Fault-Tolerant Computing: Fundamental Concepts

Victor P. Nelson
Auburn University

Fault tolerance is crucial in military and aerospace computing, and desirable in other applications. This review discusses basic concepts, mechanisms, and strategies and sketches future directions.

Digital systems have been entrusted with increasingly more critical responsibilities, requiring high dependability. Often the use of high-quality components and design techniques does not sufficiently reduce the likelihood of system failures, and means must be provided to tolerate faults in the system.

This article reviews the basic concepts of fault-tolerant computing, focusing on hardware. It examines failures, faults, and errors in digital systems and defines measures of dependability, which dictate and evaluate fault-tolerance strategies for different classes of applications. The various mechanisms for implementing a fault-tolerance strategy are reviewed, including error detection, fault masking, fault confinement, system reconfiguration and repair, and system recovery.

Failures, faults, and errors

When applied to digital systems, the terms failure, fault, and error have different meanings.[1,2] Failure denotes an element's inability to perform its designed function because of errors in the element or its environment, which in turn are caused by various faults.

A fault is an anomalous physical condition. Causes include design errors, such as mistakes in system specification or implementation; manufacturing problems; damage, fatigue, or other deterioration; and external disturbances, such as harsh environmental conditions, electromagnetic interference, ionizing radiation, unanticipated inputs, or system misuse. Faults resulting from design errors and external factors are especially difficult to model and protect against, because their occurrences and effects are hard to predict.

An error is a manifestation of a fault in a system, in which the logical state of an element differs from its intended value. A fault in a system does not necessarily result in an error. An error occurs only when a fault is "sensitized"; in other words, for a particular system state and input excitation, an incorrect next state and/or output results. A fault is referred to as latent if it has not yet been sensitized in the system. The term soft is often applied to errors that persist after the originating fault disappears. Once corrected, soft errors usually leave no damage in the system.

Hierarchical models of faults and errors. Device testing and fault-tolerant design require fault and error modeling at one or more levels of design abstraction, with various trade-offs between accuracy and ease of modeling and analysis. At the lowest level, faults are technology dependent. Such physical defects as shorts or opens in metal or polysilicon signal lines can alter voltages, switching times, and other properties.[3] External disturbances also work at this level, affecting signal lines, charge storage, and other properties.

At the logical level, a digital system is modeled with gates and memory elements, with all signals represented as binary values. Low-level fault-tolerance strategies are designed to detect or mask faults that produce erroneous logical values. Because of its simplicity, the "stuck-at" model is the most widely used logical fault model, assuming that a fault manifests itself as a fixed logical value on a signal line. A more complex model is the "bridging" fault, in which coupling between signal lines results in the logical value of one line affecting the value of another. Other complex faults alter the basic logical function of a gate, as often happens in programmable logic arrays, where the presence or absence of connections in an AND/OR array results in implicants being added to or removed from a function.

At higher levels of abstraction (registers, arithmetic logic units, processors, etc.) faults typically appear as changes in the module's behavior, as represented by its truth table or state table. At this level fault modeling is usually more abstract to facilitate simulation at the behavioral level; hence, accuracy is often sacrificed.

Fault properties. A fault can be classified by its duration, nature, and extent. The duration of a fault can be transient, intermittent, or permanent. A transient fault, often the result of external disturbances, exists for a finite length of time and is nonrecurring. A system with an intermittent fault oscillates between faulty and fault-free operation. Usually, an intermittent fault results from marginal or unstable device operation. Permanent or "hard" faults are device conditions that do not correct with time. They result from component failures, physical damage, or design errors. Transient and intermittent faults typically occur with greater frequency than permanent faults and are more difficult to detect, since they may disappear after producing errors.

The nature of a fault is determined by its behavior in the system. A logical fault produces errors that can be represented as logical values, while errors resulting from indeterminate faults do not have logical equivalents. For example, the shorting of a logic gate input to ground can be modeled as a stuck-at-0 fault at that input. However, the behavior of a gate input whose signal voltage floats between the logic 1 and 0 thresholds cannot be represented as a simple logical value. Other indeterminate faults affect propagation times and other electrical parameters, making them difficult to model.

