relatively easier modification and refinement of these software features after their introduction into the system. The main disadvantage of software redundancy is the difficulty of assuring that the software features will be able to function correctly after the occurrence of a fault and that they will be invoked sufficiently early, that is, before the fault-caused errors have irrevocably disrupted the programs or mutilated the data base. Other disadvantages include the relatively high cost of generating the required software, the storage requirements, including the need to tolerate failures of memories holding the software, and the difficulty of estimating and proving the completeness or the adequacy of the software redundancy features. It must be stressed that dynamic hardware redundancy and software redundancy are not mutually exclusive in practice.[12] A system with all-out emphasis on self-contained dynamic hardware techniques still needs cooperation from the executive program to complete some recovery actions. Conversely, an all-software controlled fault-tolerant system has high risks of excessive delays in initiating recovery without at least some hardware methods for fault-detection. It also needs redundant storage modules and hardware protection of critical decision-making logic. Combinations of software and hardware redundancy are employed in most fault-tolerant systems, but they differ in the choice of the point in the detection and recovery sequence at which software takes over control.[9,11–14]

Time (execution) redundancy

This form of redundancy consists of repeating or acknowledging machine operations at various levels: micro-operations, single instructions, program segments, or entire programs. It is usually employed together with dynamic hardware and software redundancy techniques. Two distinct goals of time redundancy are:

—fault detection by means of repeated execution or acknowledgments;
—recovery by program restarts or operation retries after fault detection or reconfiguration has occurred.

The repeated execution of a program is probably the oldest form of fault detection. While suitable to detect errors due to transient faults, it is limited by the fact that consistent errors will be produced by permanent faults, and comparison will fail to reveal the same error in the results. The use of retransmission and of other forms of acknowledgments ("handshakes") has been extensively used in general purpose systems, especially for error detection in secondary storage, channels, and I/O devices.[13,22]

Another common use of time redundancy is found in the identification and correction of errors caused by transient faults, and in program restarts after a hardware reconfiguration.[2] This is accomplished by the repetition after error-detection or 'rollback' of single instructions, segments of programs, or entire programs. While single-instruction retries are transparent to the programmer, longer rollbacks require programming constraints as well as protected storage for the rollback address and for the state vector, including its double-buffering. "Singular" events in a computer are program-controlled events which should not be repeated as part of a program rollback operation, for example, real-time output commands which initiate irreversible actions in the system under computer control. The potential damage makes it imperative that the provision for handling of singular events should be incorporated in rollback procedures.[11,21]

Checkout and extension of fault-tolerance features

The introduction of redundancy poses the problem of verifying that the redundant parts are ready to be used when faults occur. Implementation of checkout encounters difficulties in systems with static hardware redundancy, especially in component-redundant systems.[7,8] Dynamically redundant systems are inherently better suited for redundancy checkouts, since they possess extensive fault-detection and permit sequential switching-in of spare modules to be tested.[11] One critical requirement of checkout is the systematic verification that all fault-indicating signals are operational. Self-checking logic[15] is suitable for this purpose. In software-controlled fault-tolerance this function is carried out by a special fault-signal-test instruction.[16]

The techniques of fault-tolerance can be systematically extended beyond the boundaries of the fault-tolerant computer to effect automatic maintenance of various peripheral systems which communicate with the computer. The methodology of extending fault-

tolerance consists of the development of fault-tolerant interfaces, introduction of fault-detection methods in the systems outside the computer, and programming of recovery sequences to be executed by the computer. A case study of the application of these techniques in a spacecraft system is presented in Reference 17. In commercial general purpose systems, the reverse process has taken place. Because of the relatively high unreliability of peripheral mechanical devices, fault-tolerance began at the peripherals and only later was brought into the CPU and main memory.[28]

FAULT-TOLERANT SYSTEMS

The currently existing and proposed fault-tolerant computer systems may be conveniently classified according to the method used to control the recovery. *Hardware-controlled* systems use dedicated hardware which collects fault indications and initiates recovery. While recovery control may be transferred to software after its operability has been assured, it is completed automatically (without external aid). *Software-controlled* systems depend on special programs to interpret fault indications and to carry out the automatic recovery procedures. *Manually-controlled* systems require the participation of a maintenance operator in the completion of recovery; they are not fault-tolerant in the full sense of the word, although they may employ many fault-tolerance techniques.

