TABLE 22.3 Operations for the Radix-4 Modified Booth Algorithm

a_{i+1}	a_i	a_{i-1}	Operation
0	0	0	$P = P/4$
0	0	1	$P = (P + B)/4$
0	1	0	$P = (P + B)/4$
0	1	1	$P = (P + 2B)/4$
1	0	0	$P = (P - 2B)/4$
1	0	1	$P = (P - B)/4$
1	1	0	$P = (P - B)/4$
1	1	1	$P = P/4$

numbers, where each cycle consists of either an n-bit addition and a shift, an n-bit subtraction and a shift, or a shift without any other arithmetic operation.

22.3.3.2 Sequential Modified Booth Multiplier

The radix-4 modified Booth multiplier [MacSorley 1961] uses $n/2$ cycles in which each cycle examines three adjacent bits; adds or subtracts 0, B, or $2B$; and shifts 2 bits to the right. Table 22.3 shows the operations as a function of the three bits: a_{i+1}, a_i, and a_{i-1}. The radix-4 modified Booth multiplier requires half as many cycles as the standard (radix-2) Booth multiplier although the operations performed during each cycle are slightly more complex (because it is necessary to select one of five possible addends instead of one of three). Extensions to higher radices that examine more than 3 bits are possible but generally are not attractive because the addition/subtraction operations involve multiples of B that are not powers of two (such as 3, 5, etc.), which raises the complexity.

22.3.3.3 Array Multipliers

An alternative approach to multiplication involves the combinational generation of all bit products and their summation with an array of full adders. The block diagram for an 8-by-8 array multiplier is shown in Figure 22.9. It uses an 8-by-8 array of AND gates to form the bit products (some of the AND gates are denoted G in Figure 22.9 and the remainder are contained within the FA and HA blocks) and an 8-by-7 array of adders [48 full adders (denoted FA in Figure 22.9) and 8 half-adders (denoted HA in Figure 22.9)] to sum the 64-bit products. Summing the bits requires 14 adder delays.

Modification of the array multiplier to multiply two's complement numbers requires inverting most of the signals in the most significant row and column and also adding a few correction terms [Baugh and Wooley 1973, Blankenship 1974]. Array multipliers are easily laid out in a cellular fashion, making them suitable for very large scale integrated (VLSI) implementation, where minimizing the design effort may be more important than maximizing the speed of the multiplier.

22.3.3.4 Wallace Tree/Dadda Fast Multiplier

A method for fast multiplication was developed by Wallace [1964] and refined by Dadda [1965]. With this method, a three-step process is used to multiply two numbers: (1) the bit products are formed; (2) the bit product matrix is reduced to a two-row matrix in which the sum of the rows equals the sum of the bit products; and (3) the two numbers are summed using a fast adder to produce the product. Although this may seem to be a complex process, it yields multipliers with delay proportional to the logarithm of the operand word size, which is faster than the Booth multiplier, the modified Booth multiplier, or array multipliers, all of which have delays proportional to the word size.

The second step in the fast multiplication process is shown for an 8-by-8 Dadda multiplier in Figure 22.10. An input 8-by-8 matrix of dots (each dot represents a bit product) is shown as Matrix 0. Columns having more than six dots (or that will grow to more than six dots due to carries) are reduced by the use of half-adders (each half-adder takes in two dots and outputs one in the same column and one in the next

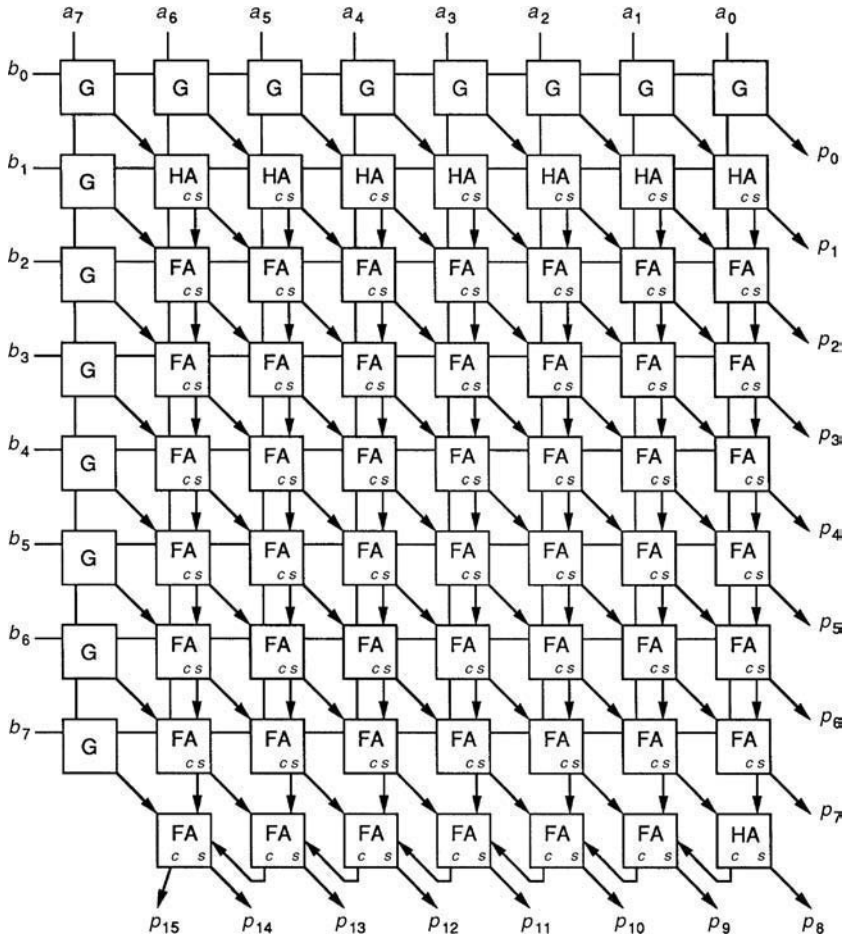


FIGURE 22.9 Unsigned 8-by-8 array multiplier.

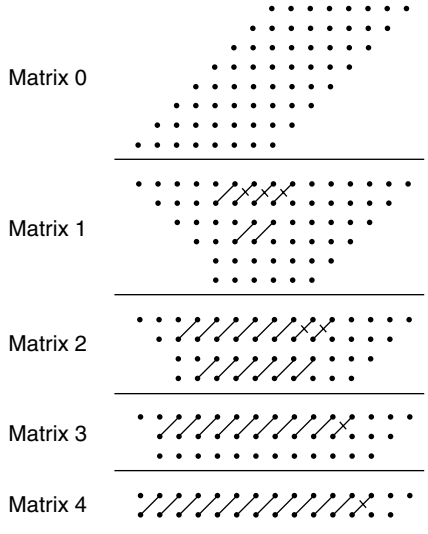


FIGURE 22.10 Unsigned 8-by-8 Dadda multiplier.

more significant column) and full adders (each full adder takes in three dots and outputs one in the same column and one in the next more significant column) so that no column in Matrix 1 will have more than six dots. Half-adders are shown by two dots connected with a crossed line in the succeeding matrix and full adders are shown by two dots connected with a line in the succeeding matrix. In each case, the rightmost dot of the pair connected by a line is in the column from which the inputs were taken for the adder. In the succeeding steps, reduction to Matrix 2 with no more than four dots per column, Matrix 3 with no more than three dots per column, and finally Matrix 4 with no more than two dots per column is performed. The height of the matrices is determined by working back from the final (two-row) matrix and limiting the height of each matrix to the largest integer that is no more than 1.5 times the height of its successor. Each matrix is produced from its predecessor in one adder delay. Because the number of matrices is logarithmically related to the number of bits in the words to be multiplied, the delay of the matrix reduction process is proportional to $\log n$. Because the adder that reduces the final two-row matrix can be implemented as a carry lookahead adder (which also has logarithmic delay), the total delay for this multiplier is proportional to the logarithm of the word size.

22.3.4 Fixed Point Division

Division is traditionally implemented as a digit recurrence requiring a sequence of shift, subtract, and compare operations, in contrast to the shift and add approach employed for multiplication. The comparison operation is significant. It results in a serial process, which is not amenable to parallel implementation. There are several digit recurrence-based division algorithms, including binary restoring, nonperforming, nonrestoring, and Sweeney, Robertson, and Tocher (SRT) division algorithms for a variety of radices [Ercegovac and Lang 1994].

22.3.4.1 Nonrestoring Divider

Traditional nonrestoring division is based on selecting digits of the quotient Q (where $Q = N/D$) to satisfy the following equation:

$$P_{k+1} = r\,P_k - q_{n-k-1}\,D \qquad \text{for } k = 1, 2, \ldots, n-1 \tag{22.25}$$

where P_k is the partial remainder after the selection of the kth quotient digit, $P_0 = N$ (subject to the constraint $|P_0| < |D|$), r is the radix ($r = 2$ for binary nonrestoring division), q_{n-k-1} is the kth quotient digit to the right of the binary point, and D is the divisor. In this section, it is assumed that both N and D are positive; see Ercegovac and Lang [1994] for details on handling the general case.

In nonrestoring division, the quotient digits are constrained to be ± 1 (i.e., q_k is selected from $\{1, \bar{1}\}$). The digit selection and resulting partial remainder are given for the kth iteration by the following relations:

$$\text{If } P_k \geq 0, \qquad q_{n-k-1} = 1 \qquad \text{and} \qquad P_{k+1} = r\,P_k - D \tag{22.26}$$

$$\text{If } P_k < 0, \qquad q_{n-k-1} = \bar{1} \qquad \text{and} \qquad P_{k+1} = r\,P_k + D \tag{22.27}$$

This process continues either for a set number of iterations or until the partial remainder is smaller than a specified error criterion. The kth most significant bit of the quotient is a 1 if P_k is 0 or positive and is a $\bar{1}(-1)$ if P_k is negative. The algorithm is illustrated in Figure 22.11, where $\frac{5}{16}$ is divided by $\frac{3}{8}$. The result ($\frac{13}{16}$) is the closest 4-bit fraction to the correct result of $\frac{5}{6}$.

The signed digit number (comprising ± 1 digits) can be converted into a conventional binary number by subtracting n, the number formed by the negative digits (with 0s where there are 1s in Q and 1s where

Nonrestoring division

$N = \frac{5}{16} = 0.0101$

$D = \frac{3}{8} = 0.0110$

$P_{(0)} = N$

Since $P_{(0)} \geq 0$, $q_3 = 1$ and $P_{(1)} = 2P_{(0)} - D$

$$\begin{array}{rl} 2P_{(0)} & = 0.1010 \\ \underline{-D} & \underline{\quad 1.1010} \\ P_{(1)} & = 0.0100 \end{array}$$

Since $P_{(1)} \geq 0$, $q_2 = 1$ and $P_{(2)} = 2P_{(1)} - D$

$$\begin{array}{rl} 2P_{(1)} & = 0.1000 \\ \underline{-D} & \underline{\quad 1.1010} \\ P_{(2)} & = 0.0010 \end{array}$$

Since $P_{(2)} \geq 0$, $q_1 = 1$ and $P_{(3)} = 2P_{(2)} - D$

$$\begin{array}{rl} 2P_{(2)} & = 0.0100 \\ \underline{-D} & \underline{\quad 1.1010} \\ P_{(3)} & = 1.1110 \end{array}$$

Since $P_{(3)} < 0$, $q_0 = \bar{1}$ and $P_{(4)} = 2P_{(3)} + D$

$$\begin{array}{rl} 2P_{(3)} & = 1.1100 \\ \underline{+D} & \underline{\quad 0.0110} \\ P_{(4)} & = 0.0010 \end{array}$$

$Q = 0.111\bar{1} = 0.1101 = \frac{13}{16}$

FIGURE 22.11 Example of nonrestoring division.

there are $\bar{1}$s in Q), from p, the number formed by the positive digits (with 0s where there are $\bar{1}$s in Q and 1s where there are 1s in Q). For the example of Figure 22.11:

$$\begin{aligned} Q &= 0.111\bar{1} \\ n &= 0.0001 \\ p &= 0.1110 \\ p - n &= 0.1110 - 0.0001 \\ &= 0.1110 + 1.1111 \\ &= 0.1101 \end{aligned}$$

Other digit recurrence division algorithms include the restoring algorithm and the SRT algorithms [Robertson 1958]. The restoring algorithm is similar to the nonrestoring algorithm presented here except that either subtract and shift or shift are permitted for each iteration instead of subtract and shift or add and shift as in Equations 22.26 and 22.27. Restoring division forms the quotient from the digits 0 and 1 so that no signed digit to binary conversion is required (as it is for nonrestoring division). For all of the digit recurrence algorithms, the number of steps is proportional to the number of quotient digits.