The extent of a fault is determined by the area affected at the level of abstraction being considered: Local faults affect single components, and global faults affect multiple components. Because of cost constraints, many fault-tolerance and device-testing strategies address only single, statistically independent faults. Multiple faults require more extensive fault models and global approaches to fault tolerance. However, multiple faults become more likely at increased very large scale integration levels. In addition, external disturbances tend to have global effects, especially in military and aerospace applications subject to electromagnetic interference and ionized-particle radiation. Hence, multiple faults are receiving increasing attention.

A fault-tolerant system is not necessarily highly dependable, nor does high dependability necessarily require fault tolerance.

Evaluating dependability and fault tolerance

The goal of fault-tolerant design is to improve dependability[2] by enabling a system to perform its intended function in the presence of a given number of faults. Note, however, that a fault-tolerant system is not necessarily highly dependable, nor does high dependability necessarily require fault tolerance.

Dependability can be quantified by deterministic or probabilistic measures. A deterministic goal for a fault-tolerant system might be that no single fault can cause system failure. Many commercial system manufacturers advertise their systems' ability to tolerate some maximum number of processor, disk drive, and other component failures. However, such advertising does not mention the frequency or likelihood of such failures, or their cost.

Reliability and availability. Dependability is most often quantified probabilistically in terms of either reliability or availability. Reliability, $R(t)$, is the conditional probability that a system can perform its designed function at time t, given that it was operational at time $t = 0$. Thus $R(t)$ is a function of the fault processes affecting the system, and of any mechanisms that prevent system failure when a fault occurs. Many real-time systems, such as those used for aircraft or nuclear power plant control, require a high $R(t)$ because a single error could be fatal. For long-life unattended systems, such as those used in deep-space probes, the probability of multiple faults increases dramatically with mission time. Automatic repairs must be made with spare resources to maintain reliability over the life of the mission, although some performance degradation may be acceptable during these repairs.

Where cost prohibits sufficient fault tolerance to ensure continuous error-free operation, some amount of downtime for repair is inevitable. Availability, $A(t)$, is a useful measure for systems subject to failure and repair; it is defined as the probability that a system is operational at time t. Availability is often expressed as a steady-state value, either as the probability that the system is operational at any random time, or as a given amount of downtime over a specified interval. For example, the availability goal for the Bell System electronic switching system was specified as two minutes of downtime per year.[4] Commercial systems, which must be affordable as well as dependable, are normally designed for high availability. They use fault-tolerant protocols and other operations to protect the database from contamination, while using redundant processors and other resources for diagnosis and repair. Some systems can continue operating at a degraded level during repair.

Statistical mean values of system failure and repair times are often used in system evaluation. However, they can be misleading, since they are computed over infinite time intervals rather than the relatively short lifetime of the evaluated system. The two most common parameters are "mean time to failure" (MTTF), which is the expectation of the time at which the system will fail, and "mean time to repair" (MTTR), the expectation of the time to restore a failed system to correct operation. These two parameters are most often used to compute steady-state availability, given by

$$A_{\text{steady-state}} = MTTF/MTTF + MTTR \quad (1)$$

If a system is highly reliable — that is, if *MTTF* is large relative to *MTTR* — then availability is close to 1. For smaller *MTTF* values, availability varies significantly with repair time. Complete derivations of the above parameters and other reliability and availability measures are discussed elsewhere.[1] K.S. Trivedi discusses system reliability modeling in this issue of *Computer*.[5]

Improving reliability with fault tolerance. The effects of a fault-tolerant design strategy on system reliability can be expressed as follows:

$$R_{\text{system}} = Pr\{\text{no fault}\} + Pr\{\text{correct operation/fault}\} * Pr\{\text{fault}\} \quad (2)$$

The first term is the probability that no fault will occur. It is maximized by "fault-intolerant" design, that is, by high-quality components, proofs of design correctness, and other formal design methodologies. If $Pr\{\text{no fault}\}$ can be made sufficiently high, a target system reliability can be achieved without fault-tolerance strategies.

