CONTENTS

1

Introduction

Before designing a dependable system, we need to have enough knowledge of the system's faults, errors, and failures of the dependable techniques including coding techniques, and of the design process for practical codes. This chapter provides the background on code design for dependable systems.

1.1 FAULTS AND FAILURES

First, we need to make clear the difference between three frequently encountered technical terms in designing dependable systems—namely faults, errors, and failures. These terms are fully defined in [LAPR92, AVIZ04]. Faults are primarily identified as the generic sources of abnormalities that alter the operation of circuits, devices, modules, systems, and / or software products. Failure can arise from any type of possible faults. Faults are often called *defects* when they occur in hardware and *bugs* when in software.

1.1.1 Faults

As causes of failure, faults are sometimes predictable but difficult to identify. Faults can occur during any stage in a system's or product's life cycle: during specification, design, production, or operation. Faults are characterized by their origin and their nature [LAPR92, GEFF02].

Origin of Faults Timing is a factor because faults can provoke failure in the operation phase at any one of a system's previous life phases: specification, design, production, and operation.

During the specification phase, for example, an incomplete definition of services may lead to different interpretations by the client, the designer, and the user. Eventually, in the

Code Design for Dependable Systems: Theory and Practical Applications, by Eiji Fujiwara

operation phase, the failure becomes evident when the services provided differ from the user's expectations.

During the design and the production phases, for example, a designer's lack of sufficient knowledge of architectural levels, structural levels, and the like, may result in a type of physical defect that induces, for example, short or open circuits.

During the operation phase, for example, an elevation of ambient temperature can cause electronic devices and products to malfunction.

Nature of Faults During the specification and the design phases, faults that occur are called *human-made faults*. During the production and the operation phases, these may occur *physical faults*, *hardware faults*, *or solid faults.* Each type is due to some physical abnormality in the component arising from aging or defective materials. Faults are of two types in their duration:

1. *Permanent.* These faults arise, for example, from a power supply breakdown, defective open or short circuits, *bridging* or *open lines*, *electro-migration*, and so forth. The defects in the input / output of the logical circuits or lines are called *stuck-at '1' faults* or *stuck-at '0' faults*.
2. *Temporal*. These faults can be *transient* or *intermittent*. Transient faults occur randomly and externally because of external noise, namely environmental problems of external *electromagnetic waves* but also external particles such as α-particles and neutrons. Intermittent faults occur randomly but internally because of unstable or marginally stable hardware, varying hardware or software state as a function of load or activity, or *signal coupling* (i.e., *crosstalk*) between adjacent signal lines. Some intermittent faults may be due to *glitches* [LO05], which are unpredictable spike noise pulses occurring and propagated especially in large exclusive-OR (XOR) tree networks (see Chapter 8). Parallel decoding circuits of error control codes with large code lengths require large exclusive-OR tree networks, so glitches can become serious problems. This topic will be covered in more detail in Section 8.3.

Transient faults and Intermittent faults are the major source of errors in modern-day digital systems. Some reports show that more than 60% of all failures in computer systems are caused by transient or intermittent faults. For example, in DRAM (Dynamic Random Access Memory) chips, transient errors result mainly from α*-particles* emitted by the decay of radioactive particles in the semiconductor materials [MAY79, NOOR80, SAIH82]. One identified source of α-particles is the lead solder balls used to attach the chip to the substrate. As they pass through the chip, α-particles create sufficient electron-hole pairs to add charge to the DRAM capacitor cells. These particles have low energy level, and thus have very low probability of causing more than one memory cell to flip when the memory cells are not packed in extreme density. In today's ultra–high-density RAMs, not only DRAMs but also SRAMs (Static Random Access Memories), it has been recognized that multiple cosmic-ray-induced transient errors are a serious problem [OSAD03, 04].

Temporal errors have also been observed in microprocessor chips. The trend toward smaller geometries by ever-shrinking semiconductor designs results in lower operating signal voltages and higher speed operation, and therefore brings additional transient or intermittent errors into play [KARN04]. In today's ubiquitous digital device or system environment, PDAs and personal computers equipped with these high-speed microprocessor chips and high-density RAM chips are further prone to be damaged by even worse circumstances when operated in airplanes at high altitude or near the high-voltage electric power lines.

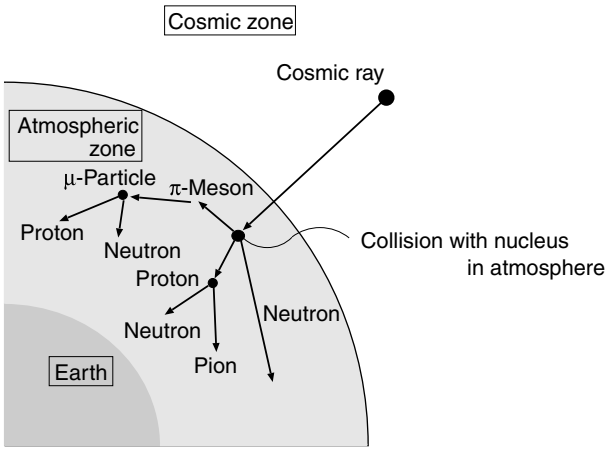


Figure 1.1 *Cosmic rays.*

The important point is that the faults due to temporary environmental problems do not need repair because the hardware is physically undamaged.

Cosmic rays, however, can give rise to significant transient errors, called soft errors [KARN04, MAKI00, HAZU00, ZIEG98, MASS96, CALV94]. Figure 1.1 shows the cosmic ray and its influence at the earth surface level. In the cosmic environment heavy particles with very high energy from solar winds can penetrate the semiconductor chips in satellite digital systems and cause more than double-bit errors [MUEL99]. Sometimes they can cause physical faults such as *latchup* in CMOS circuits.

A detailed report of field testing for soft errors due to cosmic rays was presented in 1996 [ZIEG96a, 96b, 96c, OGOR96, SRIN96]. In the report cosmic rays are defined as particles in solar wind originating in the sun or as galactic particles that enter the solar system striking atmospheric atoms and creating a shower of secondary particles. Most such particles produced by the shower either decay spontaneously or lose energy gradually, and eventually lose all energy in the cascade. Some of these particles may strike the earth. Therefore the cosmic rays at sea level consist mostly of neutrons, protons, pions, muons, electrons, and photons. About 95% of these particles are neutrons with no charge but with the high energy (more than 10 MeV) that causes significant soft errors or *latchups* in electronic circuits. So cosmic rays can create multiple errors. Altitude causes the neutron flux to increase exponentially, and hence the fail rate of electronic circuits at airplane altitude is about one hundred times worse than at terrestrial level. Concrete shielding with several feet of thickness can significantly attenuate the flux of these high-energy particles.

Figure 1.2 shows how neutrons and other particles, including α-particles, generated by the collision of nuclei in the atmosphere, can strike silicon chips and produce sufficient electron-hole pairs in the chips to impair their functioning.