The more advanced SRT division process for radix-2 and higher radices is similar to nonrestoring division in that the recurrence of Equation 22.25 is used, but the set of allowable quotient digits is increased. The process of quotient digit selection is sufficiently complex that a research monograph has been devoted to SRT division [Ercegovac and Lang 1994].

22.3.4.2 Newton–Raphson Divider

A second division algorithm uses a form of Newton–Raphson iteration to derive a quadratically convergent approximation to the reciprocal of the divisor, which is then multiplied by the dividend to produce the quotient. In systems that include a fast multiplier, this process is often faster than conventional division [Ferrari 1967].

$N = 0.625$

$D = 0.75$

$R_{(0)} = 2.915 - 2 \cdot D$ — 1 Shift, 1 Subtract
$= 2.915 - 2 \cdot .75$
$R_{(0)} = 1.415$

$R_{(1)} = R_{(0)}\,(2 - D \cdot R_{(0)})$ — 2 Multiplys, 1 Subtract
$= 1.415\,(2 - .75 \cdot 1.415)$
$= 1.415 \cdot .93875$
$R_{(1)} = 1.32833125$

$R_{(2)} = R_{(1)}\,(2 - D \cdot R_{(1)})$ — 2 Multiplys, 1 Subtract
$= 1.32833125\,(2 - .75 \cdot 1.32833125)$
$= 1.32833125 \cdot 1.00375156$
$R_{(2)} = 1.3333145677$

$R_{(3)} = R_{(2)}\,(2 - D \cdot R_{(2)})$ — 2 Multiplys, 1 Subtract
$= 1.3333145677\,(2 - .75 \cdot 1.3333145677)$
$= 1.3333145677 \cdot 1.00001407$
$R_{(3)} = 1.3333333331$

$Q = N \cdot R_{(3)}$ — 1 Multiply
$= 0.625 \cdot 1.3333333331$
$Q = 0.83333333319$

FIGURE 22.12 Example of Newton–Raphson division.

The Newton–Raphson division algorithm to compute $Q = N/D$ consists of three basic steps:

1. Calculate a starting estimate of the reciprocal of the divisor $R_{(0)}$. If the divisor D is normalized (i.e., $\frac{1}{2} \le D < 1$), then $R_{(0)} = 3 - 2D$ exactly computes $1/D$ at $D = 0.5$ and $D = 1$ and exhibits maximum error (of approximately 0.17) at $D = \sqrt{\frac{1}{2}}$. Adjusting $R_{(0)}$ downward to by half the maximum error gives:

$$R_{(0)} = 2.915 - 2D \tag{22.28}$$

   This produces an initial estimate that is within about 0.087 of the correct value for all points in the interval $\frac{1}{2} \le D < 1$.
2. Compute successively more accurate estimates of the reciprocal by the following iterative procedure:

$$R_{(i+1)} = R_{(i)}(2 - DR_{(i)}) \qquad \text{for } i = 0, 1, \ldots, k \tag{22.29}$$

3. Compute the quotient by multiplying the dividend times the reciprocal of the divisor,

$$Q = NR_{(k)} \tag{22.30}$$

   where i is the iteration count and N is the numerator. Figure 22.12 illustrates the operation of the Newton–Raphson algorithm. For this example, three iterations (one shift, four subtractions, and seven multiplications) produce an answer accurate to nine decimal digits (approximately 30 bits).

With this algorithm, the error decreases quadratically, so that the number of correct bits in each approximation is roughly twice the number of correct bits on the previous iteration. Thus, from a $3\frac{1}{2}$-bit initial approximation, two iterations produce a reciprocal estimate accurate to 14 bits, four iterations produce a

reciprocal estimate accurate to 56 bits, etc. The number of iterations is proportional to the logarithm of the number of accurate quotient digits.

The efficiency of this process is dependent on the availability of a fast multiplier, since each iteration of Equation 22.29 requires two multiplications and a subtraction. The complete process for the initial estimate, three iterations, and the final quotient determination requires a shift, four subtractions, and seven multiplications to produce a 16-bit quotient. This is faster than a conventional nonrestoring divider if multiplication is roughly as fast as addition, a condition that is usually satisfied for systems that include hardware multipliers.

22.4 Floating Point Arithmetic

Advances in VLSI have increased the feasibility of hardware implementations of floating point arithmetic units. The main advantage of floating point arithmetic is that its wide dynamic range virtually eliminates overflow for most applications.

22.4.1 Floating Point Number Systems

A floating point number, A, consists of a signed significand, S_a, and an exponent, E_a. The value of a number, A, is given by the equation

$$A = S_a r^{E_a} \tag{22.31}$$

where r is the radix (or base) of the number system. The significand is generally normalized by requiring that the most significant digit be nonzero. Use of the binary radix (i.e., $r = 2$) gives maximum accuracy but may require more frequent normalization than higher radices.

The IEEE Standard 754 single precision (32-bit) floating point format, which is widely implemented, has an 8-bit biased integer exponent, which ranges from 0 to 255 [IEEE 1985]. The exponent is expressed in excess 127 code, which means that its effective value is determined by subtracting 127 from the stored value. Thus, the range of effective values of the exponent is −126 to 127, corresponding to stored values of 1 to 254, respectively. A stored exponent value of 0 ($E_{\min}$) (effective value = −127) serves as a flag for 0 (if the significand is 0) and for denormalized numbers (if the significand is nonzero). A stored exponent value of 255 ($E_{\max}$) (effective value = 128) serves as a flag for infinity (if the significand is 0) and for "Not A Number"(NAN) (if the significand is nonzero). The significand is a 25-bit sign magnitude mixed number (the binary point is to the right of the most significant bit, which is always a 1 except for denormalized numbers). More details on floating point formats and on the various considerations that arise in the implementation of floating point arithmetic units are given in Gosling [1980], Goldberg [1990], Parhami [2000], Flynn and Oberman [2001], and Koren [2002].

22.4.1.1 Floating Point Addition

A flowchart for a floating point addition algorithm is shown in Figure 22.13. For this flowchart (and those that follow for multiplication and division), the significands are assumed to have magnitudes in the range [1, 2).

On the flowchart, the operands are (E_a, S_a) and (E_b, S_b), and the result is (E_s, S_s). In step 1, the operand exponents are compared; if they are unequal, the significand of the number with the smaller exponent is shifted right in either step 3 or step 4 by the difference in the exponents to properly align the significands. For example, to add 0.867×10^5 and 0.512×10^4, the latter would be shifted right and 0.867 added to 0.0512 to give a sum of 0.9182×10^5. The addition of the significands is performed in step 5. Steps 6 through 8 test for overflow and correct if necessary by shifting the significand one position to the right and incrementing the exponent. Step 9 tests for a zero significand. The loop of steps 10 and 11 scales unnormalized (but non-0) significands upward to normalize the significand. Step 12 tests for underflow.

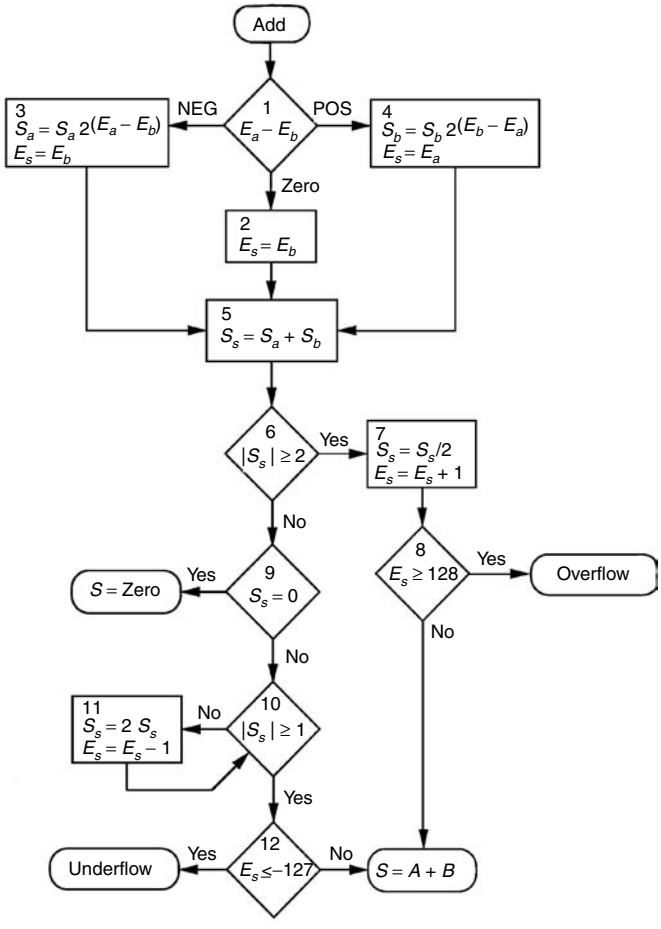


FIGURE 22.13 Floating point addition.

Floating point subtraction is implemented with a similar algorithm. Many refinements are possible to improve the speed of the addition and subtraction algorithms, but floating point addition will, in general, be much slower than fixed point addition as a result of the need for preaddition alignment and postaddition normalization.

22.4.1.2 Floating Point Multiplication

The algorithm for floating point multiplication is shown in the flowchart of Figure 22.14. In step 1, the product of the operand significands and the sum of the operand exponents are computed. Steps 2 and 3 normalize the significand if necessary. For radix-2 floating point numbers, if the operands are normalized, at most a single shift is required to normalize the product. Step 4 tests the exponent for overflow. Finally, step 5 tests for underflow.

22.4.1.3 Floating Point Division

The floating point division algorithm is shown in the flowchart of Figure 22.15. The quotient of the significands and the difference of the exponents are computed in step 1. The quotient is normalized (if necessary) in steps 2 and 3 by shifting the quotient significand while the quotient exponent is adjusted appropriately. For radix-2, if the operands are normalized, only a single shift is required to normalize the quotient. The computed exponent is tested for underflow in step 4. Finally, the fifth step tests for overflow.

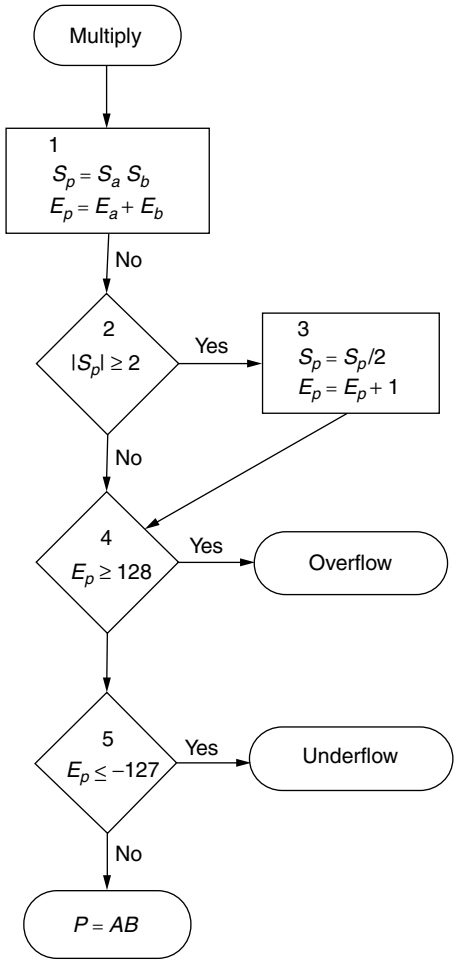


FIGURE 22.14 Floating point multiplication.

