Theme Feature

Toward Systematic Design of Fault-Tolerant Systems

After 30 years of study and practice in fault tolerance, high-confidence computing remains a costly privilege of several critical applications. It is time to explore ways to deliver high-confidence computing to *all* users

Algirdas Avizienis
University of California, Los Angeles; Vytautas Magnus University

As computing and communications become irreplaceable tools of modern society, one fundamental principle emerges: The greater the benefits these systems bring to our well-being and quality of life, the greater the potential for harm when they fail to perform their functions or perform them incorrectly. Consider air, rail, and automobile traffic control; emergency response systems; airline flight controls; nuclear power plant safety systems; and most of all, our rapidly growing dependence on health care delivery via high-performance computing and communications. When these systems fail, lives and fortunes may be lost.

At the same time, threats to dependable operation are growing in scope and severity. Leftover design faults (bugs and glitches) cause system crashes during peak demand, resulting in service disruptions and financial losses. Complex systems suffer stability problems due to unforeseen interactions of overlapping fault events and mismatched defense mechanisms. Hackers and criminally minded individuals invade systems, causing disruptions, misuse, and damage. Accidents result in severed communication links, affecting entire regions. Finally, we face the possibility of systems damage by "info-terrorists."

Fault tolerance is our best guarantee that high-confidence systems will not betray the intentions of their builders and the trust of their users by succumbing to physical, design, or human-machine interaction faults, or by allowing viruses and malicious acts to disrupt essential services.

ORIGIN OF FAULT TOLERANCE

I originally formulated the concept of fault tolerance in 1967: "We say that a system is fault-tolerant if its programs can be properly executed despite the occurrence of logic faults."[1] The fault tolerance concept resulted from three converging developments. First, the earliest use of computers made it apparent that even with careful design and good components, physical defects and design errors were unavoidable. Thus, designers of early computers used practical techniques to increase reliability: They used redundant structures to mask failed components; error-control codes and duplication or triplication with voting to detect or correct information errors; diagnostic techniques to locate failed components; and automatic switchovers to replace failed subsystems.[2]

Second, in parallel with these evolving engineering techniques, computing pioneers such as John von Neumann[3] and Edward F. Moore and Claude E. Shannon[4] addressed the general problem of building reliable systems from unreliable components. William H. Pierce unified their theories of masking redundancy and incorporated some others.[5]

Third, in 1958 NASA challenged Caltech's Jet Propulsion Laboratory to build unmanned spacecraft for interplanetary exploration. These missions would last 10 years or more and require onboard computing. The task of designing computers that could survive a journey of several years and then deliver peak performance deep in space was unprecedented. Existing studies indicated that providing a large number of spare subsystems promised longevity, if the spares could be employed in sequence. JPL's problem was to translate this idealized "spare replacement" system model into a flight-worthy implementation of a spacecraft guidance and control computer.

The proposal to do this, "A Self-Testing-and-Repairing System for Spacecraft Guidance and Control" (STAR), was presented in October 1961.[2] With the support of JPL and NASA, eight years later the effort (led by myself and five researchers: George

0018-9162/97/$10.00 © 1997 IEEE

The possible causes of faults include natural phenomena of internal or external origin, and accidental or intentional human actions.

C. Gilley, Francis P. Mathur, David A. Rennels, John A. Rohr, and David K. Rubin) culminated in a laboratory model of the JPL-STAR computer.[6] US Patent No. 3,517,671 granted to me in 1970 and assigned to NASA, validated the concept's originality. A flight model of the JPL-STAR was designed for a 10- to 15-year space mission,[6] but construction stopped when NASA discontinued the Grand Tour mission.

Nevertheless, the project did afford the study of all accessible engineering solutions and theory concerning reliability. From a confusing variety of theories and techniques[2-5] emerged the unifying concept of fault tolerance.[1] And in 1971 JPL and the IEEE Computer Society cosponsored the first International Symposium on Fault-Tolerant Computing.