The effects of fault tolerance on reliability are represented by the second term in Equation 2, which is the probability that a fault will occur but will not result in system failure, computed over all possible faults. $Pr\{\text{correct operation/fault}\}$, referred to as the coverage of the fault-tolerance mechanism, is the conditional probability that a system will continue to operate correctly given the occurrence of a particular fault. Each coverage term is weighted by the probability that the corresponding fault will occur, so for a cost-effective system design, fault-tolerance mechanisms should be targeted at the most likely faults. Note that if fault probabilities are high, a system may be able to tolerate all of a given set of faults and yet not be sufficiently reliable for the application. Automatic fault-detection, diagnosis, repair, and recovery mechanisms can reduce or eliminate downtime, improving availability.

Fault tolerance in a digital system is achieved through redundancy in hardware, software, information, and/or computations.

A fault-tolerant-system designer must also consider performance, complexity, cost, size, and other constraints, all of which are affected by the redundancy and fault-tolerance strategies used. These costs must be weighed against such consequences of system failure as lost production or danger to life, which may be difficult to quantify.

Elements of fault-tolerance strategies

Fault tolerance in a digital system is achieved through redundancy in hardware, software, information, and/or computations. Such redundancy can be implemented in static, dynamic, or hybrid configurations. A fault-tolerance strategy includes one or more of the following elements:

- *Masking.* Dynamic correction of generated errors.
- *Detection.* Detection of an error — a symptom of a fault.
- *Containment.* Prevention of error propagation across defined boundaries.
- *Diagnosis.* Identification of the faulty module responsible for a detected error.
- *Repair/reconfiguration.* Elimination or replacement of a faulty component, or a mechanism for bypassing it.
- *Recovery.* Correction of the system to a state acceptable for continued operation.

For short-term ultrareliable operation, where no time is available for off-line fault diagnosis and repair, a static or passive configuration of elements is designed to mask a given maximum number of faults. Dynamic redundancy, on the other hand, involves the switching of modules or rerouting of communications as faults occur. The faulty components are detected, diagnosed, and repaired or replaced.

In a hybrid approach a static base configuration masks a given number of faults, while faulty modules are detected and replaced within the configuration. Hybrid redundancy is desirable for long-term ultrareliable applications in which the probability of multiple faults is high.

High-availability applications do not necessarily require continuous error-free operation, although databases and other critical resources must be protected. In such cases, errors are detected and contained within replaceable modules, rather than masked. System operation is then degraded or halted to perform diagnosis, reconfiguration or repair, and recovery.

Error detection, masking, and correction. Component complexity affects the ability to distinguish errors from correct values. Errors occurring in data-storage components, such as registers and memory, or during data transmission via buses or network links, are more easily detected than errors originating within modules that generate or transform data. Masking or correcting errors is more difficult, requiring multiple copies of an element or other redundancy so that correct data can be extracted from the redundant information. Error detection and correction can be concurrent with normal system operations or executed off line during specified testing intervals.

Error detection and correction codes. Coding theory is the most widely developed mechanism for error detection and correction in digital systems, typically requiring less redundancy than other error detection and correction schemes. A code's error detection and correction properties are based on its ability to partition a set of 2^n n-bit words into a code space of 2^m words and a noncode space of $2^n - 2^m$ words. For most codes, each word comprises m bits of information and $k = n - m$ check bits. Each code is designed so that a given number of errors transforms a code-space word into a word in the noncode space. Errors are detected by decoding circuits that identify any word outside the code space. Error correction is performed by more extensive decoding that uniquely associates a noncode-space word with the original code word transformed by the errors.