Hardware-controlled recovery

This approach depends on special hardware to carry out fault detection and to control the initial recovery procedure. After the procedure has established the existence of an operational software system, the completion of recovery is usually transferred to software control. It is evident that further software systems may be superimposed on the hardware-controlled design, leading to a multilevel recovery procedure. A special case of hardware-controlled recovery is found in statically-redundant systems in which faults are masked by redundant hardware, and are totally invisible to the software. Two examples of such systems are the OAO data processor which used component redundancy and the CPU of the SATURN V guidance computer, which used TMR protection.[8] A separate software-controlled recovery system is needed in statically-redundant systems if they are to continue operating in reconfigured mode after the first fault that escapes the masking effect and affects the software.

Dynamically redundant systems usually depend on a dedicated hardware module that gathers fault signals and initiates recovery. Different uses of duplexing and hardware-controlled switchover techniques are found in the memory, power supply, and peripheral units of SATURN V computer in combination with a TMR-protected serial CPU unit.[8] Separate fault-detection and switchover-control units were used for every functional unit. Probably the first operational computer with fully hardware-controlled dynamic redundancy was the experimental JPL-STAR computer.[11] Intended for self-contained multiyear space missions, this computer employs a special Test-And-Repair-Processor (TARP) module to control recovery and self-repair. Software assistance is invoked only to perform memory copying and to resume standard operation after self-repair.[14] The French MECRA computer is another early experimental design.[18] A few other hardware-controlled system designs that have not reached operation have been described in recent literature.[12,16]

The principal advantage of hardware-controlled recovery systems lies in their independence of the operation of any software immediately after the fault has occurred. The recovery process is transferred to software only after its ability to operate has been assured. The relatively late appearance of such systems may be attributed to the need to introduce the recovery module into the design at its inception, thus requiring an early commitment to the hardware-controlled approach.

Software-controlled recovery

In contrast to the previous class, the software-controlled systems depend on special software to initiate recovery action upon the detection of a fault. Fault signals are obtained by both hardware and software methods, for example: parity checkers, comparators, power level monitors, test programs, reasonableness checks, etc. The main limitation of these systems is the need for the recovery software to remain operational in the presence of faults, since recovery cannot otherwise be initiated.

A significant advantage of this approach is that existing 'off-the-shelf' hardware system modules may be used to assemble fault-tolerant organizations. These modules contain various forms of hardware fault-detection, which usually are supplemented by further software methods. For this reason software-controlled systems have appeared earlier and are currently being used in numerous applications requiring high reliability and availability. While every modern operating system incorporates some recovery features, the present paper will be limited to selected illustrations of historically important and advanced systems.

An important early design of the 1950's with complete duplication and extensive recovery provisions was the SAGE system.[19] The IBM System/360 architecture contains very complete provisions for multi-system operation in order to attain high availability, reconfiguration, and fail-soft operation.[21] An early example of a multi-system which includes further extensions of the System/360 design is the IBM 9020 multiprocessing system for air traffic control applications.[13] Noteworthy are the operational error analysis program and

the diagnostic monitor of the 9020. The recent IBM System/370 hardware incorporates automatic retry of CPU operations, error coding to correct single-bit errors in processor and control storage, and I/O retry facilities. The software provides recovery management support routines, I/O and channel error recovery, checkpoint/restart facilities, microdiagnostics and on-line diagnostics of I/O device errors.[22] An interesting illustration of extensive use of backup storage and dynamic reconfiguration in a general-purpose time-shared system is found in the MIT Multics System.[23] Another experimental system for high-availability performance in an interactive time-shared environment is PRIME.[24] The Pluribus is a minicomputer/multiprocessor system with extensive fault-tolerance provisions which serves as a switching node in the ARPA Network.[20]

Another direction of software controlled system development is in aerospace applications. The principal illustrations of this approach are the SIFT design,[25] the RCS system,[26] and the C.S. Draper Laboratory modular system.[27] One more area of application which requires fault-tolerant operation and very high availability for several years of continuous operation is the control of electronic switching systems for telephone exchanges. These systems usually employ manual repair by replacement of a failed part as the last (off-line) step of the recovery procedure, while maintaining normal operation by means of the remaining system modules. A well documented illustration is found in the ESS systems of Bell Laboratories.[9,10] ESS systems employ a variety of hardware techniques (duplication, matching, error codes, function monitors) and special software (check routines, diagnostics, audits) as well as software and hardware emergency procedures when normal recovery action does not succeed.