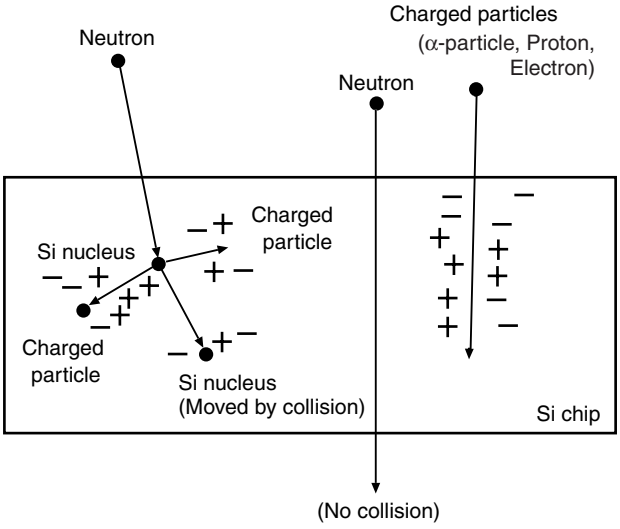


Figure 1.2 *Electron holes in a silicon chip caused by particles.*

1.1.2 Failures

A failure is defined as *nonperformance that occurs when a delivered service no longer complies with its specifications* [LAPR92], and a failure is also defined as *nonperformance when the system or component is unable to perform its intended function for a specified time under specified environmental conditions* [LEVE95].

Some types of failure are defined with respect to specific conditions. For example, a *value failure* means that the value of the delivered service does not comply with the specification and a *timing failure* represents a response in incorrect timing, either faster or slower than the specified time. A *temporary failure* means an erroneous behavior at a certain moment lasting only a short time. A *crash failure*, or *catastrophic failure*, is the one that stops the mission because the system is completely blocked.

1.2 ERROR MODELS

An error is a manifestation of an unexpected fault within a system that is liable to lead to system failure. The transformation of a fault to an error is called *fault activation*. The mechanism that creates errors in the system and finally provokes a failure is called *error propagation*. Before provoking a failure, errors can be masked or corrected by some error control mechanisms such as error correcting codes, retries, or triple modular redundancy (TMR) and thus recovered without inducing a system failure.

A fault remains in passive mode until an error first appears at some structure of the system. This occurrence is called an *initial activation* and the error is called a *primitive error*. In this case *latency* is defined as the mean time between the fault's occurrence and its initial activation as an error. Figure 1.3 presents the causal relationship between fault, error, and failure. Various types of errors can occur, and these different types are covered below.

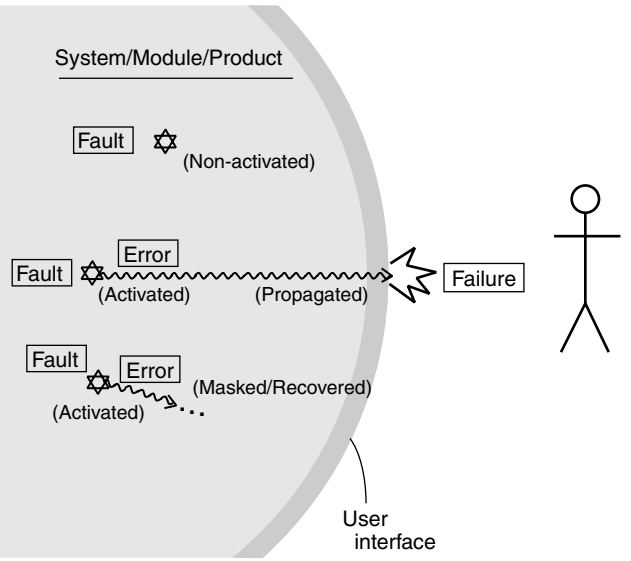


Figure 1.3 *Fault, error, and failure.*

1.2.1 Hard Errors and Soft Errors

Hard errors are caused by permanent faults; they therefore affect the system functions for a long period of time. This type of error is typically provoked by faults that appear as open or short anywhere on the chips, modules, cards, or boards. Hard errors are also called *permanent errors*.

Soft errors, on the other hand, are caused by temporal faults, especially those resulting from external causes. Soft errors have a limited duration, meaning they interrupt system functions for a very short time period. The most likely sources of soft errors are radioactive particles and external noise. Alpha particles and cosmic particles [ZIEG96a, ZIEG96b, ZIEG96c, OGOR96, SRIN96] are the major contributors mentioned previously. Therefore soft errors are also called *transient errors*. The *intermittent errors* are provoked by intermittent faults.

1.2.2 Random Errors, Clustered Errors, and Their Mixed-Type Errors

Multiple errors that occur randomly in time and / or space are called *random errors*. Error can occur in every bit position of a word with almost equal probability. The random type of error is unpredictable and is typically caused by white noise or external particles.

Errors may cluster non-uniformly in a word, and these multiple errors may gather in particular and unpredictable positions in the word. *Clustered errors* include *burst errors* and *byte errors*. Burst errors occur typically in disks or tape memory. Byte errors are typically found in semiconductor memory. The difference is in the data-recording medium. In disk memory, the data are recorded on a continuous surface. In semiconductor memory, the data are stored in RAM chips, and a data fragment, called a *byte*, is read or stored in each chip. In disk or tape memory, defects or dust particles on the recording surface can cause burst errors to occur anywhere in the continuous recording medium.

Clustered errors may occur in the two-dimensional matrix symbols as well as in the tape or disk memory of a continuous two-dimensional recording medium. In semiconductor memory, on the other hand, byte errors may occur in a fragment of readout data, namely in a single byte, corresponding to the faulty chip. This is because each chip is physically separated and independent, and therefore the presence of a fault in a chip does not extend to the adjacent chips. Figure 1.4 illustrates the different cases of random errors, byte errors, and burst errors.