22.4.1.4 Floating Point Rounding

All floating point algorithms may require rounding to produce a result in the correct format. A variety of alternative rounding schemes have been developed for specific applications. Round to nearest, round toward ∞, round toward $-\infty$, and round toward 0 are all available in implementations of the IEEE floating point standard.

22.5 Conclusion

This chapter has presented an overview of binary number systems, algorithms for the basic integer arithmetic operations, and a brief introduction to floating point operations. When implementing arithmetic units there is often an opportunity to optimize both the performance and the area for the requirements of the specific application. In general, faster algorithms require either more area or more complex control: it is often useful to use the fastest algorithm that will fit the available area.

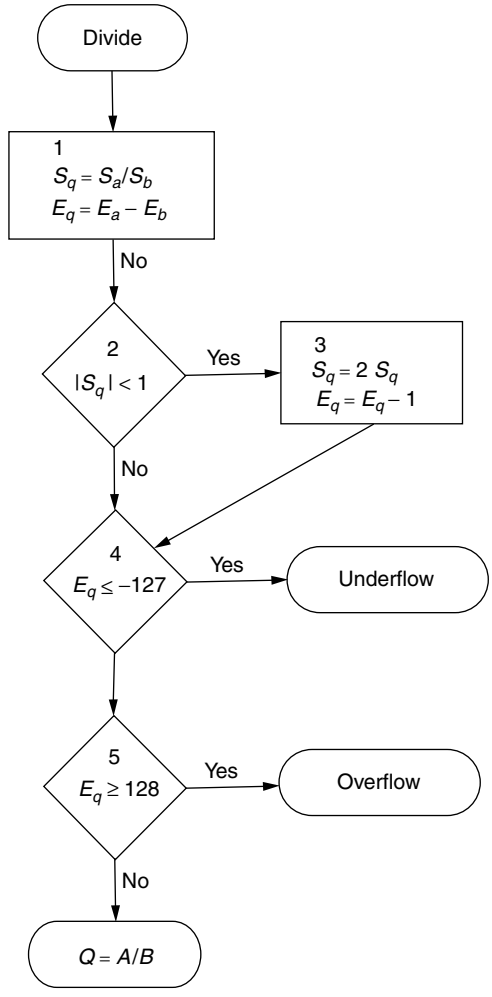


FIGURE 22.15 Floating point division.

Acknowledgments

Revision of chapter originally presented in Swartzlander, E.E., Jr. 1992. Computer arithmetic. In *Computer Engineering Handbook*. C.H. Chen, Ed., Ch. 4, pp. 4-1–4-20. McGraw–Hill, New York. With permission.

References

Baugh, C.R. and Wooley, B.A. 1973. A two's complement parallel array multiplication algorithm. *IEEE Trans. Comput.*, C-22:1045–1047.

Blankenship, P.E. 1974. Comments on "A two's complement parallel array multiplication algorithm." *IEEE Trans. Comput.*, C-23:1327.

Booth, A.D. 1951. A signed binary multiplication technique. *Q. J. Mech. Appl. Math.*, 4(Pt. 2):236–240.

Chen, P.K. and Schlag, M.D.F. 1990. Analysis and design of CMOS Manchester adders with variable carry skip. *IEEE Trans. Comput.*, 39:983–992.

Dadda, L. 1965. Some schemes for parallel multipliers. *Alta Frequenza*, 34:349–356.

Ercegovac, M.D. and Lang, T. 1994. *Division and Square Root: Digit-Recurrence Algorithms and Their Implementations*. Kluwer Academic, Boston, MA.

Ferrari, D. 1967. A division method using a parallel multiplier. *IEEE Trans. Electron. Comput*., EC-16:224–226.

Flynn, M.J. and Oberman, S.F. 2001. *Advanced Computer Arithmetic Design*. John Wiley & Sons, New York.

Goldberg, D. 1990. Computer arithmetic (App. A). In *Computer Architecture: A Quantitative Approach*, D.A. Patterson and J.L. Hennessy, Eds. Morgan Kauffmann, San Mateo, CA.

Gosling, J.B. 1980. *Design of Arithmetic Units for Digital Computers*. Macmillan, New York.

IEEE. 1985. *IEEE Standard for Binary Floating-Point Arithmetic*. IEEE Std 754-1985, Reaffirmed 1990. IEEE Press, New York.

Kilburn, T. , Edwards, D.B.G., and Aspinall, D. 1960. A parallel arithmetic unit using a saturated transistor fast-carry circuit. *Proc. IEE*, 107(Pt. B):573–584.

Koren, I. 2002. *Computer Arithmetic Algorithms*. 2*nd* edition, A.K. Peters, Natick, MA.

MacSorley, O.L. 1961. High-speed arithmetic in binary computers. *Proc. IRE*, 49:67–91.

Parhami, B. 2000. *Computer Arithmetic: Algorithms and Hardware Design*. Oxford University Press, New York.

Robertson, J.E. 1958. A new class of digital division methods. *IEEE Trans. Electron. Comput*., EC-7:218–222.

Shaw, R.F. 1950. Arithmetic operations in a binary computer. *Rev. Sci. Instrum*., 21:687–693.

Turrini, S. 1989. Optimal group distribution in carry-skip adders. In *Proc. 9th Symp. Computer Arithmetic*, pp. 96–103. IEEE Computer Society Press, Los Alamitos, CA.

Wallace, C.S. 1964. A suggestion for a fast multiplier. *IEEE Trans. Electron. Comput*. EC-13:14–17.

Waser, S. and Flynn, M.J. 1982. *Introduction to Arithmetic for Digital Systems Designers*. Holt, Rinehart and Winston, New York.

Weinberger, A. and Smith, J.L. 1958. A logic for high-speed addition. *Nat. Bur. Stand. Circular 591*, pp. 3–12. National Bureau of Standards, Washington, D.C.

23

Parallel Architectures

Michael J. Flynn
Stanford University

Kevin W. Rudd
Intel, Inc.

23.1 Introduction

Parallel or concurrent operation has many different forms within a computer system. Multiple computers can be executing pieces of the same program in parallel, or a single computer can be executing multiple instructions in parallel, or some combination of the two. Parallelism can arise at a number of levels: task level, instruction level, or some lower machine level. The parallelism may be exhibited in space with multiple independently functioning units, or in time, where a single function unit is many times faster than several instruction-issuing units. This chapter attempts to remove some of the complexity regarding parallel architectures (unfortunately, there is no hope of removing the complexity of programming some of these architectures, but that is another matter).

With all the possible kinds of parallelism, a framework is needed to describe particular instances of parallel architectures. One of the oldest and simplest such structures is the stream approach [Flynn 1966] that is used here as a basis for describing developments in parallel architecture. Using the stream model, different architectures will be described and the defining characteristics for each architecture will be presented. These characteristics provide a qualitative feel for the architecture for high-level comparisons between different processors — they do not attempt to characterize subtle or quantitative differences that, while important, do not provide a significant benefit in a larger view of an architecture.

23.2 The Stream Model

A parallel architecture has, or at least appears to have, multiple interconnected **processor elements** (PE in Figure 23.1) that operate concurrently, solving a single overall problem. Initially, the various parallel architectures can be described using the *stream* concept. A stream is simply a sequence of objects or

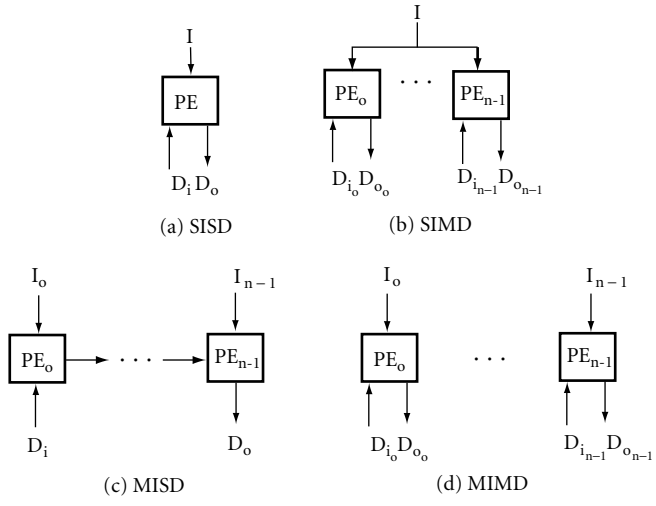


FIGURE 23.1 The stream model.

actions. Since there are both instruction streams and data streams (I and D in Figure 23.1), there are four combinations that describe most familiar parallel architectures:

1. SISD — single instruction, single data stream. This is the traditional uniprocessor [Figure 23.1a].
2. SIMD — single instruction, multiple data stream. This includes vector processors as well as massively parallel processors [Figure 23.1b].
3. MISD — multiple instruction, single data stream. These are typically systolic arrays [Figure 23.1c].
4. MIMD — multiple instruction, multiple data stream. This includes traditional multiprocessors as well as newer work in the area of networks of workstations [Figure 23.1d].

The stream description of architectures uses as its reference point the programmer's view of the machine. If the processor architecture allows for parallel processing of one sort or another, then this information must be visible to the programmer at some level for this reference point to be useful.

An additional limitation of the stream categorization is that, while it serves as a useful shorthand, it ignores many subtleties of an architecture or an implementation. Even an SISD processor can be highly parallel in its execution of operations. This parallelism is typically not visible to the programmer even at the assembly language level, but it becomes visible at execution time with improved performance.

There are many factors that determine the overall effectiveness of a parallel processor organization, and hence its eventual speedup when implemented. Some of these, including networks of interconnected streams, will be touched upon in the remainder of this chapter. The characterizations of both processors and networks are complementary to the stream model and, when coupled with the stream model, enhance the qualitative understanding of a given processor configuration.

23.3 SISD

The SISD class of processor architectures is the most familiar class — it can be found in video games, home computers, engineering workstations, and mainframe computers. From the reference point of the assembly language programmer there is no parallelism in an SISD organization, yet a good deal of concurrency can be present. **Pipelining** is an early technique that is used in almost all current processor implementations. Other techniques aggressively exploit parallelism in executing code whether it is declared statically or determined dynamically from an analysis of the code stream.

FIGURE 23.2 Canonical pipeline.

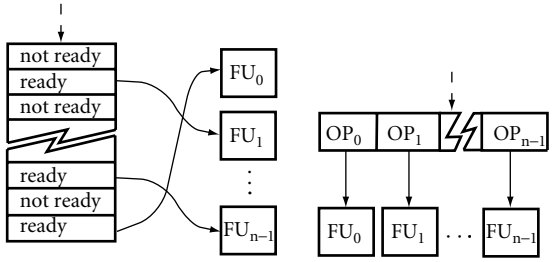


FIGURE 23.3 Instruction issue for superscalar processors.