During the succeeding 25 years the set of faults that fault-tolerant systems had to tolerate grew extensively. The original concept dealt with transient and permanent logic faults of physical origin. The increasing complexity of software and VLSI chip logic emphasized the impossibility of removing all design faults prior to operation. Thus, faults due to human design errors were added to the demands of fault tolerance. Experience also led to the addition of *interaction* faults—those faults inadvertently introduced by humans during computer operation or maintenance. Finally, consequences of malicious actions intended to alter or interrupt service were recognized as *intentional* design faults. This concept established a common ground for the unified treatment of security and fault tolerance concerns in system design. Assuring full compatibility and integration of security and fault tolerance techniques is a major challenge for contemporary designers.

The possible causes of faults, then, include natural phenomena of internal or external origin and accidental or intentional human actions. In either case, faults will cause errors. If error detection and recovery do not take place in a timely manner, a failure will occur that will be manifested by the denial or an undesirable change of service.

There are two ways to build a fault-tolerant system. The *bottom-up* approach entails designing an infrastructure of autonomously fault-tolerant subsystems (microprocessors, memories, sensors, displays, and so on) and integrating this infrastructure with global fault tolerance functions such as reconfiguration and externally supported recovery. The *top-down* approach allows a system to be built using existing (off-the-shelf) subsystems that may have little or no fault tolerance at all. A global monitoring function is then implemented to provide fault tolerance. Top-down design is the prevailing practice.

An examination of both approaches clearly makes a case for the long-range merits of the bottom-up technique. Moreover, similarities between fault tolerance and the human immune system suggest an analogy that offers developers and users of high-confidence systems a greater understanding and acceptance of bottom-up fault tolerance.

DESIGN PARADIGM FOR FAULT-TOLERANT SYSTEMS

Building the JPL-STAR computer involved much improvisation and experimentation with design alternatives. It became apparent that the lessons learned could serve as the foundation for a more orderly approach that would guide designers in treating fault tolerance as a systematic issue.

My 1967 paper[1] was followed by publications over the next 25 years that formulated an evolving view of how to systematically introduce fault tolerance in hardware, software, communication, and man-machine-interface subsystems and how to integrate these elements during design.[2] The appearance of new, successful fault-tolerant systems offered additional design insights and more operational experience.

Here I summarize the most mature version of the guidelines for bottom-up fault tolerance. An abstraction of observed design processes in which steps often overlap, it is offered as a way to minimize the probability of oversights, mistakes, and inconsistencies that may occur during the implementation of fault tolerance. The first three steps—specification, implementation, and evaluation—deal with the building of a new system. Implementation and evaluation are concurrent. Step four—modification—addresses the repair or augmentation of an existing design.

Specification

System requirements that describe the services to be delivered in terms of functionality and performance are the starting point in specifying fault tolerance.

Mission phases. First we identify different phases, characterized by different environments and operation conditions during the system's lifetime or mission.

Dependability conditions. Then we identify four dependability conditions for each phase:

- *Fault classes*. I have already summarized the potential fault classes to be tolerated. Environmental conditions and operating interfaces will affect the expected fault classes. Each mission phase may encounter *internal* faults that arise within the system and *external* faults that introduce errors into the system via the interfaces: I/O links, external fault tolerance support, and human-machine interaction.
- *External support*. Here we determine the availability of repair by external agents and the form of this support (on-demand or periodic) as well as

the availability of remote support for fault tolerance functions (for example, external diagnosis and software reloading).
- *Service modes*. Here we determine the acceptability of different modes of service (full, reduced, degraded, emergency, safe shutdown, and so on) for each phase and establish each mode's required service level.
- *Dependability goals*. Here we determine each phase's requirements for system availability, reliability, maintainability, and safety, as well as potentially varying criticality of service delivery and system security goals during different phases.