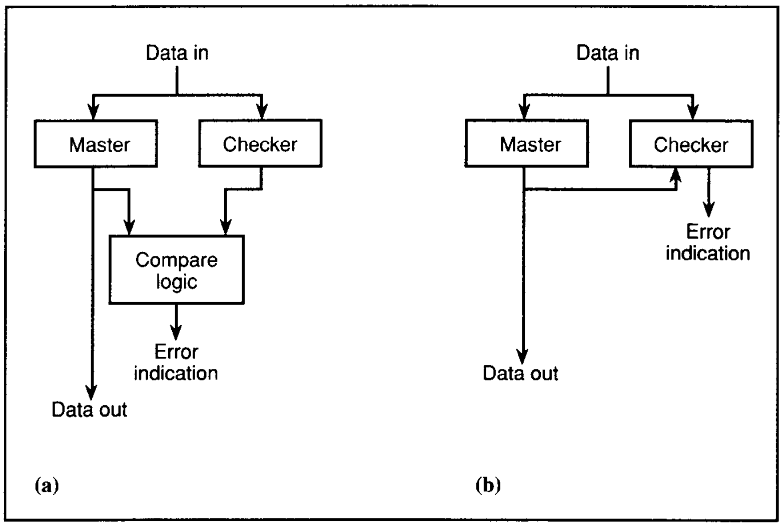

Figure 1. Replicated lockstep operation of modules with redundant outputs checked in each clock cycle: (a) logic compared externally; (b) logic compared on chip.

Within a single word, the number of errors detectable or correctable by a given code is related to the minimum separation or Hamming distance between the words of the code space. The distance is the minimum number of bit positions by which any two words from the code space differ. If two words differ by only one bit position, then an error in that bit transforms one word into the other. If the minimum distance is 2, a single error can produce only a noncode word, with at least two errors required to transform one code word into another. If the minimum separation is 3, any noncode word produced by a single error is distance 1 from the original code word and at least 2 from any other code word, allowing the original word to be uniquely identified.

Larger separations permit detection and/or correction of greater numbers of errors, generally by increasing the size of the noncode space (2^{m+k}) relative to that of the code space (2^m), making it more likely that errors will result in noncode words. The cost of this increased coverage is usually a lower code efficiency (code bits versus total bits) or a more complex encoding algorithm.

Error detection and correction codes vary widely in detection and correction properties, encoding and decoding complexity, and code efficiency. The most common codes include simple parity checks to detect errors in buses, memory, and registers. Parity-based Hamming codes detect and correct errors in memory; cyclic redundancy checks and other cyclic codes detect and correct errors in communications channels and disk storage; *m*-out-of-*n* codes detect errors in microprogram control stores and other ROMs; and arithmetic codes detect errors originating within arithmetic logic units.

Many computer memory subsystems include single-error correction and double-error detection using inexpensive Hamming-code-based support chips that efficiently encode and decode words during memory operations. Other commercial very large scale integration components include parity generators for buses and storage elements, and encoding/decoding circuits for disk drives, tapes, networks, and other communications channels. Some new VLSI components incorporate on-chip parity generation and checking logic; for example, the Advanced Micro Devices Am29300 chip set generates and checks parity on data paths to and from the device, and on internal data paths. In addition, several recent VLSI memories incorporate on-chip error detection and correction to mask memory cell faults arising in manufacturing or normal operation.

Self-checking logic. Self-checking logic designs detect faulty logic circuits,[6] especially in code checkers and other circuits that could be single points of failure in a system.[4] (Several experimental VLSI designs have been implemented entirely with self-checking circuits.) Each self-checking circuit has coded inputs and outputs, typically in the form of 2-bit "dual-rail" logic, which has two valid code words and two noncode words for each logic line. A circuit is classified as fault secure if, for any specified fault within the circuit, the circuit never produces an incorrect output code word when stimulated by a correct input code word. A self-testing circuit, on the other hand, outputs a noncode word for at least one code word input for each possible fault. A totally self-checking circuit has properties of both fault-secure and self-testing circuits; hence, no internal fault can convert an erroneous input into a valid output, and at least one normally occurring input will detect each possible internal fault.