Pipelining is a straightforward approach to exploiting parallelism that is based on concurrently performing different phases of processing an instruction. These phases often include fetching an **instruction** from memory, decoding an instruction to determine its **operation** and **operands**, accessing its operands, performing the computation specified by the operation, and storing the computed value (IF, DE, RF, EX, and WB, respectively, in Figure 23.2). Pipelining assumes that these phases are independent between different operations and can be overlapped — when this condition does not hold, the processor stalls the downstream phases to enforce the dependency. Thus multiple operations can be processed simultaneously with each operation at a different phase of its processing. For a simple pipelined machine, only one operation is in each phase at any given time — thus one operation is being fetched, one operation is being decoded, one operation is accessing operands, one operation is executing, and one operation is storing results. With this scheme, assuming that each phase takes a single cycle to complete (which is not always the case), a pipelined processor could achieve a speedup of five over a traditional nonpipelined processor design.

Unfortunately, it can be argued that pipelining does not achieve true concurrency but that it only eliminates (in the limit) the overhead associated with instruction processing. With this viewpoint, the only phase of instruction processing that is not overhead is the evaluation of the result — everything else just supports this evaluation. Even so, there is still a speedup of five over a nonpipelined processor, but this speedup serves only to bring the maximum execution rate up from one operation every five cycles to one operation every cycle (for the pipeline just described). This is the best performance that can be achieved by a processor without true parallel execution.

While pipelining does not necessarily lead to achieving true concurrency, there are other techniques that do. These techniques use some combination of static scheduling and dynamic analysis to perform concurrently the actual evaluation phase of several different operations — potentially yielding an execution rate of greater than one operation every cycle. This kind of parallelism exploits concurrency at the computation level (in contrast to pipelining, which exploits concurrency at the overhead level). Since historically most instructions consist of only a single operation (and thus a single computation), this kind of parallelism has been named *instruction-level parallelism* (ILP).

Two architectures that exploit ILP — **superscalar** and **VLIW** (very long instruction word) — use radically different techniques to achieve more than one operation per cycle. A superscalar processor dynamically examines the instruction stream to determine which operations are independent and can be executed concurrently. Figure 23.3a shows the issue of ready instructions from a window of available instructions and the routing of these instructions to the appropriate function units (FU). A VLIW processor depends on the compiler to analyze the available operations (OP) and to schedule independent operations into wide instruction words; it then executes these operations in parallel with no further analysis. Figure 23.3b shows the issue of

TABLE 23.1 Typical Scalar Processors

Processor	Year of Introduction	Number of Function Units	Issue Width	Scheduling	Issue/Complete Order
Intel x86	1978	2	1	Dynamic	In-order/In-order
Stanford MIPS-X	1981	1	1	Dynamic	In-order/In-order
Berkeley RISC-I	1981	1	1	Dynamic	In-order/In-order
Sun SPARC	1987	2	1	Dynamic	In-order/In-order
MIPS R3000	1988	2	1	Dynamic	In-order/In-order
MIPS R4000	1992	2	1	Dynamic	In-order/In-order
HP PA-RISC 1.1	1992	1	1	Dynamic	In-order/In-order

operations from a wide instruction word to all function units in parallel. In the superscalar processor, even if two operations have been determined to be independent by the compiler and are scheduled properly for the current processor, at execution time the dependency analysis must still be performed to ensure that the proper ordering is maintained. In the VLIW processor, since the compiler is depended upon to ensure the proper scheduling of operations, any operations that are improperly scheduled result in indeterminate (and probably bad!) results. Considering the programmer's reference point, the kind of parallelism that superscalar processors exploit is invisible, while the kind of parallelism that VLIW processors exploit is visible only at the assembly level where the explicit packing of multiple operations into instructions is visible — the high-level language programmer in both cases is isolated from the machine-exploited ILP. Actually, these statements are true only in the general sense; for the superscalar processor, the assembly language programmer may be aware of the organization and characteristics of the machine and be able to schedule instructions so that they can be executed in parallel by the processor (this scheduling is usually performed by the compiler, although there are assemblers which perform minor scheduling transformations). For the VLIW processor, the assembly language programmer must be aware of the specific characteristics of the machine to ensure the proper scheduling of operations (the assembler could perform the analysis and scheduling although this is typically not desired by the programmer). For both processors, even at the high-level language level there are ways of writing programs that make it easy for the compiler to find the latent parallelism.

Both superscalar and VLIW use the same compiler techniques to achieve their superscalar performance. However, a superscalar processor is able to execute code scheduled for any instruction-compatible processor while a VLIW processor can only execute code that was specifically scheduled for execution on that particular processor.* This flexibility is not for free. While a superscalar processor does execute inappropriately scheduled code, the achieved performance can be significantly worse than if it were appropriately scheduled. Nevertheless, the flexibility is an important feature in a marketplace with a significant investment in software where binary compatibility is more important than raw performance.

A SISD processor has four defining characteristics. The first characteristic is whether or not the processor is capable of executing multiple operations concurrently. The second characteristic is the mechanism by which operations are scheduled for execution — statically at compile time, dynamically at execution, or possibly both. The third characteristic is the order in which operations are issued and retired relative to the original program order — these can be in order or out of order. The fourth characteristic is the manner in which exceptions are handled by the processor — precise, imprecise, or a combination. This last characteristic is not of immediate concern to the applications programmer, although it is certainly important to the compiler writer or operating system programmer, who must be able to properly handle exceptional conditions. Most processors implement precise exceptions, although a few high-performance architectures allow imprecise floating-point exceptions (with the ability to select between precise and imprecise exceptions).

Tables 23.1, 23.2, and 23.3 present some representative (pipelined) scalar and super-scalar (both superscalar and VLIW) processor families. As Table 23.1 and Table 23.2 show, the trend has been from a scalar

*This does not have to be true — although past VLIW processors have had this restriction, this has been due to engineering decisions in the implementation and not to anything inherent in the specification. For two variant approaches to providing support for dynamic scheduling in VLIW processors see Rau [1993] and Rudd [1994].

TABLE 23.2 Typical Superscalar Processors

Processor	Year of Introduction	Number of Function Units	Issue Width	Scheduling	Issue/Complete Order
DEC 21064	1992	4	2	Dynamic	In-order/In-order
Sun UltraSPARC	1992	9	4	Dynamic	In-order/Out-of-order
MIPS R8000	1994	6	4	Dynamic	In-order/In-order
DEC 21164	1994	4	2	Dynamic	In-order/In-order
Motorola PowerPC 620	1995	6	4	Dynamic	In-order/Out-of-order
HP PA-RISC 8000	1995	10	4	Dynamic	Out-of-order/Out-of-order
MIPS R10000	1995	5	5	Dynamic	Out-of-order/Out-of-order
Intel Pentium Pro	1996	5	3	Dynamic	In-order x86 Out-of-order uops/Out-of-order
AMD K5	1996	6	4	Dynamic	In-order x86 Out-of-order ROPs/Out-of-order

TABLE 23.3 Typical VLIW Processors

Processor	Year of Introduction	Number of Function Units	Issue Width	Scheduling	Issue/Complete Order
Multiflow Trace 7/200	1987	7	7	Static	In-order/In-order
Multiflow Trace 28/200	1987	28	28	Static	In-order/In-order
Cydrome Cydra 5	1987	7	7	Static	In-order/In-order
Philips TM-1	1996	27	5	Static	In-order/In-order

to a compatible superscalar processor (except for the DEC Alpha and the IBM/Motorola/Apple PowerPC processors, which were designed from the ground up to be capable of superscalar performance). There have been very few VLIW processors to date, although advances in compiler technology may cause this to change. Philips has explored VLIW processors internally for years, and the TM-1 is the first of a planned series of processors. After the demise of both Multiflow and Cydrome, HP acquired both the technology and some of the staff of these companies and has continued research in VLIW processors.

23.4 SIMD

The SIMD class of processor architectures includes both array and vector processors. The SIMD processor is a natural response to the use of certain regular data structures, such as vectors and matrices. From the reference point of an assembly language programmer, programming an SIMD architecture appears to be very similar to programming a simple SISD processor except that some operations perform computations on aggregate data. Since these regular structures are widely used in scientific programming, the SIMD processor has been very successful in these environments.

Two types of SIMD processor will be considered: the **array processor** and the **vector processor**. They differ both in their implementations and in their data organizations. An array processor consists of many interconnected processor elements that each have their own local memory space. A vector processor consists of a single processor that references a single global memory space and has special function units that operate specifically on vectors.

23.4.1 Array Processors

The array processor is a set of parallel processor elements (typically hundreds to tens of thousands) connected via one or more networks (possibly including local and global interelement data and control communications). Processor elements operate in lockstep in response to a single broadcast instruction

TABLE 23.4 Typical Array Processors

Processor	Year of Introduction	Memory Model	Processor Element	Number of Processors
Burroughs BSP	1979	Shared	General purpose	16
Thinking Machines CM-1	1985	Distributed	Bit-serial	Up to 65,536
Thinking Machines CM-2	1987	Distributed	Bit-serial	4,096–65,536
MasPar MP-1	1990	Distributed	Bit-serial	1,024–16,384

from a control processor. Each processor element has its own private memory, and data are distributed across the elements in a regular fashion that is dependent on both the actual structure of the data and also the computations to be performed on the data. Direct access to global memory or another processor element's local memory is expensive (although scalar values can be broadcast along with the instruction), so intermediate values are propagated through the array through local interelement connections. This requires that the data are distributed carefully so that the routing required to propagate these values is simple and regular. It is sometimes easier to duplicate data values and computations than it is to effect a complex or irregular routing of data between processor elements.

Since instructions are broadcast, there is no means local to a processor element of altering the flow of the instruction stream; however, individual processor elements can conditionally disable instructions based on local status information — these processor elements are idle when this condition occurs. The actual instruction stream consists of more than a fixed stream of operations — an array processor is typically coupled to a general-purpose control processor that provides both scalar operations (that operate locally within the control processor) as well as array operations (that are broadcast to all processor elements in the array). The control processor performs the scalar sections of the application, interfaces with the outside world, and controls the flow of execution; the array processor performs the array sections of the application as directed by the control processor.

A suitable application for use on an array processor has several key characteristics: a significant amount of data which has a regular structure; computations on the data which are uniformly applied to many or all elements of the data set; simple and regular patterns relating the computations and the data. An example of an application that has these characteristics is the solution of the Navier–Stokes equations, although any application that has significant matrix computations is likely to benefit from the concurrent capabilities of an array processor.

The programmer's reference point for an array processor is typically the high-level language level — the programmer is concerned with describing the relationships between the data and the computations, but is not directly concerned with the details of scalar and array instruction scheduling or the details of the interprocessor distribution of data within the processor. In fact, in many cases the programmer is not even concerned with size of the array processor. In general, the programmer specifies the size and any specific distribution information for the data, and the compiler maps the implied virtual processor array onto the physical processor elements that are available and generates code to perform the required computations. Thus, while the size of the processor is an important factor in determining the performance that the array processor can achieve, it is not a defining characteristic of an array processor.

The primary defining characteristic of a SIMD processor is whether the memory model is shared or distributed. In this chapter, only processors using a distributed memory model are described, since this is the configuration used by SIMD processors today and the cost of scaling a shared-memory SIMD processor to a large number of processor elements would be prohibitive. Processor element and network characteristics are also important in characterizing a SIMD processor, and these are described in Section 23.2 and Section 23.6.

There have not been a significant number of SIMD architectures developed, due to a limited application base and market requirement. Table 23.4 shows several representative architectures.

23.4.2 Vector Processors

A vector processor is a single processor that resembles a traditional SISD processor except that some of the function units (and registers) operate on vectors — sequences of data values that are seemingly operated

on as a single entity. These function units are deeply pipelined and have a high clock rate; while the vector pipelines have as long or longer latency than a normal scalar function unit, their high clock rate and the rapid delivery of the input vector data elements results in a significant throughput that cannot be matched by scalar function units.