Evaluation methods. Now we choose how we are going to evaluate the likelihood that the design will meet the dependability goals. We must also agree upon and specify independent means of validating the evaluation and define fault- and error-occurrence scenarios to enable evaluation. For example, how will the system respond to two or more independent, near-coincident faults with overlapping effects? How will it respond to latent errors?

Resource allocation. Finally, since the resources for making a system dependable are limited, we must decide before implementation begins how much to spend on fault tolerance versus fault avoidance during implementation.

Implementation

Fault tolerance is implemented through the sequence of system partitioning, subsystem design, and systemwide integration.

System partitioning. System partitioning is the definition of system structure, expressed by the interconnection of its building blocks and communication links. Functionality, performance, and fault tolerance requirements, available technologies, and past experience all affect the choice of the hardware, software, communication, and interface (I/O) subsystems that constitute the system being designed. For fault tolerance, partitioning defines potentially replaceable units and their *error containment boundaries*, which are intended to ensure failure independence and the absence of erroneous outputs for individual subsystems. Partitioning involves three major decisions:

- *Fault tolerance hierarchy*. Here we define a systemwide strategy for coordinated error detection and recovery. We may define detection and recovery functions for an individual subsystem (local), to be shared by a set of identical subsystems or extended over a group of diverse subsystems (intermediate), or to be systemwide (global). A hierarchy of three or all four approaches may be used in larger systems. At this time we define specialized support subsystems for fault tolerance (for example, service processors, software monitors, and dedicated communication links) and add them to the system structure.
- *Redundancy methods*. We can incorporate redundant subsystems into a design using multiple-channel computation (duplex, triplex, and so on), spare subsystems, or degradation by exclusion of failed subsystems. Or we can combine these approaches (for example, hybrid redundancy). Time redundancy (repetition of computations) is another option.
- *Design diversity*. Now we must decide if design diversity[7] (diverse hardware and software in each channel) is to be implemented. Design diversity is an effective way to ensure safety in critical applications and to protect the most vital subsystems of complex systems and networks against design faults.

Once these decisions have been made, every subsystem is characterized by its own error-containment requirements and its participation in set, group, and/or global detection and recovery functions is determined. Then we can apportion the system-level goals for availability, reliability, maintainability, safety, and security among the sets of subsystems that constitute the system.

Subsystem design. In a subsystem, fault tolerance is implemented in two parts: local error detection and recovery and support for higher level (intermediate, global) error detection and recovery. The adequacy of the chosen local detection and recovery methods is assessed by evaluations that occur during subsystem design and often lead to modification of initial choices. Both parts of the fault tolerance implementation entail the following steps:

- *Error detection*. Errors are undesired states caused by active faults and may lead to subsystem failure. Error detection is either concurrent with delivery of service or preemptive, in which case service delivery is suspended. Error-detecting codes are an example of the former method; BIST (built-in self-test) exemplifies the latter. Error detection methods must be able to detect errors produced by faults in the fault tolerance mechanisms themselves. Furthermore, they must be able to check spare subsystems for dormant faults and detect latent errors in stored information that is not subject to other forms of error detection.
- *Recovery*. An error signal invokes a recovery procedure, which attempts to restore the system to a valid state. Recovery is either backward—

How will the system respond to two or more independent, near-coincident faults with overlapping effects?

A systematically designed infrastructure is autonomous: It does not depend on other parts of the system for support.

returning the system to a previous, error-free state—or forward—constructing a valid, error-free new state from existing (usually redundant) information. A recovery sequence includes fault diagnosis and removal, error elimination, state restoration, and recovery validation. When diagnosis identifies a permanently faulty subsystem, fault removal is performed by either substituting a good spare subsystem or reconfiguring the system to function without the faulty subsystem. Error elimination and state restoration complete the recovery. Independent validation of successful recovery is desirable for every subsystem. Two special cases of recovery are error correction, which allows a subsystem (for example, memory) to continue with a permanent fault, and masking redundancy (for example, triple modular redundancy with voting), which masks a fault's presence without further recovery action.