Module replication for error detection and masking. With circuits that generate or transform information, complete module replication is often the only cost-effective approach for error detection and correction. Figure 1 shows the most straightforward approach to error detection: The outputs of identical modules operating in lockstep are compared. Several commercial transaction-processing systems have been built around pairs of off-the-shelf microprocessors with comparator circuits at their bus interfaces to detect processor faults (Figure 1a).

Simple disagreement detection indicates a fault but cannot identify the faulty unit. The system must be interrupted for further diagnosis. Continuous operation can be attained by using additional error-detection mechanisms to make the duplicated modules self-checking, as in the AT&T 3A electronic switching system processor, which uses self-checking logic circuits.[4] Figure 2a shows that when one module signals an error, it can be disabled while the other module continues to supply correct information, effectively masking the fault in the failed unit. Normally the disagreement detector between modules is eliminated and all errors are assumed to be detected within the redundant modules. Figure 2b shows how self-checking modules can be built with off-the-shelf components: One of the configurations of Figure 1 is duplicated, so four units and two comparators are needed for continuous fault masking. This approach has been used in the Stratus computer family and other systems.

Continuous operation is often provided

by using the majority vote of the outputs of three or more identical modules, masking failures of the minority. Triple modular redundancy has been used extensively in ultrareliable systems for aerospace and industrial applications, with two out of three votes masking single-module failures. Additional fault coverage can be attained with N modules by deploying them in a hybrid modular-redundant configuration, in which failed modules are replaced within a triple modular-redundant core configuration. Hybrid modular-redundant configurations can mask failures of all but two modules, compared with a simple minority in M-out-of-N majority-voting systems.

A significant problem with module replication is synchronization of the redundant modules. If comparison or voting is done in hardware, tight coupling of the redundant modules is needed to ensure that comparison or voting takes place on valid data samples. Fault-tolerant clocking schemes and other means of synchronization have been studied extensively, and several recent commercial VLSI chips include on-chip support for duplex, master/checker operation. Figure 1b shows paired master and checker chips operating in lockstep, with all corresponding pins connected to the same input/output lines. Both chips receive all inputs and perform all operations. The output lines are driven only by the master, with output also routed into the corresponding pins of the checker to on-chip comparators for comparison with values produced by the checker. The result is indicated by a match or an error signal.

An alternative to tight coupling is to compare only selected outputs from loosely synchronized units. In the SIFT system,[7] critical-process outputs are exchanged by the redundant processors in each process step and compared in subsequent process steps by a software voter. In the space shuttle, selected data values are mathematically combined into "compare words," which are periodically exchanged and compared by software in four redundant processors.[8]

Voters and comparators, although typically much more reliable than the redundant modules they protect, represent potential single-failure points in replicated systems. Fault tolerance and reliability can be increased by replicating the comparators or voters, usually at the module inputs, as in the triple modular-redundant system stage of Figure 3. Failure of any single voter or the module to which its output is connected is masked by the voters at subsequent module inputs. Redundancy schemes have also been extended to many nondigital devices (motors, actuators, sensors) used in redundant systems to minimize the number of single-failure points.