Early vector processors processed vectors directly from memory. The primary advantage of this approach was that the vectors could be of arbitrary lengths and were not limited by processor resources; however, the high startup cost, limited memory system bandwidth, and memory system contention proved to be significant limitations. Modern vector processors require that vectors be explicitly loaded into special vector registers and stored back into memory — the same course that modern scalar processors have taken for similar reasons. However, since vector registers can rapidly produce values for or collect results from the vector function units and have low startup costs, modern register-based vector processors achieve significantly higher performance than the earlier memory-based vector processors for the same implementation technology.

Modern processors have several features that enable them to achieve high performance. One feature is the ability to concurrently load and store values between the vector register file and main memory while performing computations on values in the vector register file. This is an important feature, since the limited length of vector registers requires that vectors that are longer than the registers be processed in segments — a technique called strip mining. Not being able to overlap memory accesses and computations would pose a significant performance bottleneck.

Just like their SISD cousins, vector processors support a form of result bypassing — in this case called chaining — which allows a follow-on computation to commence as soon as the first value is available from the preceding computation. Thus, instead of waiting for the entire vector to be processed, the follow-on computation can be significantly overlapped with the preceding computation that it is dependent on. Sequential computations can be efficiently compounded and behave as if they were a single operation with a total latency equal to the latency of the first operation with the pipeline and chaining latencies of the remaining operations but none of the startup overhead that would be incurred without chaining. For example, division could be synthesized by chaining a reciprocal with a multiply operation. Chaining typically works for the results of load operations as well as normal computations. Most vector processors implement some form of chaining.

A typical vector processor configuration might have a vector register file, one vector addition unit, one vector multiplication unit, and one vector reciprocal unit (used in conjunction with the vector multiplication unit to perform division); the vector register file contains multiple vector registers (eight registers with 64 double-precision floating-point values is typical). In addition to the vector registers there are also a number of auxiliary and control registers, the most important of which is the vector length register. The vector length register contains the length of the vector (or the loaded subvector if the full vector length is longer than the vector register itself) and is used to control the number of elements processed by vector operations — there is no reason to perform computations on nondata that are useless or could cause an exception.

As with the array processor, the programmer's reference point for a vector machine is the high-level language. In most cases, the programmer sees a traditional SISD machine; however, since vector machines excel on vectorizable loops, the programmer can often improve the performance of the application by carefully coding the application — in some cases explicitly writing the code to perform strip mining — and by providing hints to the compiler that help to locate the vectorizable sections of the code. This situation is purely an artifact of the fact that the programming languages are scalar-oriented and do not support the treatment of vectors as an aggregate data type but only as a collection of individual values. As languages are defined (such as Fortran 90 or High Performance Fortran) that make vectors a fundamental data type, then the programmer is exposed less to the details of the machine and to its SIMD nature.

The vector processor has one primary characteristic. This characteristic is the location of the vectors — vectors can be memory- or register-based. There are many features that vector processors have that are not included here due to their number and many variations. These include variations on chaining, masked vector operations based on a Boolean mask vector, indirectly addressed vector operations (scatter/gather), compressed/expanded vector operations, reconfigurable register files, multiprocessor support, etc.

TABLE 23.5 Typical Vector Processors

Processor	Year of Introduction	Memory-or Register-Based	Number of Processor Units	Maximum Vector Length	Number of Vector Units
Cray 1	1976	Register	1	64	7
CDC Cyber 205	1981	Memory	1	65,535	3–5
Cray X-MP	1982	Register	1–4	64	8
Hitachi HITAC S-810	1984	Register	1		4–8
Fujitsu FACOM VP-100/200[a]	1985	Register	3	32–1024	4
Cray 2	1985	Register	5	64	4
ETA ETA	1987	Memory	2–8	65,535	3–5
Cray C90		Register		64	
NEC SX-3	1990	Register	1–4		4–16
NEC SX-4	1994	Register	1–512		4–8
Cray T90	1995		1–32		2

[a]Sold as the Amdahl VP1100/VP1200 in the United States.

Vector processors have developed dramatically from simple memory-based vector processors to modern multiple-processor vector processors that exploit both SIMD vector and MIMD-style processing. Table 23.5 shows some representative vector processors.

23.5 MISD

While it is easy to both envision and design MISD processors, there has been little interest in this type of parallel architecture. The reason, so far anyway, is that there are no ready programming constructs that easily map programs into the MISD organization.

Abstractly, the MISD can be represented as multiple independently executing function units operating on a single stream of data, forwarding results from one function unit to the next. On the microarchitecture level, this is exactly what the vector processor does. However, in the vector pipeline the operations are simply fragments of an assembly-level operation, as distinct from being a complete operation in themselves. Surprisingly, some of the earliest attempts at computers in the 1940s could be seen as the MISD concept. They used plugboards for programs, where data in the form of a punched card were introduced into the first stage of a multistage processor. A sequential series of actions were taken where the intermediate results were forwarded from stage to stage until at the final stage a result was punched into a new card.

There are, however, more interesting uses of the MISD organization. Nakamura [1995] has pointed out the value of an MISD machine called the SHIFT machine. In the SHIFT machine, all data memory is decomposed into shift registers. Various function units are associated with each shift column. Data are initially introduced into the first column and are shifted across the shift-register memory. In the SHIFT-machine concept, data are regularly shifted from memory region to memory region (column to column) for processing by various function units. The purpose behind the SHIFT machine is to reduce memory latency. In a traditional organization, any function unit can access any region of memory and the worst-case delay path for accessing memory must be taken into account. In the SHIFT machine, we must allow for access time only to the worst element in a data column. The memory latency in modern machines is becoming a major problem — the SHIFT machine has a natural appeal for its ability to tolerate this latency.

23.6 MIMD

The MIMD class of parallel architecture brings together multiple processors with some form of interconnection. In this configuration, each processor executes completely independently, although most applications require some form of synchronization during execution to pass information and data between processors.

While there is no requirement that all processor elements be identical, most MIMD configurations are homogeneous with all processor elements being identical. There have been heterogeneous MIMD configurations that use different kinds of processor elements to perform different kinds of tasks, but (with the possible exception of recent work aimed at using networked workstations as a loosely coupled MIMD configuration) these configurations have not lent themselves to general-purpose applications. We limit ourselves to homogeneous MIMD organizations in the remainder of this section.

Up to this point, the MIMD processor with its multiple processor elements interconnected by a network appears to be very similar to a SIMD array processor. This similarity is deceptive, since there is a significant difference between these two configurations of processor elements — in the array processor the instruction stream delivered to each processor element is the same, while in the MIMD processor the instruction stream delivered to each processor element is independent and specific to each processor element. Recall that in the array processor, the instruction stream for each processor element is generated by the control processor and that the processor elements operate in lockstep. In the MIMD processor, the instruction stream for each processor element is generated independently by that processor element as it executes its program. While it is often the case that each processor element is running pieces of the same program, there is no reason that different processor elements could not run different programs.

The interconnection network in both the array processor and the MIMD processor passes data between processor elements; however, in the MIMD processor it is also used to synchronize the independent execution streams between processor elements. When the memory of the processor is distributed across all processors and only the local processor element has access to it, all data sharing is performed explicitly using messages and all synchronization is handled within the message system. When the memory of the processor is shared across all processor elements, synchronization is more of a problem — certainly messages can be used through the memory system to pass data and information between processor elements, but this is not necessarily the most effective use of the system.

When communications between processor elements is performed through a shared memory address space — either global or distributed between processor elements (called distributed shared memory to distinguish it from distributed memory) — there are two significant problems that arise. The first is maintaining **memory consistency** — the programmer-visible ordering effects of memory references both within a processor element and between different processor elements. The second is **cache coherency** — the programmer-invisible mechanism to ensure that all processor elements see the same value for a given memory location. Neither of these problems is significant in SISD or SIMD array processors. In a SISD processor, there is only one instruction stream and the amount of reordering is limited, so the hardware can easily guarantee the effects of perfect memory reference ordering and thus there is no consistency problem; since a SISD processor has only one processor element, cache coherency is not applicable. In a SIMD array processor (assuming distributed memory), there is still only one instruction stream and typically no instruction reordering; since all interprocessor element communication is via messages, there is neither a consistency problem nor a coherency problem.

The memory consistency problem is usually solved through a combination of hardware and software techniques. At the processor element level, the appearance of perfect memory consistency is usually guaranteed for local memory references only — this is usually a feature of the processor element itself. At the MIMD processor level, memory consistency is often guaranteed only through explicit synchronization between processors. In this case, all nonlocal references are only ordered relative to these synchronization points (such as fences or acquire/release points). While the programmer must be aware of the limitations imposed by the ordering scheme, the added performance achieved using nonsequential ordering can be significant. Table 23.6 shows the common memory consistency schemes and a brief description of their basic characteristics ($\surd$ indicates that the given feature exists, and $\sim\surd$ indicates that a restricted form of the feature exists).

The cache coherency problem is usually solved exclusively through hardware techniques. This problem is significant because of the possibility that multiple processor elements will have copies of data in their local caches and these copies could have differing values. There are two primary techniques to maintain cache coherency. The first technique is to ensure that all processor elements are informed of any change

TABLE 23.6 Simple Categorization of Consistency Models

Model[a]	Relaxation:[b] W → R Order	W → W Order	W → RW Order	Read Other's Write Early	Read Own Write Early	Explicit Synchronization[c]
SC					√	
TSO, PC	√			~ √	√	RMW
PSO	√	√			√	RMW, fence
WO	√	√	√		√	Fence
RC	√	√	√	~ √	√	Release, acquire, nsync, RMW

[a]SC = sequential consistency, TSO = total store order, PC = processor consistency, PSO = partial store order, WO = weak order, RC = release consistency.

[b]$x \rightarrow y$ represents the relaxation of the logical ordering between the reference x and a following reference y; R = read reference, W = write reference, RW = read or write reference.

[c]RMW is an atomic read–modify–write operation. Fetch-and-add is a common example of the general Fetch-and-Φ operation. (*Source:* Based on information in Adve and Gharachorloo [1995] and used with their permission.)

TABLE 23.7 Cache Coherency Summary

Coherency Model	Protocol Type	Modification Policy	Exclusive Use State	Exclusive Write State
Write once	Invalidate	Copyback on first write		
Synapse $N + 1$	Invalidate	Copyback	√	
Berkeley	Invalidate	Copyback		√
Illinois	Invalidate	Copyback	√	
Firefly	Broadcast	Copyback private, writethrough shared	√	

to the shared memory state — these changes are broadcast throughout the MIMD processor, and each processor element monitors these changes (commonly referred to as "snooping"). The second technique is to keep track of all users of a memory address or block in a directory structure and to specifically inform each user when there is a change made to the shared memory state. In either case the result of a change can be one of two things — either the new value is provided and the local value is updated, or all other copies of the value are invalidated.

As the number of processor elements in a system increases, a directory-based system becomes significantly better, since the amount of communications required to maintain coherency is limited to only those processors holding copies of the data. Snooping is frequently used within a small cluster of processor elements to track local changes — here the local interconnection can support the extra traffic used to maintain coherency since each cluster has only a few (typically two to eight) processor elements in it. Table 23.7 shows the common cache coherency schemes and a brief description of their basic characteristics (√ indicates that the given state exists).

The primary characteristic of a MIMD processor is the nature of the memory address space — it is either separate or shared for all processor elements. The interconnection network is also important in characterizing a MIMD processor and is described in the next section. With a separate address space (distributed memory), the only means of communications between processor elements is through messages, and thus these processors force the programmer to use a message-passing paradigm. With a shared address space (shared memory), communication between processor elements is through the memory system — depending on the application needs or programmer preference, either a shared-memory or a message-passing paradigm can be used.