Systemwide integration. The desired result of system partitioning and subsystem design is an integrated set of local, intermediate, and global fault tolerance functions that serve as a protective infrastructure to ensure the timely and correct delivery of system services.

A systematically designed infrastructure is autonomous; that is, it does not depend on other parts of the system (operating system, applications, personnel, and so on) for support. It is also distributed and fault tolerant itself. It has dedicated communication links and can also use the main links. And it is managed by a highly protected "hard core" subsystem (or a hierarchy of such subsystems) that executes global decisions to assure system recovery. These properties are analogous to those of the human immune system.

This phase has two major goals: to verify the infrastructure's completeness and consistency and to evaluate its ability to handle two or more nearly concurrent fault manifestations. To accomplish these goals, we need in-depth analysis and experimental fault injection, using the fault and error scenarios developed during specification.

Evaluation

Evaluation of fault tolerance is continuous during system partitioning, subsystem design, and system integration. At each step evaluation is an important design tool that facilitates the choices between fault tolerance techniques and assesses the likelihood of meeting the dependability goals. Successful completion of design requires a convincing verification of the design's completeness and its potential to meet the dependability goals. Verification consists of two distinct evaluations: first qualitative, then quantitative.

Qualitative evaluation generates deterministic predictions. It must be satisfied prior to quantitative evaluation, which generates probabilistic predictions. Otherwise, the evaluation may generate unreasonably optimistic predictions of availability and reliability because unjustified simplifications will go unnoticed.

Qualitative evaluation: Deterministic goals. The outcome of qualitative evaluation is a yes/no conclusion with respect to four goals:

- *Fault tolerance completeness and consistency*. This evaluation is part of systemwide integration. Here we use checklists of questions derived from the design paradigm, and experimental fault injection using worst-case scenarios.
- *Absolute tolerance*. This evaluates whether the system can survive one (or more than one) fault from a specified set and then execute a safe shutdown, usually stated as fail operational/fail safe. A detailed design analysis is needed to prove this property.
- *Absence of design faults*. Here formal and heuristic methods such as proof of correctness, testing, and experimentation are applicable.
- *System security goals*. To evaluate deterministic security requirements, we use the same tools used to evaluate the absence of design faults.

Quantitative evaluation: Probabilistic goals. This evaluation requires three steps:

- Describe the design using a system evaluation model that is characterized by sets of physical, structural, repair, fault tolerance, and performance parameters for every subsystem.
- Obtain coverage and execution-time parameters for all local, intermediate, and global detection and recovery functions. Fault injection experiments are essential for this task.
- Use the model to predict system reliability, availability, maintainability, and safety. The existence of multiple service modes necessitates the prediction of *mean time between mode reductions* and *duration of mode reduction* in suitable measures (mean, 99th percentile, and so on) instead of a single availability prediction.

Modification

An existing system is modified for repair—the removal of newly discovered faults—or for augmentation of functionality, performance, and/or fault tolerance. In both cases subsystems are modified or new subsystems are added. When this happens, it is essential to modify the specification first and then reimplement the subsystems with a complete reexamination and reevaluation of detection and recovery functions.

Failure to return to the specification may cause gaps in fault tolerance protection.

NASA experienced such a gap on April 10, 1981. A timely synchronization check was omitted after the addition of an alternate reentry program. As a result, the first flight of the US space shuttle program was aborted 19 minutes before launch.

OFF-THE-SHELF APPROACH

The bottom-up approach of the design paradigm results in fault-tolerant systems that are composed of an integrated set of fault-tolerant subsystems. However, development time and cost constraints often lead developers to use off-the-shelf subsystems—including microprocessors, operating systems, and applications—as building blocks in the design of systems that are expected to be highly dependable. OTS items usually have few fault tolerance functions—sometimes none at all.