Another error model consists of mixed clustered and random errors in the operational phase. The clustered errors mentioned above are sometimes caused by physical faults due

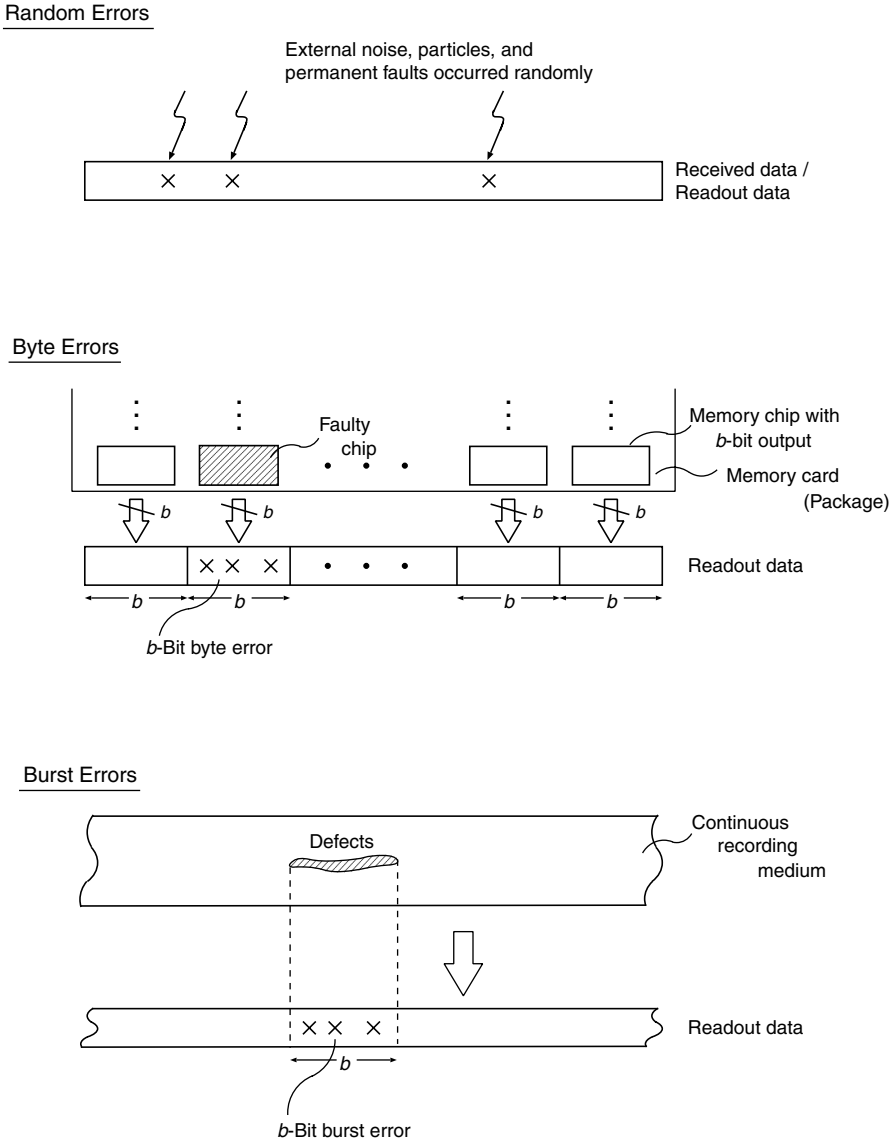


Figure 1.4 *Models of random errors, byte errors, and burst errors.*

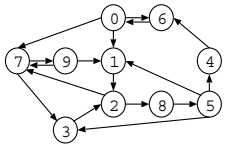
to aging problems. However, systems and devices are more prone to damage from transient faults than from physical faults. Transient faults are source of random errors. Therefore, when a physical fault occurs during the operational phase, both types of error—clustered and random—must be taken into account. For example, in semiconductor memories with byte-organized RAM chips, the major types of errors are transient errors, (i.e., random bit errors) caused by α-particles or external noises. After some time in operation, byte errors will occur due to the aging of RAM chips. Therefore both bit errors and byte errors, meaning both random errors and permanent errors, may occur separately or simultaneously. A similar situation holds for transmission systems, where both random bit errors and burst errors can occur. Chapter 6 deals with the codes which control the mixed type of single-byte errors and random bit errors.

1.2.3 Symmetric Errors, Asymmetric Errors, and Unidirectional Errors

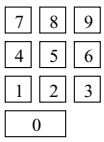
In binary systems the probability of errors that force 0 to 1, called *0-errors*, is, in general, equal to those going from 1 to 0, called *1-errors*. This class of errors is known as *symmetric errors*. When these errors occur with unequal probabilities, they are called *asymmetric errors*. In the binary asymmetric error model, only one type of error, either 0-errors or 1-errors, can occur, and the error type is known a priori. If both error types occur but are not mixed, then this class of errors is said to be *unidirectional errors* [BLAU93]. In binary systems these errors are caused by symmetric faults, asymmetric faults, or unidirectional faults.

In nonbinary systems using numerals, $0, 1, 2, 3, \ldots, 9$, or alpha-numeric symbols, asymmetric errors are the type that occur. That is, the probability of an error that forces one nonbinary symbol A to another symbol B is sometimes different from that of symbol A forced to yet another symbol C. For example, in handwritten character recognition systems, the probability of a 7 being mistaken for a 9 is much higher than that of a 7 being mistaken for a 4, or $p(9|7) \gg p(4|7)$, where $p(B|A)$ means probability of a symbol A being mistaken for another symbol B. This is because the numbers 7 and 9 are close in shape whereas 7 and 4 are not so similar. Likewise in keyboard input systems the symbols located on adjacent keys can be more easily mistyped. Figure 1.5 shows examples of these error models. In the asymmetric error model, the error graphs are not perfect and sometimes not bi-directional. On the other hand, in the symmetric nonbinary error model, they are perfect and bi-directional.

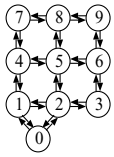
If symbols are removed or added in a word, as is sometimes caused by human mistakes (i.e., *human-made faults*), this class of errors is called *deletion errors* or *insertion errors*, respectively.



(a) Example of an asymmetric error graph for handwritten character (numerals) recognition systems



(Numeric key pad layout)



(b) Asymmetric error graph for keyboard systems

Figure 1.5 *Asymmetric errors in nonbinary systems.* Source: *[KANE04] © 2004 IEEE.*

1.2.4 Unequal Error Probability Model and the Unequal Error Protection Model

The probability of error appearing in any position of a word is usually considered to be equal. However, there is an error model to consider where some positions of a word have higher error probability than other positions. These are sometimes caused in the system by using devices with low reliabilities in the corresponding positions of a word, or by having error-sensitive areas in some positions of a word which are more vulnerable to external noises or have a low noise margin. In such cases the erroneous positions or areas with high error probabilities are known a priori. The type of error model that is relevant here is known as the *unequal error probability model*. The codes based on this error model are called *unequal error control (UEC) codes*. Chapter 10 will discuss the UEC codes and present its application to holographic memory, which has non-uniform error probability in the recording medium.

Some types of computer words or communication messages have a structure such that the information included in one part of the word is more important or more valuable than that in other parts. Control and address information in the computer or communication messages, and pointer information in the database words are good examples. In general, errors in this part, such as errors in control information or in pointer information, will cause much more serious damage to the subsequent processes in the system. Another example is error in the decimal numbers. During processing of digital data of conventional decimal numbers or measurement data, errors in the higher order digits will yield more devastating effects on the subsequent processes in digital systems than errors in the lower order digits. Therefore the higher order digits should be more strongly protected against errors than the lower order digits. This type of error model is known as an *unequal error protection model*. The codes based on this are called *unequal error protection (UEP) codes* and will also be discussed in Chapter 10.