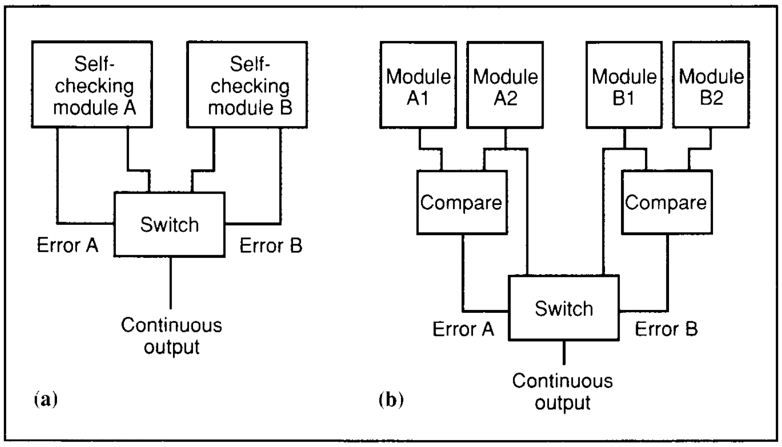

Figure 2. Continuous operation with duplex self-checking modules: (a) two self-checked modules; (b) four simple modules as two self-checked pairs.

Protocol and timing checks. The behavior of most sequential logic circuits and systems can be described by state machines or other protocols. Protocol variation resulting from a fault can be detected several ways without massive replication of modules.[9] Selected process states or module outputs can be compared with predicted values or other heuristic information, generated by alternative algorithms or off-line units. Data values can be checked for proper structure or consistency with previous or predicted values. Handshaking sequences between elements involved in data transfers can be monitored by hardware or software, especially over buses and network links. Operational "capabilities," — the activities allowed by various processes — can be verified before allowing an operation on a critical resource. Such approaches often reduce hardware redundancy requirements but may be more difficult to implement, requiring application-specific information

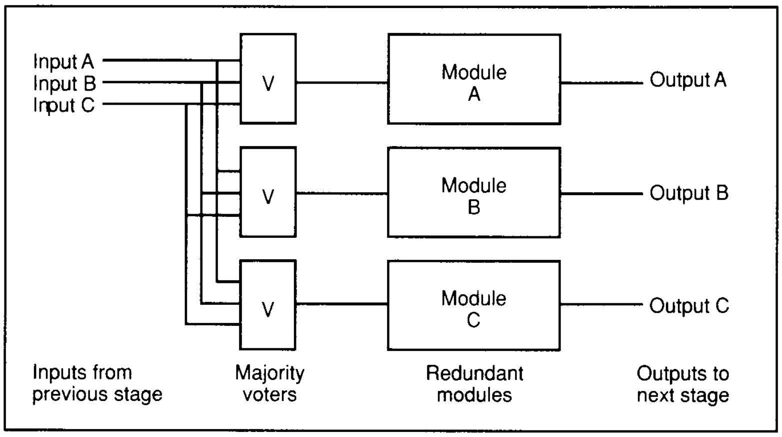

Figure 3. Triplicated voters and modules forming one triple modular-redundant stage of a system, with voting at module inputs.

which might, in turn, depend on unpredictable system inputs.

A simple fault-detection mechanism is the time-out check. An event failing to take place within some predefined time interval usually indicates a fault (an event can be a single data transfer or an entire process step). Such occurrences can be monitored by a "watchdog timer" set at the beginning of each event to time-out after some time T_{max}, interrupt the system, and signal an error. If the event completes before T_{max} has elapsed, the timer is stopped and reset for the next event.

Error correction without massive redundancy is difficult. However, for many transient faults, simple repetition of an operation after the fault disappears may produce correct results, provided the system state can be restored to the beginning of the operation. Many processors support single-instruction retry, with facilities to detect errors and save and restore register values. Several microprocessors also support bus-cycle retries, which can be performed with minimal saving of information. In both cases, hard faults are signaled if errors persist after some maximum number of retries.

Fault containment. To protect critical system resources and minimize recovery time, errors must be confined to the module in which they originate. Typically, error-containment boundaries are hierarchically defined, with errors confined at the lowest level to single replaceable or repairable modules, and additional boundaries set around subsystems containing these modules. Johnson's excellent case study of fault-containment boundary definition and support describes the establishment of containment boundaries around buses, processors, and memory modules in the former Intel iAPX-432 family.[10]

Containment boundaries can be established in two ways: Each module can check its own outputs, or each can check all incoming information. The most common approach is to require each module to suspect all incoming information and correct or block faulty data at the module interface. Voters in software[7] or hardware[11] are used in the logical configuration shown in Figure 3.