Fault-tolerant memories and processors

Besides the complete systems discussed above, significant efforts have been carried out in providing fault-tolerance for storage subsystems. This is especially true for secondary and mass storage which has been characterized by relatively low reliability in the past. Representative error coding applications include the use of codes for error control in data communications, magnetic tape units, disc files, primary random access storage, and a photo-digital mass store.[28] Single-error correcting codes are used in the control storage of ESS No. 1 and the main and control storage of IBM System/370 computers.[9,22] Error-correcting codes have been shown to provide a very effective method for fault-tolerance in the storage medium, and remaining problems are concentrated in providing fault-tolerance in the memory access and readout circuitry.

Recent studies have considered the problem of fault-tolerance in associative memories and processors.[29] In general, processor fault-tolerance has been provided by duplication and reconfiguration at the system level. Some investigations have been conducted in the use of arithmetic error codes as the means for error-detection for processor faults[30] and an experimental processor has been designed and constructed for the JPL-STAR computer.[11] The increasing availability of microprocessors makes further emphasis on duplication very likely, although error-detecting codes remain a convenient method for the identification of the faulty processor in a disagreeing pair.

RELIABILITY MODELING AND PREDICTION

The initial choice of redundancy techniques requires verification that the redundant system possesses the expected fault-tolerance. Insufficiencies of the original design may be uncovered, and the design can be refined by changes or additions of various forms of redundancy. The process is repeated until a fully satisfactory design is attained. The principal quantitative measures are reliability[31,32] (with respect to permanent faults), survivability[4,37] (with respect to transient faults), and availability.[32] Two approaches to the prediction of fault-tolerance are:

—the *analytic* approach, in which fault-tolerance measures of the system are obtained from a mathematical model of the system, and
—the *experimental* approach, in which faults are inserted either into a simulated model of a system, or into a prototype of the actual hardware, and fault-tolerance measures are estimated from statistical data.

A quantitative reliability prediction for the computer being designed requires numerical failure rates for the components. When technologies which are under development are to be used, the failure rates for currently used components need to be extrapolated to the new choice of component technology. It is important to recognize that different failure rates or distributions may apply to failures causing distributed faults. The principal measure of fault tolerance with respect to permanent faults is the reliability $R(t)$, which is a function of the failure rates and directly predicts only the probability of hardware survival. Fault-tolerance is attained only if correct program execution is maintained by the surviving hardware; for this reason transient faults must also be considered. A very common quantitative measure has been the MTBF (mean time between failures), defined as $\text{MTBF} = \int_0^\infty R(t)\,dt$. Given the non-redundant reliability $R(t) = e^{-\lambda t}$, we have $\text{MTBF} = 1/\lambda$, and the comparison of the MTBFs directly compares the total failure rates (λ) of the competing systems. When redundancy is introduced, the reliability function $R(t)$ is a polynomial in $e^{-\lambda t}$ and the $R(t)$ curves of systems being compared may have crossover points. Then the area under the $R(t)$ curve does not indicate which system is better at

a given time, and the MTBF may become a misleading measure. Given a fixed 'mission time' T, the comparison of two or more systems requires only the values of $R(T)$ in order to select the best system. If a fixed mission time is not available, the time interval during which the reliability remains above a given value serves as a convenient comparison measure.[31]

It is essential to note that reliability modeling remains useful even if definite numerical failure rates and mission times are not available, since it permits the comparison of many alternate designs using normalization with respect to the (failure rate × mission time) product λT.