1.3 ERROR RECOVERY TECHNIQUES FOR DEPENDABLE SYSTEMS

Error detection is an essential part of a dependable system design. Ideally, error detection will block the propagation of an error during online operations, before it reaches the system interface and causes a system failure. The error is best be detected immediately as it occurs so that its effect can be minimized.

Upon detecting an error by an error detection mechanism, some error recovery technique must mask the fault or remove it, and thus block the error's propagation. Among such mechanisms, error correcting codes and triple modular redundancy (TMR) correct errors or mask faults directly, that is, without an additional error detection procedure.

Some important error detection techniques and error recovery techniques, comparative to the error control coding techniques, are briefly described below. For more information, the reader is referred to the following excellent texts and papers on dependable systems or design techniques for fault tolerance: [AVIZ78, SIEW82, RENN84, EZHI86, ABRA86, PRAD86, JOHN89, LEE90, AVRE00].

1.3.1 Error Detection / Error Checking

Prediction & Comparison The basic error detecting or checking concept for online operations exists in *prediction & comparison*. That is, the output of the circuit / module is predicted from the input, and then the predicted output and the original circuit / module

output are compared bit by bit. The errors are detected if the actual output is not perfectly matched to the predicted output.

Duplication is an important and popularly used error detection technique in dependable digital systems. This is a special case of prediction & comparison, because the output is generated, or predicted, by a copy of the circuit / module and then compared with that of the original. This concept exists also in *software duplication* where a copy of the same or equivalent software is prepared and executed, and then the outputs are compared.

Parity-prediction is another important and popularly applied technique. The output parity bit is predicted from the input, and then compared with the parity bit generated from the original output.

Error Detecting Codes Error detecting codes typically deal with simple parity-check codes, cyclic codes, checksum codes, and other basic linear codes, as will be explained in Section 2.3. Some further important and newly developed codes will be presented in later chapters.

The application of error detecting codes in online operations is also called *checking* or an *online testing*. The error detection circuit is denoted as a *checker.* These applications will be examined in-depth in Section 12.1 where the self-checking concept is presented. Additional topics on how to detect errors caused by faults in the checker itself and how to design such checkers are covered in Section 12.2 where self-testing checkers are discussed. In summary, Chapter 12 covers error-checking concepts, self-testing checker design methodologies, and concrete checker design for logic circuits and for computer systems.

Watchdog Timer and Watchdog Processor A watchdog timer is very useful for detecting faults in a system. The idea behind this scheme is that some part of the system should act to indicate fault-free status so that absence of this action is indicative of a fault. Also the timer must be repeatedly reset by the system. Failure of the system to perform the reset function results in the system being turned off to prevent a system failure from occurring.

A watchdog timer can be used to detect faults in both the hardware and the software of a system. In many applications software routines are expected to execute within prespecified time frame. In digital control systems, for example, the routines execute repetitively at specified intervals. If a routine suddenly needs more than the expected time to execute, the fault may be in the software's, for example, infinite loop [JOHN89]. In this regard the watchdog timer is an important *control flow check tool*.

A watchdog processor is an extension of the concept of a watchdog timer. This is a special subprocessor that checks the online operations of the processor being checked. The watchdog processor runs the watchdog programs that collect information from the processor being checked and generate *signatures*, such as address and data information, and processor state information, during online operations. The new information is then compared to that already prepared in the watchdog program.

1.3.2 Error Recovery / Error Masking

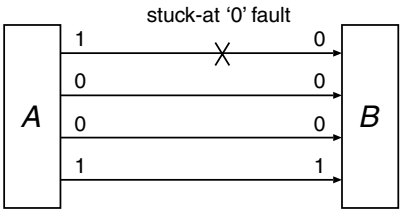
Error recovery techniques are essential to improving system reliability. The important recovery techniques, as was mentioned before, include coding techniques and some modular redundancy techniques, such as TMR, that correct or mask the faults directly. Other effective error detection methods are also available to mask the faults after the detection of errors, for example, self-checked duplication and sift-out redundancy, as discussed below.

Error Correcting Codes Many different error control codes have been studied and developed to correct and / or detect the types of errors mentioned in Section 1.2. Among the most practical matrix codes are those presented in this book.

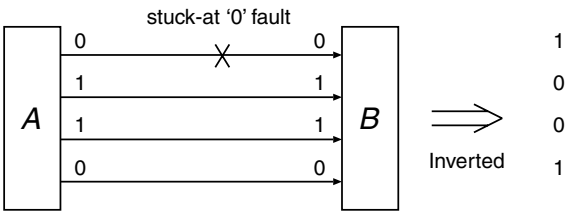
Error correcting codes head the list of the most effective and efficient techniques used to mask faults, both temporal and permanent. The coding approach involves some redundancy, for example, additional check bits, additional hardware in the form of encoding / decoding logic circuits, and additional decoding time delay. Nevertheless, the coding performance is superior to that competitive techniques, especially in quickly masking of temporal faults. For this reason error control codes are still being extensively applied to various digital systems to improve their reliability.

Retry Just as *space redundancy* requires additional hardware resources, the retry method called *time redundancy* which requires additional time to perform multiple identical operations of commands or programs immediately after errors are detected. This very simple technique requires almost no additional hardware but can very effectively recover system operations from temporary faults, meaning transient and intermittent faults. Therefore the retry method is popularly applied to digital systems, including processors, main memories, disk memories, tape memories, and I/O devices.

Alternate data retry, abbreviated by *ADR* [SHED78], is a kind of retry operation that is effective in masking permanent faults besides temporary faults. Figure 1.6 presents the principle behind masking a single permanent fault by ADR. Note that this simple example shows the even-parity encoded bus circuit with four lines, including a parity line. Figure 1.6(a) shows that if a stuck-at '0' permanent fault occurs in the first bus line, then the even-parity encoded data from circuit *A*, here 1001, is received at the input of the circuit *B* as 0001, which is an odd-parity encoded data. Therefore a single error can be detected by examining the parity check of the data. Next, by the ADR method, in Figure 1.6(b), the bit-by-bit complement of the original data, which is 0110, is transmitted from circuit *A* to



(a) Error detection by parity check



(b) Retried by complemented data

Figure 1.6 *Principle of ADR (illustrated by even-parity encoded bus line circuits as an example).*

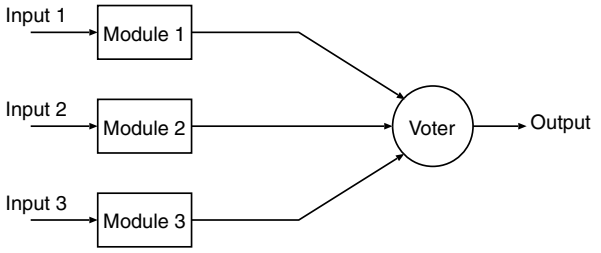
the input of the circuit B. Even though the first line is still preserving a stuck-at '0' fault, the fault is masked because the data on this line are also a '0'. Finally the received data are inverted, and then the original correct data, 1001, are recovered. In this example, a permanent fault is masked at the second stage of ADR, and finally the correct data are recovered at the third stage of ADR. Also, in this example, if the fault in Figure 1.6(a) is a temporary fault, the error it caused can be completely masked and will have no effect because the temporary fault will disappear by the time of the second stage of ADR.