The implementation of a distributed-memory machine is far easier than the implementation of a shared-memory machine when memory consistency and cache coherency are taken into account. However, programming a distributed-memory processor can be much more difficult, since the applications must be written to exploit and not be limited by the use of message passing as the only form of communications between processor elements. On the other hand, despite the problems associated with maintaining

TABLE 23.8 Typical MIMD Systems

System	Year of Introduction	Processor Element	Number of Processors	Memory Distribution	Programming Paradigm	Interconnection Type
Alliant FX/2800	1990	Intel i860	4–28	Central	Shared memory	Bus + crossbar
Stanford DASH	1992	MIPS R3000	4–64	Distributed	Shared memory	Bus + mesh
CRAY T3D	1993	DEC Alpha 61064	128–2048	Distributed	Shared memory	3-D torus
MIT Alewife	1994	Sparcle	1–512	Distributed	Message passing	Mesh
Convex C4/XA	1994	Custom	1–4	Global	Shared memory	Crossbar
Thinking Machines CM-500	1995	SuperSPARC	16–2048	Distributed	Message passing	Fat tree
Tera Computers MTA	1995	Custom	16–256	Distributed	Shared memory	3-D torus
SGI Power Challenge XL	1995	MIPS R8000	2–18	Global	Shared memory	Bus
Convex SPP1200/XA	1995	HP PA-RISC 7200	8–128	Global	Shared memory	Crossbar + ring
CRAY T3E	1996	DEC Alpha 61164	16–2048	Distributed	Shared memory	3-D torus
Network of Workstations	1990s	Various	Any	Distributed	Message passing	Ethernet

consistency and coherency, programming a shared-memory processor can take advantage of whatever communications paradigm is appropriate for a given communications requirement and can be much easier to program. Both distributed and shared-memory processors can be extremely scalable and neither approach is significantly more difficult to scale than the other. Some typical MIMD systems are described in Table 23.8.

23.7 Network Interconnections

Both SIMD array processors and MIMD processors rely on networks for the transfer of data between processor elements or processors. A bus is a simple kind of network — it serves to interconnect all devices that are plugged into it — but is not commonly referred to as a network. We discuss here only the aspects of networks that are of interest in characterizing a processor — particularly the SIMD array processors and MIMD processors — and present some network characteristics that provide a qualitative sense that is useful for understanding the basic nature of a multiprocessor interconnect.

There are three primary characteristics of networks. The first is the method used to transfer the information through the network — either using packet routing or circuit switching. The second characteristic is the mechanism that connects source and destination nodes — either the connections are static and fixed or they are dynamic and reconfigurable. The third characteristic is whether the network is a single-level or a multiple-level network. While these characteristics leave out a significant amount of detail about the actual network, they qualitatively describe the network connections and how information is routed between processor elements.

Packet routing is efficient for small random packets, but it has the drawback that neither the latency nor the bandwidth is necessarily deterministic and thus packets may not be delivered in the same order that they were sent; circuit switching achieves high bandwidth for a given connection between processor nodes and guarantees uniform latency and proper receipt ordering, but it has the drawback that the latency for small packets becomes the latency for setting up and breaking down the connection.

Dynamic networks allow network reconfiguration so that there are essentially direct connections between nodes across the network, producing high bandwidth and low latency but limiting the scalability of the system; static networks improve the scalability, since connections are node to node and any two nodes can be connected either directly or through intermediate nodes, resulting in longer latency and lower-bandwidth connections. Use of multilevel networks, which use clusters of processor elements at each network node, increases the complexity of the system but reduces congestion on the global interconnect and leads to a more scalable system — intracluster communications are performed on a local interconnect that is much faster and does not leave the cluster. Single-level networks are more general but

less scalable, since all communications must use the global interconnect, and traffic can be much higher for the same number of processor elements.

23.8 Afterword

In this chapter we have reviewed a number of different parallel architectures organized by the stream model. We have described some general characteristics that offer some insight into the qualitative differences between different parallel architectures but, in the general case, provide little quantitative information about the architectures themselves — this would be a much more significant task, although there is no reason why a significantly increased level of detail could not be described. Just as the stream model is incomplete and overlapping (consider that a vector processor can be considered to be a SIMD, MISD, or SISD processor depending on the particular categorization desired), so the characteristics for each class of architecture are also incomplete and overlapping. However, the general insight gained from considering these general characteristics leads to an understanding of the qualitative differences between gross classes of computer architectures, so the characteristics that we have described provide similar benefits and liabilities.

In a sense, the characterizations that we have provided for each architectural class of processor can be considered as specializations on the stream model. Thus a superscalar processor could be described as a SISD processor which supports concurrent execution of multiple operations that are scheduled dynamically, performs issue and retire out of order, and provides precise interrupts. A similar superscalar processor that does not provide precise interrupts would be described almost identically, but the description would provide an insight into one significant difference between these two processors — the first superscalar processor would be more complicated to design and might run more slowly than the second superscalar processor, but it would support more efficient exception recovery. While this comparison does not, in all likelihood, provide sufficient information to make a design choice in many cases, it does provide a basis for processor comparison.

For a MIMD or a SIMD processor, although the primary characteristic is the characterization of the memory address space, the system can be more completely described by including the description of both the processor element (most likely a SISD processor) along with a description of any networks that are included in the processor. This results in a description of the processor that provides a more complete understanding of the system as a whole but now including much more information about the remainder of the system.

This is not meant to imply that the aggregate of the stream model along with the relevant characteristics is a complete and formal extension to the original taxonomy — far from it. There are still a wide range of processors that are problematic to describe well in this (and likely in any) framework. The example was given earlier concerning the appropriate placement of a vector processor. Another example is the proper placement of an architecture that is designed to support multiple threads on a single processor element. These processor elements could be considered to be just SISD processors which have a specialized operating system that provides this support (albeit requiring hardware support as well), but there is some reason to believe that multiple threads are a significant feature, especially in the case of the Tera MTA architecture [Alverson et al. 1990], where the threads are interleaved through the execution units on a cycle-by-cycle basis — clearly a distinct difference beyond simply performing efficient task switches.

Whatever the problems with classifying and characterizing a given architecture, processor architectures, particularly multiprocessor architectures, are developing rapidly. Much of this growth is the result of significant improvements in compiler technology that allow the unique capabilities of an architecture to be efficiently exploited. In many cases, the design of a system is based on the ability of a compiler to produce code for it. It may be that a feature is unable to be utilized if a compiler cannot exploit it and thus the feature is wasted (although perhaps the inclusion of such a feature would spur compiler development). It may also be that an architectural feature is added specifically to support a capability that a compiler readily supports and thus performance is improved. Compiler development is clearly an integral part of system design and architectural effectiveness is no longer limited only to concerns for the processor itself.

Defining Terms

Array processor: An array of processor elements operating in lockstep in response to a single instruction and performing computations on data that are distributed accross the processor elements.

Cache coherency: The programmer-invisible mechanism that ensures that all caches within a computer system have the same value for the same shared-memory address.

Instruction: Specification of a collection of operations that may be treated as an atomic entity with a guarantee of no dependencies between these operations. A typical processor uses an instruction containing one operation.

Memory consistency: The programmer-visible mechanism that guarantees that multiple processor elements in a computer system receive the same value on a request to the same shared-memory address.

Operand: Specification of a storage location — typically either a register of a memory location — that provides data to or receives data from the results of an operation.

Operation: Specification of one or a set of computations on the specified source operands placing the results in the specified destination operands.

Pipelining: The technique used to overlap stages of instruction execution in a processor so that processor resources are more efficiently used.

Processor element: The element of a computer system that is able to process a data stream (sequence) based on the content of an instruction stream (sequence). A processor element may or may not be capable of operating as a stand-alone processor.

Superscalar processor: A popular term to describe a processor that dynamically analyzes the instruction stream and attempts to execute multiple ready operations independently of their ordering within the instruction stream.

Vector processor: A computer architecture with specialized function units designed to operate very efficiently on vectors represented as streams of data.

VLIW processor: A popular term to describe a processor that performs no dynamic analysis on the instruction stream and executes operations precisely as ordered in the instruction stream.

References

Adve, V. and Gharachorloo, K. 1995. Shared memory consistency models: a tutorial. Research Report 95/7. Digital Equipment Corp. Western Research Lab.

Alverson, R., Callahan, D., Cummings, D., Koblenz, B., Porterfield, A., and Smith, B. 1990. The Tera computer system. In *Int. Conf. on Supercomputing*, ACM, New York, June.

Flynn, M. J. 1966. Very high-speed computing systems. *Proc. IEEE* 54(12):1901–1909.

Flynn, M. J. 1995. *Computer Architecture: Pipelined and Parallel Processor Design*. Jones and Bartlett, Boston.

Hennessy, J. L. and Patterson, D. A. 1996. *Computer Architecture A Quantitative Approach*, 2nd ed. Morgan Kaufmann, San Francisco.

Hockney, R. W. and Jesshope, C. R. 1988. *Parallel Computers: Architecture, Programming and Algorithms*. Adam Hilger, Bristol.

Hwang, K. 1993. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw–Hill, New York.

Ibbett, R. N. and Topham, N. P. 1989a. *Architecture of High Performance Computers. Vol. I. Uniprocessors and Vector Processors.* Springer–Verlag, New York.

Ibbett, R. N. and Topham, N. P. 1989b. *Architecture of High Performance Computers, Vol. II. Array Processors and Multiprocessor Systems.* Springer–Verlag, New York.

Lilja, D. J. 1991. *Architectural Alternatives for Exploiting Parallelism.* IEEE Computer Society Press, Los Alamitos, CA.

Nakamura, T. 1995. The SHIFT machine. Personal correspondence.

Rau, B. R. 1993. Dynamically scheduled VLIW processors, pp. 80–92. In *26th Annual International Symp. on Microarchitecture*.

Rudd, K. W. 1994. Instruction level parallel processors — a new architectural model for simulation and analysis. Technical Report CSL-TR-94-657. Stanford Univ.
Trew, A. and Wilson, G., eds. 1991. *Past, Present, Parallel: A Survey of Available Parallel Computing Systems.* Springer–Verlag, London.

Further Information

There are many good sources of information on different aspects of parallel architectures. The references for this chapter provide a selection of texts that cover a wide range of issues in this field. There are many professional journals that cover different aspects of this area either specifically or as part of a wider coverage of related areas. Some of these are:

IEEE Transactions on Computers, Transactions on Parallel and Distributed Systems, Computer, Micro.
ACM Transactions on Computer Systems, Computer Surveys.
Journal of Supercomputing.
Journal of Parallel and Distributed Computing.

There are also a number of conferences that deal with various aspects of parallel processing. The proceedings from these conferences provide a current view of research on the topic. Some of these are:

International Symposium on Computer Architecture (ISCA).
Supercomputing.
International Symposium on Microarchitecture (MICRO).
International Conference on Parallel Processing (ICPP).
International Symposium on High Performance Computer Architecture (HPCA).
Symposium on the Frontiers of Massively Parallel Computation.

24

Architecture and Networks

Robert S. Roos
Allegheny College

24.1 Introduction

The word "architecture," as it applies to computer networks, can be interpreted in at least two different ways. The overall design of a network (analogous to an architect's blueprint of a building) — including the interconnection pattern, communication media, choice of protocols, placement of network management resources, and other design decisions — is generally referred to as the network's architecture. However, we can also speak of the hardware and the hardware-specific protocols used to implement the various pieces of the network (much as we would speak of the beams, pipes, bricks, and mortar that comprise a building).

The goal of this overview is to concentrate on architectures of networks in both senses of the word, but with emphasis on the second, showing how established and relatively stable design technologies, such as multiplexing and signal modulation, are integrated with an array of hardware devices, such as routers and modems, into a physically realized network. More specifically, this chapter presents representative technologies whose general characteristics will remain applicable even as the hardware state of the art advances.