Reliability models

The class of *static* reliability models is suitable for the reliability prediction of systems with static hardware redundancy. The non-redundant system or its element is usually assumed to have the reliability $R(t) = e^{-\lambda t}$. The redundant elements are assumed to be permanently connected, and to fail statistically independently. They have the same failure rate and are instantaneously available to perform the masking of a failure with unity probability of success. Under these assumptions, the reliability of a redundant system is obtained as the sum of the reliabilities of all distinct configurations (including none or some failed parts) that do not lead to system failure. For example, given the simplex (one system) reliability R, the reliability of a duplicated system is $R(duplex) = R^2 + 2R(1-R)$. In general, reliability models of static redundancy are found in standard handbooks and textbooks of reliability theory and are used for reliability analysis of various physical systems.[32]

Dynamic redundancy requires the consecutive actions of fault detection and recovery in order to utilize redundant parts. The use of static reliability models for the dynamic case is equivalent to assuming unity probability of success of both actions; for this reason, very high reliabilities can be predicted as the number of spares is increased. It was recognized early in the studies of dynamic redundancy that imperfect detection and recovery may leave some spares unused.[33] The effect of such imperfections was formalized in the reliability model through the concept of "coverage", defined as the conditional probability of successful recovery, given that a fault has occurred.[31]

In general, the dynamic model[31,34,35] must represent the complete complexity of the proposed fault-tolerant system, including (1) differing failure rates for powered and unpowered modules; (2) number of spares of each module; (3) imperfect fault detection and recovery; (4) method of "hard core" protection; (5) existing intra-module fault tolerance; (6) extent and value of the faults; (7) duration and distribution of expected transient faults. Recent work has considered repairable systems[36] and models that include transient as well as permanent faults.[37,38] The principal objective of further study remains the development of models that integrate the characteristics of both hardware and software and consider both permanent and transient faults.

Analytic models of dynamically redundant systems are complex because of the number of different parameters that may be varied in the search for a balanced design. A very useful tool for reliability prediction is an interactive computer program which permits a ready variation of the important parameters of the redundant system for on-line design refinement. A pioneering effort in automated reliability modeling was the *REL* program, written in the APL language to predict system reliability for a given mission time when the system parameters have been specified.[31] Recent efforts to arrive at general and computationally efficient models have resulted in further APL programs.[38]

Experimental reliability prediction

Two approaches to experimental prediction of reliability are simulation and experimentation with a hardware prototype. While their use is more costly and time-consuming, the experimental methods are essential when the available analytic models do not adequately represent the complex structure of the system or the nature of the expected faults.

An accurate description of the system and characterization of the faults are the principal prerequisites when simulation is employed to derive the fault-tolerance estimates for the computer.[26] This approach has been extensively employed in reliability prediction for the redundant SATURN V guidance computer.[8,39] The use of hardware prototypes requires a large investment of effort in constructing the prototype, but avoids the inaccuracies which may occur in postulating the fault effects in a simulated model of the system. Two examples of use of hardware prototypes are: the switching system ESS No. 1 for which a catalog of fault symptoms was compiled by using a hardware model,[9] and the experimental fault-tolerant JPL-STAR computer.[11] In the JPL-STAR computer an electronic "black box" was used to inject faults of adjustable duration and extent at selected points in the hardware of the system. Another example of the experimental approach is the OAO processor[8] in which a component-redundant system was completely disassembled to determine the number of failed components after 3000 hours of operation.

A recent simulation and analysis system to analyze the behavior of faulty circuits is the LAMP (Logic Analyzer for Maintenance Planning) system.[40] In addition, LAMP also performs logic design verification, generates fault-detection tests, evaluates diagnostics, and produces trouble-location manuals. LAMP exemplifies the current trend toward multipurpose simulation systems in digital system design.

RELATIONSHIPS WITH THE SOFTWARE RELIABILITY PROBLEM

Discussion of the attributes of a fault-tolerant system included the goal that one of the two conditions should be satisfied with respect to design faults:

—the hardware and software should be free of design faults prior to the start of the computing process; or
—the system should contain complete provisions to detect and to circumvent the effects of hardware and software design faults during the computing process.