In general, if the logic circuit that performs the function $F(X)$ for the circuit input X satisfies the relation

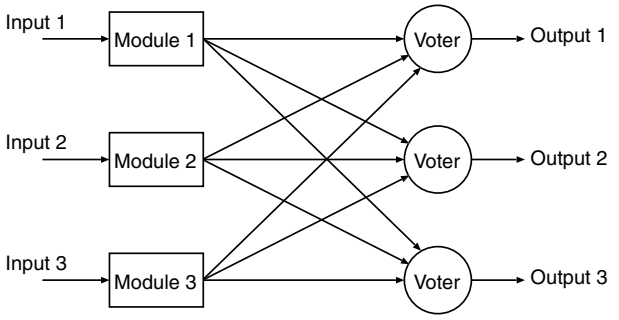
$$F(\overline{X}) = \overline{F(X)},$$

where $\overline{X}$ means the complement of X, then the ADR with bit-by-bit complementary retry at the second stage can be performed successfully. The function F that satisfies the relation above is called a *self-complementary function*, and the circuit that satisfies the relation is called a *self-complementary circuit*. The former even-parity busline circuit is a self-complementary circuit. The adder, the multiplier, and the divider are also good examples of self-complementary circuits.

N-Modular Redundancy (NMR) and Reconfiguration *Triple modular redundancy* (TMR) is the most typical form of N-modular redundancy. The TMR method triplicates the original module and performs a majority vote to determine the output of the system. If one of the modules becomes faulty, the other two fault-free modules mask the results of the faulty one when the majority vote is performed. This is shown in Figure 1.7(a). This voting concept



(a) Triple modular redundancy (TMR)



(B) Triple modular redundancy with triplicated voters

Figure 1.7 *Triple modular redundancy (TMR).*

is applied to TMR software to protect against software faults in any one of three identical or equivalent software programs that perform the same function.

The difficulty in the TMR exists in its voter. That is, if the voter fails, the system completely fails. One approach is to apply TMR to the voter itself such that three voters are used and three independent voting results are provided as shown in Figure 1.7(b). The three modules are functionally identical and receive identical inputs. The results generated by three modules are voted on by the three voters to produce three results. Each result is correct unless more than one module or input is faulty.

N-Modular redundancy (NMR) is a generalization of the TMR and is a typical *space redundancy* technique. In most cases, N is selected as an odd number so that a majority voting principle can be applied. For example, the 5MR system consists of five identical modules and a voter. This system produces correct output in the presence of, or masks, as many as two faulty modules.

The modular redundancy concept has been extended and modified by combining the concept of *reconfiguration*. The following forms show some such combinations.

Self-checked duplication is an extended form of duplication in which each module has its own self-checking mechanism in order to identify the faulty state of the module itself. In this system, two self-checked modules are operated and checked in parallel at all times. If one module is found to have errors by its own error detection mechanism, then the system output is switched to the error-free module, meaning it is reconfigured. This concept is a form of *hot standby sparing* in which the spare module operates synchronously with the online module and is prepared to take over at any time. When the online module is failed, the standby spare module takes over immediately. In contrast to the hot standby sparing, there exists *cold standby sparing* where the spare is unpowered until needed to replace a faulty module.

N-Modular redundancy with spares is also known as hybrid redundancy. It provides a basic core of N modules arranged in a voting system, and in addition spares are provided to replace faulty modules. For example, while the TMR with one spare masks one faulty module, the spare will replace the faulty module immediately upon the detection of the fault. After that spare is used, the system is still capable of masking another faulty module. Therefore two faulty modules can be masked in this system. The aforementioned 5MR requires five modules in order to mask two faulty modules, but the TMR with one spare approach requires only four modules. The system remains in the basic NMR configuration until the disagreement detector determines that a faulty module exists. One approach to fault detection is to compare the output of the voter with the individual outputs of the modules. A module that disagrees with the majority is regarded as faulty and removed from the NMR core. A spare module is then switched in to replace the faulty module. The reliability of the basic NMR system is maintained as long as the pool of spares is not exhausted. This is shown in Figure 1.8.

Self-Purging Redundancy is similar to the NMR with the spare modules approach. The main difference is that all modules operate actively in this redundancy system, unlike the NMR with spares where some spare modules are not an active part of the system until a fault occurs. This is shown in Figure 1.9. Each switch in the self-purging redundancy separates the faulty module if the module output is not equal to the voter output. The reconfiguration is essentially accomplished by the system logically removing the faulty module via the switch and thus reducing the number of N in the reconfigured NMR system.

Sift-out redundancy also requires N identical modules in the system but with every pair of two module outputs compared to identify faulty modules. If there exist $N = 5$ modules,

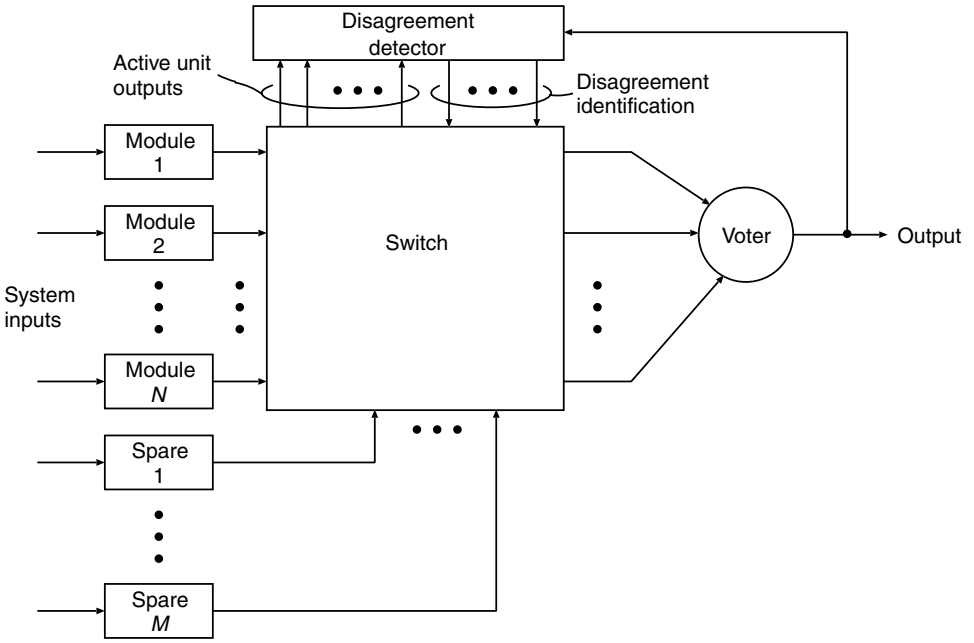


Figure 1.8 *N-modular redundancy with spares (i.e., hybrid redundancy).*

ten comparisons are performed. This redundancy requires an N-way multiplexer instead of an N-input voter, as shown in Figure 1.10. The comparator in this redundancy circuit receives all outputs of the modules and produces comparison outputs of every two modules, that is, $N(N-1)/2$ outputs, and then determines the faulty modules in the detection circuit. Finally the output of the N-way multiplexer is selected based on the faulty indication outputs of the detection circuit. This essentially masks the effects of any faulty modules.