If a module is to be responsible for its own output, it needs an error-containment boundary. An error detection or correction circuit, such as a voter, a comparator, or a code checker, is placed at the interface between the module and the system bus or communications channel, along with a circuit capable of disabling the module's output. If error correction is not possible, a faulty module must be isolated to prevent error propagation; its process is effectively halted. A disadvantage in this configuration is that the module interface often cannot protect the system from failures of the interface circuits themselves.

Reconfiguration and repair. A system is repaired either by replacing the failed module with a spare or by reconfiguring the system structure or work load distribution to circumvent the module. Module replacement restores the system to full operation but requires redundant modules not used for normal operations.

Many reconfiguration strategies use all system components to perform useful work. When a fault occurs, system performance is degraded by redistributing the work load among the remaining resources. Or system redundancy can be reduced, affecting subsequent fault tolerance. The space shuttle computer complex is an example of the latter strategy. It uses four processors with majority voting for critical operations.[8] Voting continues after one failure, but a second failure ends voting and a single processor performs all remaining operations.

A failed module may be physically or logically removed from a system. Logical removal is accomplished by switching off the module's power, forcing its output into an inactive state, or instructing all units to ignore or bypass it.

Replacement units can be either "hot" or "cold." A hot spare concurrently performs the same operations as the module it is to replace, needing no initialization when it is switched into the system. A cold spare is either not powered or used for other tasks, requiring initialization when switched into the system. System designers must weigh the cost of unused spares against that of initialization time when deciding between hot or cold spares.

If a failed module is not replaced, system operation degrades as work is distributed among remaining resources. In multiprocessors and other parallel processing systems, tasks are typically distributed across the available processors, so that processor loss only reduces system throughput.[12] This happens in commercial transaction-processing multiprocessors advertised to operate continuously in the presence of faults. In these systems, all critical data is replicated or otherwise protected to facilitate transfer of operations between processors. Special care is taken to duplicate global data or provide other redundant information to allow corrupted data to be repaired. Global data usually resides in shared memory or in "mirrored" disk volumes — duplicated disk drives and controllers accessible by multiple processors. In massively parallel machines or cellular arrays with complex interconnection architectures, algorithms reassign tasks and reroute communications to bypass faulty processing cells for graceful degradation of system operation.[13]

System recovery. If an unmasked error has propagated through a system or if system hardware or software has been reconfigured, a recovery period is needed to correct the system. The elapsed time between the occurrence and the detection of an error determines the amount of damage and the length of the recovery period.

Most system-recovery schemes restore system operation to a previous correct state or recovery point. A processor is rolled back to a recovery point by restoring registers and memories to the saved state and invalidating cache memories, forcing cached data to be restored from global memory. Global data is typically protected through redundant protocols that allow updates to be completed or undone and repeated following a failure. In shared-memory multiprocessor systems,[12] global data and lists of tasks to be performed are kept in shared memory, allowing processors to continue automatically with tasks on the list as failed processors are disabled. This approach also helps balance loads on the individual processors.

In loosely coupled systems, spare processors are periodically updated at predefined checkpoints, so that when a spare is given control of a task after failure of a master processor, processing can continue from the most recent checkpoint rather than from the beginning of the task. The degree of rollback is limited by using atomic actions — small, indivisible processing steps completed and verified before global updates and the next action. Recovery from a failure occurring before saving the results is usually performed by repeating the entire action.

Computer architectures are changing rapidly, with increased integration in VLSI devices, new parallel processing architectures, and widely distributed networks presenting new challenges to fault-tolerant-design engineers. Much previous work in fault-tolerant-hardware design focused on gate-level approaches, but now more work is needed at much higher levels of abstraction, making complete design validation more difficult. Consequently, new approaches and tools must be developed for fault-tolerant design, simulation, and reliability analysis.