This means that the software of a fault-tolerant system must either be perfect (i.e., fault-free) or fault-tolerant in the same sense as the fault-tolerant hardware. The basic difference between the two is that operational faults in hardware occur after the start of the computing process, while design faults in software (and hardware, as well) are present at the start, but become disruptive only at a later time. However, software modifications and corrections of discovered design faults occasionally lead to new design faults and therefore the discoveries of software design faults may be expected throughout the useful life of any large software system, similar to the occurrence of operational hardware faults. This practically verified observation establishes the relationship between the methodologies for dealing with operational faults and design faults: the methods of protective redundancy that have proven successful in hardware fault-tolerance may be transferable to provide fault-tolerance of a software system as well.

An overview of the procedures currently used to attain software reliability shows that the "fault-intolerance" approach of perfecting the software prior to its regular use has been the accepted practice of improving software reliability. Three aspects of relevance of fault-tolerance can be identified:[41]

—the contribution of hardware fault-tolerant systems in assuring reliable computing;
—the common aspects of fault-tolerance that are equally applicable to hardware and software;
—the transfer of fault-tolerance techniques and experience from hardware to software, considering:
  —the applicability to software;
  —the potential advantages of software fault-tolerance;
  —the cost of its use, compared against the traditional fault-intolerance techniques.

The immediate advantage to a software system which results from the existence of a fault-tolerant hardware design is the protection of the software against disruptions caused by operational faults. In the case of a fault, the fault-tolerant features execute the corrective action in the hardware and restart the software, usually at a programmer-specified restart point,[14] although in some cases there is a single-instruction restart procedure which is transparent to the programmer.[27] The cost of utilizing the fault-tolerance features to achieve software protection consists of the programming constraints that must be observed to make automatic hardware initiated restarts of the programs possible. The advantages, in addition to the protection itself, also include the ability to distinguish, with a very high probability of success, whether a system crash was hardware-caused or not. Furthermore, a direct extension of the fault-tolerance techniques may be utilized to provide hardware-controlled protection of software and the data base against deliberate attempts to disrupt its operation and to access privileged information.

An area in which a common ground exists for hardware and software reliability efforts is the analytic modeling and quantitative prediction of system reliability.[31] The recent work on software reliability models[42,43] indicates the possibility of mutual reinforcement of research that would lead to the development of analytical models for the total system reliability, including both the hardware and software aspects. A second common area is design verification, in which the rapidly evolving techniques of program testing and proving have obvious applications to the problem of verifying hardware designs.

Finally, we consider the transfer to software of those protective redundancy techniques that have been successfully used in hardware system design. In the static approach, the same computation is carried out by two or more independently written programs.[41,44] The dynamic approach uses an analog of standby sparing with fault detection and switching of software modules.[45] While the cost aspect of both statically and dynamically fault-tolerant software remains to be explored, the continued use of 'pure' fault-intolerance for software reliability cannot be justified by tradition alone. It is hoped that the success of fault-tolerant hardware will stimulate further studies of the merits of fault-tolerance and redundancy in computer software.

REFERENCES

1. Carter, W. C. and W. G. Bouricius, "A Survey of Fault-Tolerant Computer Architecture and Its Evaluation," *Computer*, Vol. 4, No. 1, January-February, 1971, pp. 9-16.
2. Avižienis, A., "Architecture of Fault-Tolerant Computing Systems," *Digest of the 1975 International Symposium on Fault-Tolerant Computing*, Paris, France, June 1975, pp. 3-16.
3. Avižienis, A., "Design of Fault-Tolerant Computers," *AFIPS Conference Proceedings*, Vol. 31, 1967, pp. 733-743.
4. Avižienis, A., "The Methodology of Fault-Tolerant Computing," *Proceedings of the First USA-Japan Computer Conference*, Tokyo, Japan, October 1972, pp. 405-413.
5. Short, R. A., "The Attainment of Reliable Digital Systems Through the Use of Redundancy—A Survey," *IEEE Computer Group News*, Vol. 2, No. 2, March 1968, pp. 2-17.