This redundancy can tolerate up to $N-2$ faulty modules. Its tolerance is therefore equal to the TMR system with $N-3$ spares and also to the self-purging system having a voter with threshold level of two.

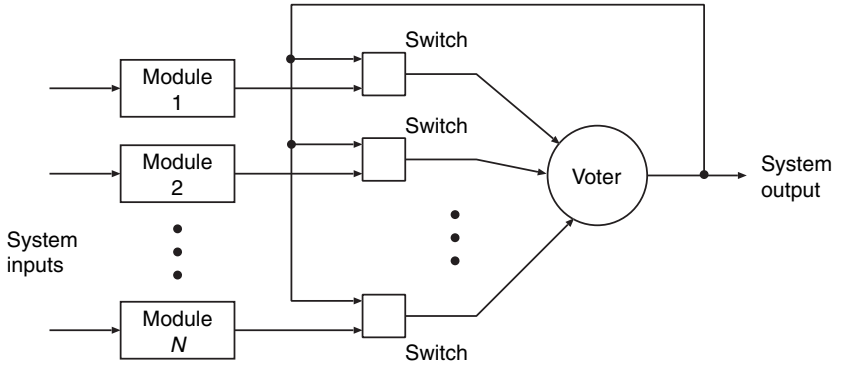


Figure 1.9 *Self-purging redundancy.*

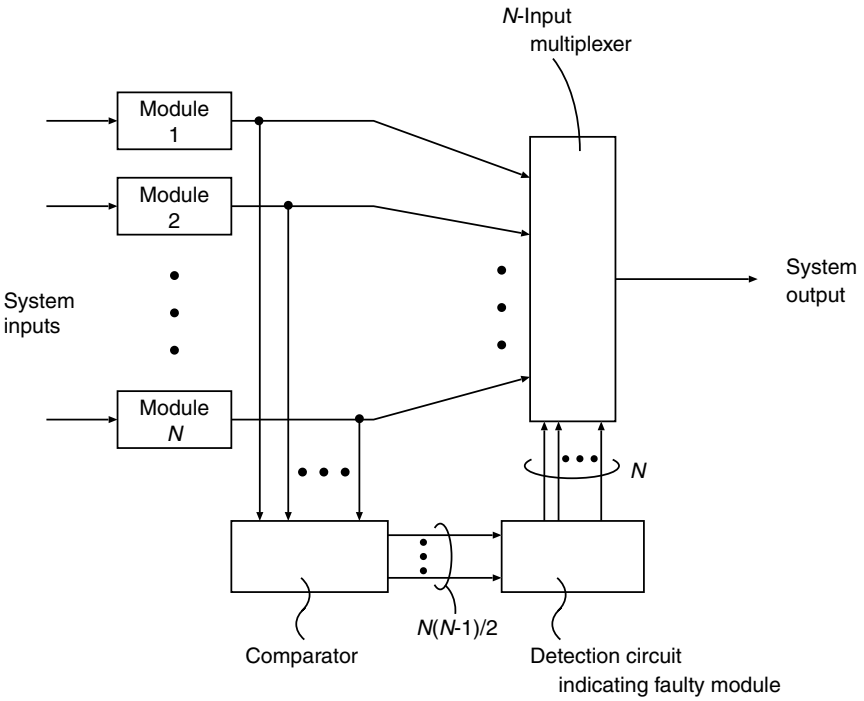


Figure 1.10 *Shift-out redundancy.*

System Recovery by Software Retry techniques require error detection by checkers, and immediately after the error detection the same operations are performed. In contrast, *checkpoint techniques* allow some latency time after error detection because the process can be restored to an earlier point of execution. *Checkpointing* is mostly implemented in software and requires some hardware to store the backup data. The techniques result from a combination of checkpointing and rollback. In checkpointing, complete copy of the system state should be saved at specific points, namely *checkpoints*, during process execution. The information to be stored is the set of system state including data, programs, machine state, and so forth, which is necessary to restart the continued successful execution from the checkpoint. *Rollback* is a part of actual recovery process and occurs after the repair, such as by reconfiguration, that removes faulty modules or equipments from the system, or after the error due to transient faults has died out. An important design criterion is how often checkpoints are to be set, that is, in determining *checkpoint intervals*. If the checkpoints are too infrequent for the actual error rate experienced, too much computation time will be lost due to rollback. On the other hand, too frequent checkpoints result in an unnecessary increase in operation time and memory due to the overhead of saving system states when establishing checkpoints.

1.4 CODE DESIGN PROCESS FOR DEPENDABLE SYSTEMS

What types of dependable techniques are the most effective in the design of dependable systems? In some cases other than coding techniques, or a combination of coding techniques and other dependable techniques, will better meet the reliability requirement or the cost / performance requirement of a system.

Before designing the error control codes, we therefore have to pay attention to a number of preconditions or preparatory measures: Where to apply the code? How to apply the code effectively? How much reliability of the system to improve and satisfy its performance by coding techniques? What are the requirements for decoding speed, and how much decoder hardware? What about the detection capability of errors falling outside the capability of the code? This section addresses all these important questions with respect to the code design process.

1.4.1 Code Functions

Error detection and error correction are the more known code functions. An important code function that lies midway between these two functions is *error location*. The error locating code indicates which blocks, or components of a word contain error but does not indicate the precise erroneous digit position nor the error value. This is a code function that is efficient for retransmission of a word segment, especially in communication systems where whole words do not need retransmitting [WOLF63]. Also in computer systems the error locating code provides the information on where to find the faulty module, faulty package / card, or faulty device, which is very useful for system maintenance. If the system is equipped with spares, then the system can be recovered by removing the faulty blocks and switching to the spares.