Pentium Pro limitations

To illustrate the nature of fault tolerance functions in, for example, OTS microprocessors, let's consider the Intel Pentium Pro.[8] Compared with Sun UltraSparc II, MIPS 10000, HP PA-8000, DEC Alpha 21164, and IBM/Apple/Motorola PowerPC 620 microprocessors, the Pentium Pro appears to have the most complete set of fault tolerance functions among contemporary microprocessors.

An ancestor of the Pentium Pro, Intel's 486, provided parity checking for data bytes. The Pentium added address parity and introduced parity checks for cache, translation lookaside buffer, and microcode storage arrays; it also introduced a Machine Check Exception with address and type registers. In addition, the Pentium reintroduced the master/checker duplexing (functional redundancy checking) option that Intel pioneered in the 432 processor chips.

The Pentium Pro integrates five Pentium components into a single component. It retains all Pentium fault tolerance techniques, replacing data-byte parity with eight ECC (error-correcting code) bits for SEC/DED (single-error correction/double-error detection) operations. It uses two parity bits and provides a retry for the address bus, and it includes parity bits for two groups of control signals. The Machine Check Exception is generalized into a Machine Check Architecture with three global control registers and five banks of four error-reporting registers each.[8]

We can see that as chip complexity increases, more fault tolerance functions are added; however, major OTS drawbacks remain in the Pentium Pro:

- Protection by parity and ECC is limited to storage arrays and communication links, which are easy to check. The more complex data- and instruction-processing logic remains unchecked.
- The extensive system developer's documentation[8] (more than 1,400 pages) commingles error handling with all other information. There is no comprehensive top-down view of the fault tolerance techniques and their interrelationships. Management of most error conditions is relegated to a "central agent," which remains unexplained. Developers run a significant risk of overlooking or misinterpreting the details of error handling.
- Use of the Machine Check Architecture is optional and must be enabled by software. This leaves open the possibility of its accidental or malicious disabling during operation. The Machine Check Exception Handler software merely logs machine status and error information and then shuts down the system, since there are no on-chip recovery procedures to invoke.
- The master/checker duplexing is a throwaway solution at twice the cost of one microprocessor. Error detection is delayed until the error reaches the component's output; by then, shutdown, BIST, and restart is the only recovery option left, even if the cause was only a soft error that a local retry could eliminate.

Retrofit solutions

Systems built from OTS subsystems are very difficult to retrofit for fault tolerance. The absence of OTS hardware support for fault tolerance means that the only solution is to build a software monitor subsystem (such as the Pentium Pro's Machine Check Exception Handler) that resides and executes on the OTS hardware elements. A software monitor tries to check all subsystems for indications of failure and records abnormal symptoms. When necessary, it initiates shutdowns, BIST, and restarts. This approach has two weaknesses: The monitor software itself is unprotected because it resides and executes on an OTS processor, and it limits recovery handling to on/off.

A costly but effective method for building high-confidence systems with OTS subsystems is to employ multiple-channel computation with diverse hardware and software in each channel.[7] Variations of this design diversity approach have been used successfully in safety-critical systems, such as flight control and rail transportation, that use well-defined cyclic control algorithms.

However, cost and application complexity preclude this solution in most distributed, heterogeneous systems with OTS components. A potential retrofit solution is to implement a small, highly fault-tolerant hardware subsystem that monitors the system's operation, ensures data integrity, and manages recovery

Systems built from OTS subsystems are very difficult to retrofit for fault tolerance.

The complexity of modern systems is already comparable to that of some living organisms.

by switching in spare resources or reconfiguring the system. An excellent example of a hardware monitor is the IBM ES/9000 Type 9021's processor controller.[9]

A hardware monitor's effectiveness is limited entirely by the nature of the OTS elements being monitored; for example, a Machine Check Exception message from a Pentium Pro means that the processor must be shut down, even if the Machine Check Exception indicates only a soft error due to an SEU (single-event upset). More survivable OTS-based systems can be built only when the OTS subsystems themselves are fault tolerant.