Large systolic arrays, massively parallel architectures, and other large-scale distributed systems with complex interconnection networks present challenges in system control, performance, and fault tolerance. Engineers working on communications structures and algorithms for mapping applications onto systolic arrays and other cellular parallel systems are also developing extensions to detect and diagnose faulty cells and circumvent them in real time.

Most of the fundamental concepts discussed here deal primarily with localized rather than system-wide fault tolerance. Localized strategies are easy to understand and apply. System-level fault tolerance requires considerable work, especially in wafer-scale systems and other highly integrated systems, which are subject to multiple component failures. System-level fault tolerance is also a challenge in distributed systems subject to synchronization problems and global upset, especially in aerospace, military, and other applications where external disturbances are likely. The challenge in commercial applications is to provide fault tolerance that is both dependable and affordable. ■

References

1. V.P. Nelson and B.D. Carroll, *Tutorial: Fault-Tolerant Computing*, CS Press, Los Alamitos, Calif., Order No. 677, 1986, Chaps. 1-2.
2. A. Avizienis and J.C. Laprie, "Dependable Computing: From Concepts to Design Diversity," *Proc. IEEE*, Vol. 74, No. 5, May 1986, pp. 629-638.
3. J.A. Abraham and W.K. Fuchs, "Fault and Error Models for VLSI," *Proc. IEEE*, Vol. 74, No. 5, May 1986, pp. 639-654.
4. W.N. Toy, "Fault-Tolerant Design of Local ESS Processors," *Proc. IEEE*, Vol. 66, No. 10, Oct. 1978, pp. 1,126-1,145.
5. K.S. Trivedi and R. Geist, "Reliability Estimation of Fault-Tolerant Systems: Tools and Techniques," *Computer*, this issue.
6. E.J. McCluskey, "Design Techniques for Testable Embedded Error Checkers," *Computer*, this issue.
7. J.H. Wensley et al., "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control," *Proc. IEEE*, Vol. 66, No. 10, Oct. 1978, pp. 1,240-1,255.
8. J.R. Sklaroff, "Redundancy Management Technique for Space Shuttle Computers," *IBM J. Research and Development*, Vol. 20, No. 1, Jan. 1976, pp. 20-28.
9. W.H. Kohler, "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems," *Computing Surveys*, Vol. 13, No. 2, June 1981, pp. 149-183.
10. D. Johnson, "The Intel 432: A VLSI Architecture for Fault-Tolerant Computer Systems," *Computer*, Vol. 17, No. 8, Aug. 1984, pp. 40-48.
11. A.L. Hopkins, Jr., et al., "FTMP—A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft," *Proc. IEEE*, Vol. 66, No. 10, Oct. 1978, pp. 1,221-1,239.
12. J.G. Kuhl and S.M. Reddy, "Fault-Tolerance Considerations in Large, Multiple-Processor Systems," *Computer*, Vol. 19, No. 3, Mar. 1986, pp. 56-67.
13. R. Negri, M. Sami, and R. Stefanelli, "Fault Tolerance Techniques for Array Structures Used in Supercomputing," *Computer*, Vol. 19, No. 2, Feb. 1986, pp. 78-87.

Victor P. Nelson is an associate professor in the Department of Electrical Engineering at Auburn University. He has been involved in fault-tolerant computing since 1976 in research projects with the US Air Force and the US Army Ballistic Missile Defense Advanced Technology Center. With the latter he was responsible for construction of the Fault Tolerance/Distributed Computing Laboratory at Auburn. Nelson has also worked as an industry consultant. His research interests include microprocessor applications, computer architecture, and computer-aided design of VLSI devices.

Nelson received a BSEE from the University of Kentucky and MS and PhD degrees from Ohio State University. He is a member of the IEEE Computer Society, ACM, Sigma Xi, and the National Society of Professional Engineers.

Readers may write to the author at Auburn University, Department of Electrical Engineering, 200 Broun Hall, Auburn, AL 36849-5201.