Figure 1.11 shows the different functions of these three code types. Because erroneous position, and error value can be determined by use of ''error correction'', all errors can be

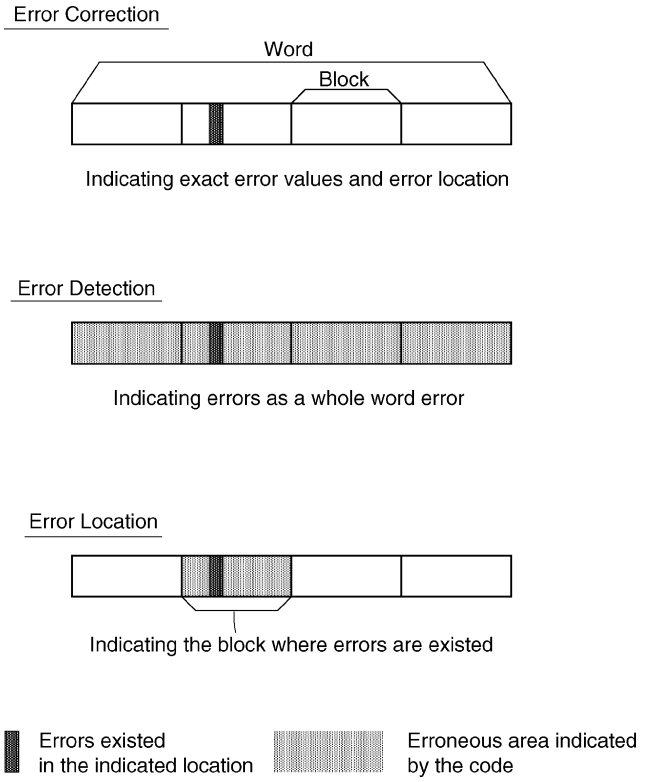


Figure 1.11 *Code functions.*

corrected. Of course, use of ''error detection'' alone does not allow any erroneous position nor error value to be determined; it only indicates the presence of error in a word. For ''error location'', as was mentioned before, only the area where the word includes an erroneous position is indicated by the code. For example, note in Figure 1.11 that the code's information is that errors exist in the second block of the word and no definite error positions in the block nor the error values are determined. Error locating codes will be covered in Chapter 9. Many practical codes, in general, have a mixture of these code functions, for example, single error correction and double error detection.

1.4.2 Code Deisgn Process

Before attempting the design of codes, we need to give the following items our careful consideration:

1. Circumstance where the systems or equipments with the coding techniques are to be applied, for example, the particular needs of medical appliances, nuclear appliances, or digital systems in aircraft or satellite,
2. Fabrication structure, that is, how the systems or the equipments are organized, for example, chip / card (package) organization, bit / byte organization, or binary / nonbinary,
3. Devices, such as memories, logic circuits, or FPGAs that are used in the system to which the coding techniques are to apply.
4. Combination of fault / error masking techniques with coding techniques.

The design process for the error control codes is presented next, and is shown in Figure 1.12. Steps 1 through 3 pertain to the phase of setting code parameters, and steps 4 and 5 are for the phase of code designing.

Step 1. Determine error rates and error types:

- Raw error rate of devices, modules, or systems, and what target error rate to attain
- Whether symmetric error, asymmetric error, or unidirectional error
- Whether equal error or unequal error
- Whether random bit error, byte error, spotty byte error,[a] or burst error
- Whether bit or byte error,[*] or rather, bit plus byte error[b]

Step 2. Determine code parameters and code constraints:

- Information-bit length, and required check-bit length
- Maximum random bit error length—or byte error length, spotty error length,[a] or burst error length
- Required decoding speed
- Required decoder hardware complexity

[a] See Chapter 7.

[b] See Chapter 6.

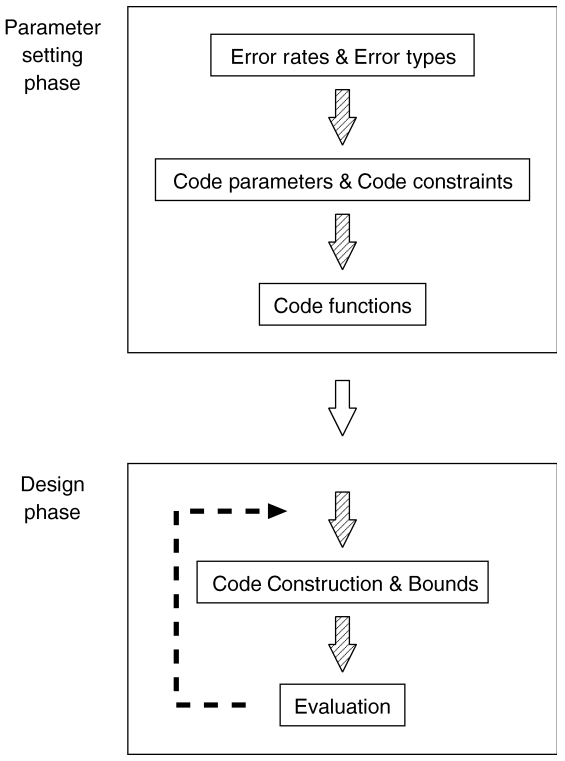


Figure 1.12 *Code design process.*

Step 3. Determine code function:

- Error detection, error correction, error location, or mixed type of these code functions

Step 4. Design code, and calculate code bounds:

- Theoretical bound on code length or check-bit length
- Mathematical knowledge required for code design, for example, algebra, combinatorial mathematics, number theory, graph theory, statistics, and probability theory

Step 5. Evaluate the code designed:

- Check-bit length, and comparison to its bound
- Decoding speed
- Decoder hardware complexity
- Error detection probability of multiple errors beyond the code capability
- If the code does not satisfy the requirements, then go back to step 4

REFERENCES

[ABRA86] J. A. Abraham and W. K. Fuchs, ‘‘Fault and Error Models for VLSI,’’ *Proc. IEEE*, 74 (May 1986): 639–654.

[AVIZ78] A. Avizienis, ''Fault Tolerance, the Survival Attribute of Digital Systems'' *Proc. IEEE*, 66 (October 1978): 1109–1125.

[AVIZ04] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, ''Basic Concepts and Taxonomy of Dependable and Secure Computing,'' *IEEE Trans. Depend. Secure Comput.*, 1 (January–March 2004): 11–33.

[AVRE00] D. R. Avresky (ed.), *Dependable Network Computing*, Kluwer Academic Publishers (2000).

[BLAU93] M. Blaum, *Codes for Detecting and Correcting Unidirectional Errors*, IEEE Computer Society Press (1993).

[CALV94]P. Calvel, P. Lamothe, and C. Barillot, ''Space Radiation Evaluation of 16 Mbit DRAMs for Mass Memory Applications,'' *IEEE Trans. Nucl. Sci.*, 41 (December 1994): 2267–2271.

[EZHI86] P. D. Ezhilchevan and S. K. Shrivastava, ''A Characterization of Faults in Systems,'' *Proc. 5th Symp. on Reliability in Distributed Software and Database Systems* (January 1986): 215–222.