6. Von Neumann, J., Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components," in *Automata Studies*, C. E. Shannon and J. McCarthy, eds., Annals of Math. Studies No. 34, Princeton, University Press, 1956, pp. 43-98.
7. Moore, E. F. and C. E. Shannon, "Reliable Circuits Using Less Reliable Relays," *Journal of the Franklin Institute*, Vol. 262, Nos. 9 and 10, September, October 1956, pp. 191-208 and 281-297.
8. Cooper, A. E. and W. T. Chow, "Development of On-Board Space Computer Systems," *IBM Journal of Research and Development*, Vol. 20, No. 1, January 1976, pp. 5-19.
9. Downing, R. W., J. S. Nowak and L. S. Tuomenoksa, "No. 1 ESS Maintenance Plan," *The Bell System Technical Journal*, Vol. 43, No. 5, Part 1, September 1964, pp. 1961-2019.
10. Beuscher, H. J. et al., "Administration and Maintenance Plan of No. 2 ESS," *The Bell System Technical Journal*, Vol. 48, October 1969, pp. 2765-2815.
11. Avižienis, A., G. C. Gilley, F. P. Mathur, D. A. Rennels, J. A. Rohr and D. K. Rubin, "The STAR (Self-Testing-And Reparing) Computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design," *IEEE Transactions on Computers*, Vol C-20, No. 11, November 1971, pp. 1312-1321.
12. Carter, W. C., D. C. Jessep, P. R. Schneider, A. B. Wadia and W. G. Bouricius, "Logic Design for Dynamic and Interactive Recovery," *IEEE Transactions on Computers*, Vol. C-20, No. 1, November 1971, pp. 1300-1305.
13. "An Application-Oriented Multiprocessing System," *IBM Systems Journal*, Vol. 6, No. 2, 1967.
14. Rohr, J. A., "STAREX Self-Repair Routines: Software Recovery in the JPL-STAR Computer," *Digest of the 1973 International Symposium on Fault-Tolerant Computing*, Palo Alto, CA, June 1973, pp. 11-16.
15. Carter, W. C. and P. R. Schneider, "Design of Dynamically Checked Computers," *Information Processing 68* (Proceedings of IFIP Congress 1968), pp. 878-883.
16. Conn, R. B., N. A. Alexandridis and A. Avižienis, "Design of a Fault-Tolerant Modular Computer with Dynamic Redundancy," *AFIPS Conference Proceedings*, Vol. 41, (Fall JCC 1972), pp. 1057-1067.
17. Gilley, G. C., "A Fault-Tolerant Spacecraft," *Digest of 1972 International Symposium on Fault-Tolerant Computing*, June 1972, pp. 105-109.
18. Maison, F. P., "The MECRA: A Self-Repairable Computer for Highly Reliable Process," *IEEE Transactions on Computers*, Vol. C-20, No. 11, November 1971, pp. 1382-1393.
19. Everett, R. R., C. A. Zraket and H. D. Benington, "SAGE—A Data-Processing System for Air Defense," *Proceedings of the Eastern Joint Computer Conference*, Washington, D.C., December 1957, pp. 148-155.
20. Ornstein, S. M., et al. "Pluribus—A Reliable Multiprocessor," *AFIPS Conference Proceedings*, Vol. 44, 1975 NCC, pp. 551-559.
21. Blaauw, G. A., "The Structure of SYSTEM/360: Part V—Multisystem Organization," *IBM System Journal*, Vol. 3, No. 2, 1964, pp. 181-195.
22. *A Guide to the IBM System/360 Model 145*, IBM Corporation, Technical Publications Department, White Plains, New York, Third Edition, August 1972.
23. Corbato, F. J., J H. Saltzer and C. T. Clingen, "Multics—The First Seven Years," *AFIPS Conference Proceedings*, Vol. 40 (Spring JCC 1972), pp. 571-583.
24. Borgerson, B. R., "Dynamic Confirmation of System Integrity," *AFIPS Conference Proceedings*, Vol. 41, Part 1, 1972, pp. 89-96.
25. Wensley, J. H., "SIFT—Software Implemented Fault Tolerance," *AFIPS Conference Proceedings*, Vol. 41, Part 1, 1972, pp. 243-254.
26. Levy, H. O. and R. B. Conn, "A Simulation Program for Reliability Prediction of Fault Tolerant Systems," *Digest of the Fifth International Symposium on Fault-Tolerant Computing*, Paris, France, June 1975, pp. 104-109.