AAS experience

The difficulties encountered in the OTS approach are illustrated by the Advanced Automation System (AAS) for air traffic control in the US,[10] in which fault tolerance was explicitly required as the means to assure very high availability. After a 42-month design competition, in 1988 the US Federal Aviation Administration awarded a $4.8-billion contract to IBM Federal Systems. The design used IBM RISC System/6000 processors as the basic subsystem, interconnected by a redundant token ring local communication network. Fault tolerance was attained by software-implemented Group and Global Service Availability Management.[11]

But in 1994 the FAA canceled the procurement of 23 large (up to 400 processors each) Area Control Computer Complex facilities and of the smaller Terminal AAS systems. Building was to continue on the less complex Tower CCC, and software for the Initial Sector Suite System, which had already cost more than $1 billion, was to be "analyzed to determine whether it can be operated and maintained." This failure of the world's most ambitious and costly attempt to build highly available systems using OTS processors and software-implemented fault tolerance attributes is disappointing—and potentially instructive. We can only hope that sufficient information on the successes and failures of the fault tolerance implementation will be made available soon to help builders of future systems.

HIGH-CONFIDENCE COMPUTING FOR ALL

The complexity of modern systems is already comparable to that of some living organisms. However, life forms and computing systems evolved with different priorities. For living organisms, survival of individuals and species came first; higher cognitive functions evolved gradually over billions of years, culminating in the emergence of Homo sapiens. Computers, on the other hand, were built to emulate human intellectual functions. Survival mechanisms, such as error detection and redundancy, came later. These mechanisms had to be introduced in early computers because vacuum tubes, relays, and other components frequently malfunctioned. A few years later, much more reliable transistors and magnetic cores were introduced, and second-generation hardware (IBM 7090, CDC 6600, and so on) dropped all checking except for some parity and error codes for storage tapes. This proved inadequate, however, and checking of general-purpose machines gradually returned with the third generation, culminating in some excellent contemporary designs, such as the IBM 9021.[9]

Another setback for high-confidence computing

The advent of the microprocessor has again caused a setback to the confidence level of general-purpose computing. The Pentium Pro falls short of the checking provided in the 1960s by the IBM System/360 and its contemporaries. The explosive growth in complexity, speed, and performance of single-chip processors has not led to proportional growth of on-chip error detection and recovery features.

Successful fault tolerance is evident in the control of high-speed transportation and in systems for dependable transaction processing. Yet on-chip checking of today's high-performance microprocessors is clearly in the pre–fault-tolerance age. At last year's Comdex, Intel's CEO Andy Grove predicted a chip with a billion transistors running at 10 gigahertz by the year 2011, but he did not mention the issue of high confidence.

Why do the most prevalent microprocessors remain low-confidence chips when high-confidence systems are flying our airliners and dispensing our cash? The answer has two parts:

- Reasonably high-confidence systems can be built from contemporary low-confidence chips when specific applications justify the high cost, and the presently small market does not motivate the production of fault-tolerant chips.
- Consumers and end users, who would benefit most from inexpensive high-confidence computing, generally are not aware of the advantages of making a commodity microprocessor a self-contained fault-tolerant system, so manufacturers aren't motivated to provide such chips.

Fault-tolerant microprocessors for all

High-confidence computing will be affordable for all users when every microprocessor is a self-contained fault-tolerant system as well as a building block for a fault-tolerant multichip system. To secure fault-tolerant microprocessors, we need to determine

- what technical know-how makes a microprocessor fault tolerant and

- what will motivate the major chip makers to build fault-tolerant chips.

This article has already answered the first question: We must treat a single chip as a self-contained system and apply the design paradigm to the chip's subsystems. At the same time, we must view the chip as a subsystem of a multichip system and design it to both provide and receive support for global fault tolerance functions at a higher level.

For example, a fault-tolerant Pentium Pro would remain functionally equivalent to the present implementation, except that it would invoke the external Exception Handler only in the rare cases when a fault has permanently damaged the chip or caused a large number of simultaneous errors. The chip would be able to handle common soft errors due to transient faults (such as SEUs) and participate in exception handling for other chips in the system. Further extensions, such as on-chip redundancy to handle permanent faults and design diversity to protect the chip's hard core against design faults, could be introduced.