The organizing principle used in this study is the notion of a network protocol stack, defined in the next section, which provides a hierarchy of communication models that abstract away from the physical details of the network. Each of the network technologies considered in this survey is best explained in relation to a particular abstraction level in this hierarchy. At times, this classification scheme may be relaxed a bit for terminological reasons; for example, voice modems are most relevant to layer 1 of the protocol stack,

whereas cable modems are really layer 2 devices, but these modems are all discussed in the same section of the chapter. On the other hand, in a few cases, the classification is difficult to apply. The word "switch" can be applied to network hubs and telephone switches (normally associated with layer 1) or to LAN (local area network) switches (normally thought of as layer 2 devices), or to routers (implementing layer 3 functionality). Unavoidably, the discussion of switching will overlap different layers.

Other chapters of this *Handbook* deal in greater depth with architecture in its first sense. Chapter 72 deals with topics such as network topologies and differences between local area and wide area networks; Chapter 73 discusses the important subject of routing in networks.

For the most part, the technologies presented in this chapter represent standards that are recognized worldwide. Several organizations provide standards for networking hardware and software. The International Telecommunication Union (which replaced the International Telegraph and Telephone Consultative Committee, or CCITT) has a sector (ITU-T) devoted to telecommunication standards. The ITU-T has recommended standards for modems (the "V" series of recommendations) and data networks and open system communication (the "X" series, including the well-known X.25 recommendation for virtual circuit connections in packet-switching networks), multiplexing standards, and many others, some of which are described in greater detail below. The International Organization for Standardization (ISO), formerly the International Standards Organization, has produced a huge number of standards documents in many areas, including information technology. (Their recommendation for the Open Systems Interconnection (OSI) reference model is described briefly in the next section.) The Institute for Electrical and Electronics Engineers (IEEE) is responsible for the "802" series of LAN standards, most of which have been adopted by the ISO. The American National Standards Institute (ANSI) is one of the founding members of the ISO and the originator of many networking standards. Gallo and Hancock [2002] provide a concise summary of most of the important regional, national, international, and industry, trade, and professional standards organizations.

24.2 Underlying Principles

We adopt Tanenbaum's definition of a computer network as an interconnected collection of autonomous computers [Tanenbaum, 1996]. We will refer to these as hosts or stations in the following discussion. Although this definition fails to take into account many network-like structures, from "networks-on-a-chip" [Benini and De Micheli, 2002] to interconnected "information appliances" such as Web-enabled cellular telephones and portable digital assistants, it is sufficient for an introductory look at the architectures of networks.

24.2.1 LANs, WANs, MANs, and Topologies

We will often make references to local area networks (LANs) and wide area networks (WANs). The phrases are fairly self-explanatory. LANs normally connect machines in a geographical area not much larger than a building or a small cluster of buildings, while WANs could connect machines separated by tens of thousands of kilometers. The higher-level architecture of LANs and WANs is described in more detail in Chapter 72. Another category, the metropolitan area network (MAN), is often mentioned in the literature. MANs lie in between LANs and WANs, perhaps joining machines distributed over the area of a large city. Classifications such as "personal area network" (PAN), "system area network" (SAN), "desk area network" (DAN), and others have been proposed, but their meanings are not yet well-established.

Many of the most popular LAN architectures represent broadcast networks: the host machines comprising the network share a common transmission medium (such as a wire in an Ethernet LAN or a block of the electromagnetic spectrum in a wireless LAN). Each host on the network sees every transmission (whether or not it is the intended recipient), and each must contend with the others for access to this medium when it wishes to transmit. The details of such multiple access methods can influence, for example, the manner in which the physical signal is encoded (for instance, code-division multiple access

methods require each host to have a unique code word that is used to represent the bits transmitted by that host).

In some instances, particularly in the area of LAN networks, our discussion of networking devices will depend upon the network topology (the interconnection pattern of the hosts comprising the network). For example, a LAN may be organized using a bus topology (such as a single Ethernet cable), a star topology (several Ethernet lines joined at a central point), or a ring topology (such as an FDDI network). Repeaters may be needed to extend a bus further than the limits imposed by the physical properties of the medium. A hub might be used to join machines into a star configuration. Rings require some sort of switching mechanism to restore the ring topology when part of the network is damaged.

Topology is less of a concern with WANs, which present an entirely different set of equipment issues. Most wide area networks are comprised of a combination of media, ranging from copper twisted-pair wires to high-bandwidth fiber optic cables to terrestrial microwave or satellite transmission. Devices are needed to deal with the interfaces between these media and to perform routing and switching.

24.2.2 Circuit Switching and Packet Switching

Data transmission in networks can generally be classified into one of two categories. If data is conveyed as a bit stream by means of a dedicated "line" (in the form of optical fiber, copper wire, or reserved bandwidth along a transmission path), it is said to be **circuit-switched**. However, if it is subdivided into smaller units, each of which finds its way more or less independently from source to destination, it is said to be **packet-switched**. At the lowest layer of the protocol stack, where we deal only with streams of bits, we are usually not concerned with this distinction; however, at higher levels, devices such as switches must be designed according to which of the two methods is used. A technique known as **virtual circuits** combines features of both of these switching categories. In a virtual circuit, a path is set up in advance, after which data is relayed along the path using packet-switching methods. Virtual circuits can be either permanent or switched; in the latter case, each virtual circuit is preceded by a set-up phase that builds the virtual circuit and is followed by a tear-down phase.

In general, packets can vary in size (for example, IP packets can be anywhere from 20 to 65,536 bytes in length). A technology known as Asynchronous Transfer Mode (ATM) uses fixed-size packets of 53 bytes, which are usually referred to as cells. The term "packet-switching" is customarily used to describe packet-switching methods other than ATM. ATM is described in greater detail in Chapter 72; however, we will discuss some of the details of ATM switching in later sections.

24.2.3 Protocol Stacks

A **communication protocol** is a set of conventions and procedures agreed upon by two communicating parties. Protocols are usually grouped together into layers (a protocol stack); the actions required to implement a protocol at the $(i+1)st$ layer of the stack require the services of the protocols at the ith layer.

This layered approach to network design has been used in many network implementations, dating back at least to the initial proposed three-layer protocol stack for the ARPA network [Meyer 1970]. In 1983, the International Standards Organization (ISO) proposed the seven-layer Open Systems Interconnection (OSI) Reference Model for network protocol stacks. Although there have been criticisms of this model, some of which are summarized by Tanenbaum [1996], it has served as a convenient framework for discussion and comparison of protocol architectures. A summary of the ISO/OSI Reference Model sufficient for the purposes of this discussion is given in Table 24.1.

Protocols in the upper layers are classified as "end-to-end" because they provide abstractions that deal only with the source and destination nodes of a network communication; they are usually more concerned with details that are global in nature, such as establishing a connection to a remote host or determining how to proceed when data is damaged or lost in transit. Details about routing, encoding, contention for network resources, etc., are reserved for layers 3 and lower, which are termed "point-to-point" because they mediate communication between adjacent nodes in the network.

TABLE 24.1 The ISO-OSI Reference Model

Layer	Name	Remarks
7	Application	
6	Presentation	
5	Session	End-to-end
4	Transport	
3	Network	
2	Data link	Point-to-point
1	Physical	

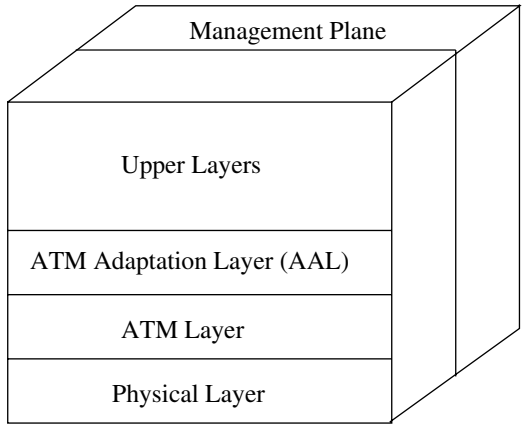


FIGURE 24.1 ATM reference model (simplified).

The protocol stack used in the Internet is often called the TCP/IP reference model, although technically it is not a model but an actual architecture. Our interest in this chapter is primarily with the lower levels of the protocol stack, and at these levels the distinctions between these two models are not critical to the discussion.

The ATM reference model is organized in a slightly different fashion, not completely analogous to the OSI stack. Figure 24.1 gives a simplified view. In addition to a vertical organization of protocols into layers, ATM specifies different *planes* that cut across the layers. The full specification includes a user plane and a control plane (not shown in the figure). The physical layer roughly corresponds to layer 1 in the OSI model, but the relationship between the ATM and AAL layers and the layers in OSI is not clear-cut. Certainly, layer 2 functionality is included in these two layers, and for the purposes of this chapter it is not necessary to make any finer distinctions.

The OSI seven-layer model does not represent the final level of granularity; most technology standards divide the OSI levels into two or more sublayers. To give just one example, the IEEE 802.3 standard for CSMA/CD networks [IEEE, 2002] subdivides the physical layer into four sublayers: PLS (physical layer signaling), AUI (attachment unit interface), MAU (medium attachment unit), and MDI (medium-dependent interface). Forouzan gives a very readable description of these sublayers [Forouzan, 2003]. In what follows, we limit ourselves to distinguishing between the logical link control and medium access control sublayers of the data-link layer and leave the remaining fine structure of this and other layers for further reading.

It is worth noting that alternatives to the layered network architecture approach have been proposed (see, for example, Braden et al. [2002]); however, this approach appears to be firmly established and unlikely to change.

24.2.4 Data Representation: Baseband Methods

There are many ways of physically representing, or keying, binary digits for transmission along some medium. To take the simplest example, consider a copper wire. We can use, for example, a voltage v_0 to represent a binary 0 and a different voltage v_1 to represent a binary 1. Although this is the most straightforward and obvious method, it suffers from several problems, most notably issues of synchronization and signal detection (a very long string of 0s or 1s, for example, would be represented as a steady voltage, making it difficult to count the bits and, if some drift in the signal value is possible, perhaps leading to the mistaken assumption that no signal is present at all). To correct this, we could use, not the voltage values themselves, but transitions between voltages (a transition from v_0 to v_1 or v_1 to v_0) to indicate bit values. This is the method used in Manchester encoding, which is the signaling method for Ethernet local area networks. Manchester encoding can be thought of as a phase modulation of a square-wave signal; phase modulation is discussed below. This encoding suffers from a different problem, namely that the transmitted signal must change values twice as fast as the bit rate, because two different voltages (low/high or high/low) are needed for each bit.

We need not restrict ourselves to just two values. If the sender and receiver are capable of distinguishing among a larger set of voltages, we could use, say, eight different voltages $v_0, \ldots, v_7$ to encode any of the eight 3-bit strings 000, 001, ..., 111. The limits of this approach are dictated by the amount of noise in the transmission medium and the sensitivity of the receiver. If just three voltages are available, say $v_0 < 0$, 0, and $v_1 > 0$, we can require the signal to return to the zero value after each bit (RZ schemes), or we can use a "non-return-to-zero" scheme (NRZ) that uses only the positive and negative values.

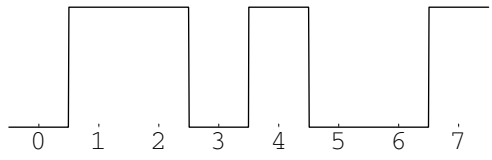
Some encoding schemes assume the existence of a clock that divides the signal into discrete time intervals; others, such as the Manchester scheme, are self-clocking because every time interval contains a transition. A compromise between these two is a block encoding scheme in which extra bits are inserted at regular intervals to provide a synchronization mechanism. One such method, called 4B5B, transmits 5 bits for every 4 bits of data; each combination of 4 bits has a corresponding 5-bit code. Fiber Distributed Data Interface (FDDI) uses a combination of 4B5B and NRZ.