27. Hopkins, A. L., Jr. and T. B. Smith III, "The Architectural Elements of A Symmetric Fault-Tolerant Multiprocessor," *IEEE Transactions on Computers*, Vol. C-24, No. 5, May 1975, pp. 498-505.
28. Tang, D. T. and R. T. Chien, "Coding for Error Control," *IBM Systems Journal*, Vol. 8, No. 1, 1969, pp. 48-86.
29. Parhami, B. and A. Avižienis, "A Study of Fault-Tolerance Techniques for Associative Processors," *AFIPS Conference Proceedings*, Vol. 43, 1974, pp. 643-652.
30. Avižienis, A., "Arithmetic Error Codes: Cost and Effectiveness Studies for Application In Digital System Design," *IEEE Transactions on Computers*, Vol. C-20, No. 11, November 1971, pp. 1322-1331.
31. Bouricius, W. G., W. C. Carter and P. R. Schneider, "Reliability Modeling Techniques for Self-Repairing Computer Systems," *Proceedings of the 24th National Conference of ACM*, 1969, pp. 295-383.
32. Barlow, R. W. and F. Proschan, *Mathematical Theory of Reliability*, Wiley and Sons, 1965.
33. Griesmer, J. E., R. E. Miller, J. P. Roth, "The Design of Digital Circuits to Eliminate Catastrophic Failures," *Redundancy Techniques for Computing Systems*, Spartan Press, Inc., Washington, D.C., 1962, pp. 328-348.
34. Bricker, J. L., "A Unified Method for Analyzing Mission Reliability for Fault-Tolerant Computer Systems," *IEEE Transactions on Reliability*, Vol. R-22, No. 2, June 1973, pp. 72-77.
35. Ng, Y. W. and A. Avižienis, "A Unifying Reliability Model for Closed Fault-Tolerant Systems," *Digest of the Fifth International Symposium on Fault-Tolerant Computing*, Paris, France, June 1975, p. 224; full text to appear in IEEE Transactions on Computers.
36. Arnold, T. F., "The Concept of Coverage and Its Effect on the Reliability Model of a Repairable System," *IEEE Transactions on Computers*, Vol. C-22, No. 3, March 1973, pp. 251-254.
37. Merryman, P. M. and A. Avižienis, "Modeling Transient Faults in TMR Computer Systems," *Proceedings 1975 Annual Reliability and Maintainability Symposium*, Washington, D.C., January 1975, pp. 333-339.
38. Ng, Y. W., "Modeling and Analysis of Fault-Tolerant Computers," Ph.D. Dissertation, UCLA, Computer Science Department, University of California, Los Angeles, 1976.
39. Hardie, F. H. and R. S. Suhocki, "Design and Use of Fault Simulation for Saturn Computer Design," *IEEE Transactions on Computers*, Vol. EC-16, August 1967, pp. 412-429.
40. Chang, H. Y., G. W. Smith, Jr. and R. B. Walford, "LAMP: System Description," *The Bell System Technical Journal*, Vol. 53, No. 8, October 1974, pp. 1431-1449.
41. Avižienis, A., "Fault-Tolerance and Fault-Intolerance: Complementary Approaches to Reliable Computing," *Proceedings of the 1975 International Conference on Reliable Software*, Los Angeles, California April 1975, pp. 458-464.
42. Shooman, M. L., "Operational Testing and Software Reliability Estimation During Program Development," *Proceedings of the 1973 IEEE Symposium on Computer Software Reliability, New York City*, 1973, pp. 51-56.
43. Moranda, P. B., "Prediction of Software Reliability during Debugging," *Proceedings of the 1975 Annual Reliability and Maintainability Symposium*, January 1975, pp. 327-332.
44. Elmendorf, W. R., "Fault-Tolerant Programming," *Digest of the 1972 International Symposium on Fault-Tolerant Computing*, IEEE Computer Society, 1972, pp. 79-83.
45. Randell, B., "System Structure for Software Fault Tolerance," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2, June 1975, pp. 220-232.