The second question can be restated as, what would motivate Intel's Andy Grove to predict a tenfold (or maybe a hundredfold?) growth in the confidence we will be able to justifiably place in his billion-transistor, 10-gigahertz chips, compared with the Pentium Pro? To make this happen we must communicate the nature and the advantages of on-chip fault tolerance to the community of users who would benefit from high-confidence communications and computing. Customer demand is what will motivate microprocessor manufacturers to postpone the race for more functions, speed, and storage capacity in order to introduce autonomous chip-level fault tolerance. The resulting components will serve to build affordable high-confidence systems that do not rely on software alone to ensure the delivery of continuous and error-free service.

Challenge of a conceptual model

The human body's defenses—the immune system, the sense of pain, and the healing processes—could serve as a conceptual model for high-confidence systems in which fault tolerance is an integral attribute of every hardware element.

The body is analogous to hardware, and the cognitive processes supported by the body are analogous to software. Upon execution, software delivers the required high-confidence services for which the system is programmed.

Four fundamental attributes of the immune system[12] are particularly relevant:

- It functions continuously and autonomously, independent of cognition.
- Its elements (lymph nodes, other lymphoid organs, and lymphocytes) are distributed throughout the body to serve all of its organs.
- It has its own communication links—the network of lymphatic vessels.
- Its elements (organs and vessels) are themselves redundant and in some cases diverse.

These four attributes would also characterize a systematically designed fault-tolerant microprocessor or some other hardware element, as well as a network of such elements. Indeed, systemwide integration of fault tolerance is one attribute of the design paradigm.

The body's sense of pain is analogous to messages from the hardware fault tolerance infrastructure to the fault management software, which may take further action (just as the mind may decide to invoke higher level defenses such as medication or physical therapy). Most of today's computing subsystems lack sufficient local error detection and recovery attributes; thus, as in the Pentium Pro, external software must be involved in managing most of their fault conditions. This is like expecting cognition to compensate for the absence of immunity or a missing sense of pain. It's a risky substitution in a computing system, just as it would be in the human body.

The body's sophisticated healing processes lack a parallel in electronics. A rough analogy is with automatic repair by the use of spares or by switch-out of failed elements, at both on-chip and complete-chip levels.

In addition to in-body diversity like that found in the immune system, there is the attribute of diversity in a species that protects it against extinction due to genetic defects in individual members. Moreover, diversity among species ensures the continuity of life on Earth. Likewise, the use of hardware and software design diversity in fault-tolerant systems helps tolerate design faults in a chip's subsystems as well as at the complete-chip level.

How can the model help?

Is the proposed model likely to result in better fault-tolerant systems? There are three reasons to hope that it will. First, it sets up an analogy with the most dependable information processing systems in existence—the various species of living creatures that have survived and evolved over millions of years on our continually changing planet. Second, because it does not require knowledge of computer science or an a priori belief in the benefits of fault tolerance, the analogy should appeal to a wide spectrum of intelligent people and raise the expectation of affordable high-confidence computing for all. Third, the model will stimulate the imaginations of the coming generation

The body is analogous to hardware, and the cognitive processes supported by the body are analogous to software.

of system builders and lead to solutions that will far surpass today's best efforts.

This model may strike some as too ambitious a goal, but it will prove its usefulness if it helps counter the trend toward faster and ever more complex low-confidence chips.