[GEFF02] J.-C. Geffroy and G. Motet, *Design of Dependable Computing Systems*, Kluwer Academic Publishers (2002), chs. 1–5.

[HAZU00] P. Hazucha, C. Svensson, and S. A. Wender, ''Cosmic-Ray Soft Error Rate characterization of a Standard 0.6-μm CMOS Process,'' *IEEE J. Solid-State Circ.*, 35 (October 2000): 1422–1429.

[JOHN89] B. W. Johnson, *Design and analysis of Fault Tolerant Digital Systems*, Addison Wesley (1989).

[KANE04] H. Kaneko and E. Fujiwara, ''A Class of *M*-Ary Asymmetric Symbol Error Correcting Codes for Data Entry Devices,'' *IEEE Trans. Comput.*, 53 (February 2004): 159–167.

[KARN04] T. Karnik, P. Hazucha, and J. Patel, ''Characterization of Soft Errors Caused by Single Event Upsets in CMOS Processes,'' *IEEE Trans. Depend. Secure Comput.*, 1 (April–June 2004): 128–143.

[LAPR92] J. C. Laprie (ed.), *Dependability: Basic Concepts and Terminology*, Springer-Verlag (1992).

[LEE90] P. A. Lee and T. Anderson, *Fault Tolerance, Principles and Practice*, Springer-Verlag (1990).

[LEVE95] N. Leveson, *Safeware*, Addison-Wesley (1995).

[LO05] J. C. Lo and E. Fujiwara,''Transient Behavior of the Encoding/Decoding Circuits of Error Control Codes,'' *Proc. IEEE Int. Symp. on Defect and Fault Tolerance in VLSI Systems* (October 2005):

[MAKI00] A. Makihara, H. Shindou, N. Nemoto, S. Kuboyama, S. Matsuda, T. Oshima, T. Hirao, H. Itoh, S. Buchner, and A. B. Campbell, ''Analysis of Single-Ion Multiple-Bit Upset in High-Density DRAMs,'' *IEEE Trans. Nucl. Sci.*, 47 (December 2000): 2400–2404.

[MASS96] L. W. Massengill, ''Cosmic and Terrestrial Single-Event Radiation Effects in Dynamic Random Access Memories,'' *IEEE Trans. Nucl. Sci.*, 43 (April 1996): 576–593.

[MAY79] T. C. May, ''Soft Errors in VLSI: Present and Future,'' *IEEE Trans. Comp. Hybrids Manuf. Technol.*, CHMT-2 (December 1979): 377–387.

[MUEL99] M. Mueller, L. C. Alves, W. Fischer, M. L. Fair, and I. Modi, ''RAS Strategy for IBM S/390 G5 and G6,'' *IBM J. Res. Dev.*, 43 (September–November 1999): 875–888.

[NOOR80] D. J. W. Noorlag, L. M. Terman, and A. G. Konheim, ''The Effect of Alpha-Particle-Induced Soft Errors on Memory Systems with Error Correction,'' *IEEE J. Solid-State Circ.*, SC-15 (June 1980): 319–325.

[OGOR96] T. J. O'Gorman, J. M. Ross, A. H. Taber, J. F. Ziegler, H. P. Muhlfeld, C. J. Montrose, H. W. Curtis, and J. L. Walsh, ''Field Testing for Cosmic Ray Soft Errors in Semiconductor Memories,'' *IBM J. Res. Dev.*, 40 (January 1996): 41–50.

[OSAD03] K. Osada, Y. Saitoh, E. Ibe, and K. Ishibashi, ''16.7-fA/Cell Tunnel-Leakage-Suppressed 16-Mb SRAM for Handling Cosmic-Ray-Induced Multierrors,'' *IEEE J. Solid-State Circ.*, 38 (November 2003): 1952–1957.

[OSAD04] K. Osada, K. Yamaguchi, Y. Saitoh, and T. Kawahara, ''SRAM Immunity to Cosmic-Ray-Induced Multierrors Based on Analysis of an Induced Parasitic Bipolar Effect,'' *IEEE J. Solid-State Circ.*, 19 (May 2004): 827–833.

[PRAD86] D. K. Pradhan, *Fault-Tolerant Computing*, Vol. 1 and 2, Prentice-Hall (1986).

[RENN84] D. A. Rennels, ''Fault-Tolerant Computing—Concepts and Examples,'' *IEEE Trans. Comput.*, C-33 (December 1984): 1116–1129.

[SAIH82] G. A. Sai-Halasz, M. R. Wordeman, and R. H. Denard, ''Alpha-Particle-Induced Soft Error Rate in VLSI Circuits,'' *IEEE J. Solid-State Circ.*, SC-17 (April 1982): 355–361.

[SELL68] F. F. Sellers, Jr., M. Y. Hsiao, L. W. Bearnson, *Error Detecting Logic for Digital Computers*, McGraw-Hill (1968).

[SHED78] J. J. Shedletsky, ''Error Correction by Alternate-Data Retry,'' *IEEE Trans. Comput.*, C-27 (February 1978): 106–112.

[SIEW82] D. P. Siewiorek and R. S. Swarz, *The Theory and Practice of Reliable System Design*, Digital Press (1982).

[SRIN96] G. R. Srinivasan, ''Modeling the Cosmic-Ray-Induced Soft-Error Rate in Integrated Circuits: An Overview,'' *IBM J. Res. Dev.*, 40 (January 1996): 77–89.

[WOLF63] J. K. Wolf and B. Elspas, ''Error-Locating Codes—A New Concept in Error Control,'' *IEEE Trans. Info. Theory*, IT-9 (April 1963): 113–117.

[ZIEG96a] J. F. Ziegler, H. W. Curtis, H. P. Muhlfeld, C. J. Montrose, B. Chin, etc., ''IBM Experiments in Soft Fails in Computer Electronics (1978–1994),'' *IBM J. Res. Dev.*, 40 (January 1996): 3–18.

[ZIEG96b] J. F. Ziegler, ''Terrestrial Cosmic Rays,'' *IBM J. Res. Dev.*, 40 (January 1996): 19–39.

[ZIEG96c] J. F. Ziegler, H. P. Muhlfeld, C. J. Montrose, H. W. Curtis, T. J. O'Gorman, and J. M. Ross, ''Accelerated Testing for Cosmic Soft-Error Rate,'' *IBM J. Res. Dev.*, 40 (January 1996): 51–72.

[ZIEG98] J. F. Ziegler, M. E. Nelson, J. D. Shell, R. J. Peterson, C. J. Gelderloos, H. P. Mahlfeld, and C. J. Montrose, ''Cosmic Ray Soft Error Rates of 16-Mb DRAM Memory Chips,'' *IEEE J. Solid-State Circ.*, 33 (February 1998): 246–252.