The speed of computing will ultimately be limited by the laws of physics, but the demand for affordable high-confidence computing will continue as long as people use computers to enhance the quality of their lives. Eventually, one enterprising chip builder will deliver the first fault-tolerant microprocessor at a competitive price, and soon thereafter fault tolerance will be considered as indispensable to computers as immunity is to humans. The remaining manufacturers will follow suit or go the way of the dinosaurs. Once again, Darwin will be proven right. ❖

Acknowledgments

In the 42 years I have worked on system dependability, I have been fortunate to work with and learn from many colleagues at the University of Illinois, Caltech's Jet Propulsion Laboratory, and the UCLA Computer Science Department. University of Illinois professors James E. Robertson and David E. Muller were mentors and role models for my academic career. Work with my friends in the Computer Society's Technical Committee on Fault-Tolerant Computing and the IFIP Working Group 10.4 on Dependable Computing and Fault Tolerance has been rewarding and a source of inspiration for this article. Yutao He contributed valuable assistance in the survey of microprocessors and in the preparation of this text.

References

1. A. Avizienis, "Design of Fault-Tolerant Computers," *Proc. 1967 Fall Joint Computer Conf.*, AFIPS Conf. Proc., Vol. 31, Thompson Books, Washington, D.C., 1967, pp. 733-743.
2. *The Evolution of Fault-Tolerant Computing*, A. Avizienis, H. Kopetz, and J.-C. Laprie, eds., Springer-Verlag, New York, 1987.
3. J. von Neumann, "Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components," *Automata Studies*, C.E. Shannon and J. McCarthy, eds., *Annals of Math Studies*, No. 34, Princeton Univ. Press, Princeton, N.J., 1956, pp. 43-98.
4. E.F. Moore and C.E. Shannon, "Reliable Circuits Using Less Reliable Relays," *J. Franklin Institute*, Vol. 262, (in two parts), Sept./Oct. 1956, pp. 191-208 and 281-297.
5. W.H. Pierce, *Failure-Tolerant Computer Design*, Academic Press, New York, 1965.
6. A. Avizienis et al., "The STAR (Self-Testing-and-Repairing) Computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design," *IEEE Trans. Computers*, Nov. 1971, pp. 1,312-1,321.
7. A. Avizienis and J.-C. Laprie, "Dependable Computing: from Concepts to Design Diversity," *Proc. IEEE*, May 1986, pp. 629-638.
8. *Pentium Pro Family Developer's Manual*, Vols. 1-3, Intel, Mt. Prospect, Ill., 1996.
9. C.L. Chen et al., "Fault-Tolerance Design of the IBM Enterprise System/9000 Type 9021 Processor," *IBM J. Research and Development*, July 1992, pp. 765-779.
10. "The FAA's Advanced Automation Program," V.R. Hunt and G.V. Kloster, eds., special issue, *Computer*, Feb. 1987.
11. F. Cristian, R.D. Dancey, and J.D. Dehn, "Fault Tolerance in the Advanced Automation System," *Proc. 20th Int'l Symp. Fault-Tolerant Computing*, IEEE CS Press, Los Alamitos, Calif., 1990, pp. 6-17.
12. G.J.V. Nossal, "Life, Death and the Immune System," *Scientific Am.*, Sept. 1993, pp. 52-62.

Algirdas Avizienis is an emeritus professor at UCLA and rector emeritus at Vytautas Magnus University in Kaunas, Lithuania. As a senior research engineer at Caltech's Jet Propulsion Laboratory, he initiated research on interplanetary spacecraft computers in 1961 and published the concept of fault tolerance in 1967. His teaching and research interests include computer design, digital arithmetic, and fault-tolerant computing. He received a BS, an MS, and a PhD, all in electrical engineering, from the University of Illinois, Urbana-Champaign. A fellow of the IEEE and a member of the Lithuanian Academy of Sciences, he received the NASA Apollo Achievement Award, the American Institute of Aeronautics and Astronautics Information Systems Award, the NASA Exceptional Service Medal, and the IFIP Silver Core, the computer Society's Technical Achievement award, and the degree Docteur Honoris Causa' from the Institut National Polytechnique in Toulouse, France.

Contact Avizienis at the University of California, Los Angeles, Computer Science Dept., Los Angeles, CA 90095; aviz@cs.ucla.edu.