Six examples of encoding schemes are shown in Figure 24.2. The straightforward binary encoding (a) would require a clocking mechanism for synchronization to deal with long constant bit strings. This can be avoided using three signal values and a return-to-zero encoding (b). The Manchester encoding (c) is an NRZ scheme because the zero is not used. A low-to-high transition signifies a binary zero; a high-to-low transition signifies a binary 1. The differential Manchester encoding (d) uses, not transitions of a particular type, but *changes* in the transition type to indicate bits — if a low-to-high transition is followed by a high-to-low transition (or vice versa), a 1 bit is indicated; but if two successive transitions are of the same type, a 0 bit is indicated. Bipolar alternate mark inversion, or AMI (e), uses alternating positive and negative values to represent the "1" bits in the signal. It, too, is subject to synchronization problems (long strings of zeros hold the signal constant, although long strings of ones are no longer a problem). Several variations of bipolar AMI counteract this effect. For example, in B8ZS, strings of eight consecutive zeros are replaced by the string "`00011011`," but in such a way that the encoding of the spurious "1"s violates the rule of sign alternation. NRZ-I (f) uses the occurrence of a signal change to indicate a 1 bit; it requires a clock or must be used in conjunction with a coding method such as 4B5B that guarantees regular changes in the signal value.

These are examples of baseband signaling — the digital signal (or an analog signal that closely approximates it) is the only thing sent. An alternative to baseband signaling is to use a carrier wave modulated by the digital signal. This process is explained in more detail below.

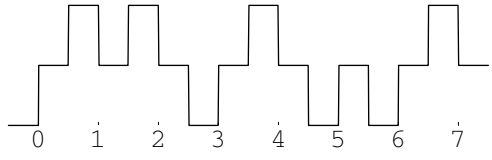
24.2.5 Data Representation: Modulation

In modulation, an underlying signal, for example, a sine wave, is used as a carrier and modifications to its amplitude, frequency, and phase are used to encode the digital information. Figure 24.3 gives a simplified view of this process, showing the same digital signal encoded three different ways: by amplitude modulation, by frequency modulation, and by phase modulation.



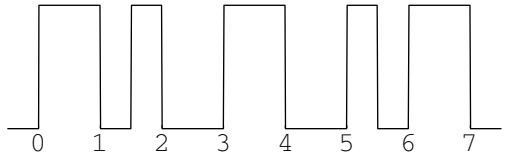
(a)

The 8-bit binary signal 01101001



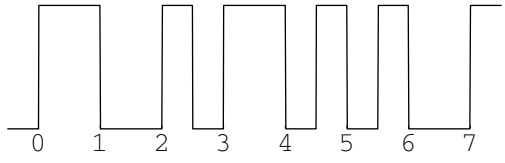
(b)

Return-to-zero (RZ) encoding



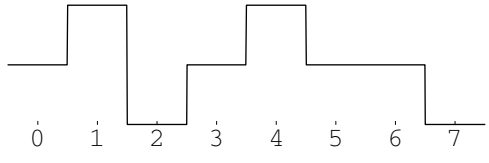
(c)

Manchester encoding — 0 is a
"low-to-high" transition, 1 is a "high-to-low"



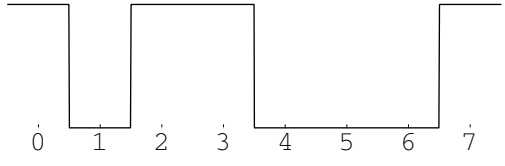
(d)

Differential Manchester encoding — 1 is represented
by a change in the transition direction



(e)

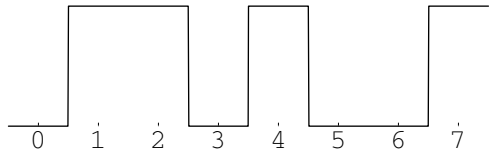
Bipolar AMI encoding — successive 1s
alternate between positive and negative



(f)

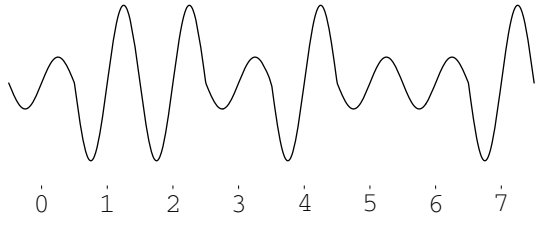
NRZ-I encoding — a 1
toggles the signal value

FIGURE 24.2 Six baseband signaling methods.



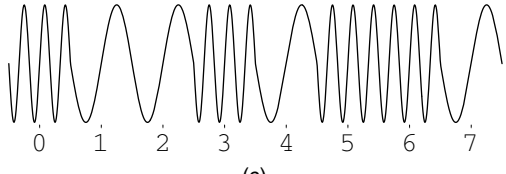
(a)

The 8-bit binary signal 01101001



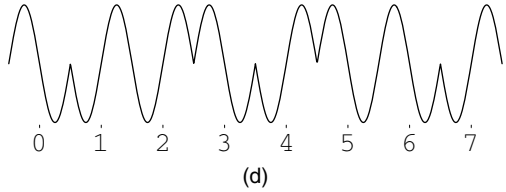
(b)

Amplitude modulation — 1 corresponds to higher amplitude than 0



(c)

Frequency modulation — 1 corresponds to lower frequency than 0



(d)

Phase modulation — 180° phase difference between 0 and 1

FIGURE 24.3 Modulation of a signal by amplitude, frequency, and phase.

It is possible to combine several types of modulation in the same signal. Furthermore, there is no reason to limit the set of possible values to simple binary choices. Using a variety of amplitudes and phases, it is possible to encode several bits in one signaling time unit (e.g., with four phases and two amplitudes, we can encode eight distinct values, or 3 bits, in one signal). This technique is called quadrature amplitude modulation, or QAM.

24.2.5.1 Multiplexing

It is often the case that many signals must be sent over a single transmission path. For a variety of reasons it makes sense to **multiplex** these signals — to merge them together before placing them on the transmission medium and separate them at the other end (the separation process is called demultiplexing). One way to do this is called time division multiplexing. Each signal is divided into segments and each segment is

allocated a certain amount of time, much as time-sharing in an operating system permits several processes to share a single processor. Another way to achieve it is to use frequency division multiplexing — different signals are allocated different frequency bands from the available bandwidth. (Standard signal processing techniques permit the receiver to discriminate among the various frequency bands used.) Wavelength division multiplexing or WDM (sometimes called *dense* WDM or DWDM) is a form of frequency division multiplexing used in optical communication and assigns different colors of the light spectrum to different signals.

Several multiplexing standards exist to facilitate connecting telephone and data networks to one another. **T-carriers** were defined by AT&T and are the standard in North America and (with slight discrepancies) Japan (where they are sometimes called J-carriers). A T1 carrier uses time-division methods to multiplex 24 voice channels together, each 8 bits wide; the data rate is 1.544 Mbps. A T2 carrier multiplexes four T1 carriers (6.312 Mbps), a T3 carrier multiplexes six T2 carriers (44.736 Mbps), and a T4 carrier multiplexes seven T3 carriers (274.176 Mbps). However, a conflicting standard, the **E-carrier**, is used in Europe and the rest of the world [ITU, 1998]. An E-1 carrier multiplexes 32 8-bit channels together and has a rate of 2.048 Mbps. E-2 and higher carriers each increase the number of channels by an additional factor of four; thus, an E-2 carrier has 120 channels (8.448 Mbps), E-3 has 480 channels (34.368 Mbps), etc.

With the increasing use of optical fiber, another set of multiplexing standards has been adopted. SONET (Synchronous Optical Network), ANSI standard T.105, and SDH (Synchronous Digital Hierarchy), developed by the ITU, are very similar and are often referred to as SONET/SDH. The basic building block, called STS-1 (for Synchronous Transport Signal), defines a synchronized sequence of 810-bit frames and a basic data rate of 51.84 Mbps. The corresponding optical carrier signal is called OC-1. STS-*n* (OC-*n*) corresponds to *n* multiplexed STS-1 signals (thus, STS-3/OC-3 corresponds to three multiplexed STS-1 signals and has a data rate of 155.52 Mbps).

Signals are sometimes multiplexed when several hosts try to simultaneously access a shared medium. In code division multiple access (CDMA), which is used in some wireless networks such as the IEEE 802.11b standard [IEEE, 1999b], each host is assigned a unique binary code word, also called a chip code or chipping code. Each `1` bit is transmitted using this code word, while a `0` bit uses the complementary bit sequence (the word "chip" suggests a fraction of a bit). If the codes satisfy certain orthogonality properties, a combination of statistical techniques and a process very similar to Fourier analysis can be used to extract a particular signal from the sum of several such signals. Chipping codes are used in Direct Sequence Spread Spectrum (DSSS) transmission; Tanenbaum [1996] and Kurose and Ross [2003] provide good introductions to CDMA.

SDMA (Spatial Division Multiple Access) is used in satellite communications. It allows two different signals to be broadcast along the same frequency but in different directions. If the recipients are sufficiently separated in space, each will receive only the intended signal.

Some wireless technologies, such as IEEE 802.11a [IEEE, 1999a] and the newly approved 54-Mbps 802.11g standard [Lawson, 2003], use a technique called ODFM (Orthogonal Frequency Division Multiplexing). Rather than combining signals from several transmitters, this variation of FDM is used for sending a single message by means of modulating multiple carriers to overcome effects such as multipath fading (in which a signal is canceled out by interference with delayed copies of itself).

24.3 Best Practices: Physical Layer Examples

24.3.1 Media

The lowest layer of the protocol stack concerns itself with simply delivering bits from one machine to another, regardless of any meaning that may be attached to the bits. Communicating a stream of bits between two computers requires a transmission medium and one or more protocols specifying the way in which the bits are represented and their rate of transmission. Typical media in modern networks include ordinary copper, usually in the form of "twisted pairs" of wires, fiber-optic cable, and wireless transmission via radio, microwave, or infrared waves. A detailed description of the physical properties of each of these

would require a separate chapter; therefore, only brief descriptions of some of the more common media are given.

Ordinary copper wire remains one of the most common and most versatile media. Occurring in the form of twisted pairs of wires or coaxial cable, copper wiring occurs in telephone wiring (particular the "last mile" connection between homes and local telephone switching offices), many local area networks such as Ethernet and ATM LANs, and other places where an inexpensive and versatile short-haul transmission medium is needed.

Optical fiber is much more expensive but has the benefits of large bandwidth and low noise, permitting larger data transfer rates over much longer distances than copper. Fiber can be either glass or plastic and transmits data in the form of light beams, generally in the low infrared region. Multi-mode fiber permits simultaneous transmission of light beams of many different wavelengths; single-mode fiber allows transmission of just one wavelength.

Wireless transmission can be at any of a number of electromagnetic frequencies, such as radio, microwave, or infrared. This overview deals primarily with guided media.

At the physical layer, among the many challenges facing network designers are signal detection, attenuation, corruption due to noise, signal drift, synchronization, and multipath fading. Standard signal processing techniques handle many of these problems; Stein [2000] gives an overview of these techniques from a computer science point of view. Some of the more interesting problems deal with interfacing transmission media (particularly issues dealing with digital vs. analog representation) and maximizing the bandwidth usage of various media. Several representative technologies for dealing with these concerns are presented in this section.

24.3.2 Repeaters and Regenerators

A repeater is essentially just a signal amplifier; the term predates computer networks, originally referring to radio signal repeaters. The term "regenerator" is sometimes used to describe a device that amplifies a binary (as opposed to an analog) signal. Repeaters are used where the transmission medium (e.g., twisted pair copper wire or microwaves) has an inherent physical range limit beyond which the signal becomes too attenuated to use. Microwave antennas serve the function of repeaters in terrestrial microwave networks; communication satellites can be thought of as repeaters that amplify electromagnetic signals and retransmit them back to the ground. Optical repeaters, used with fiber-optic cable, sometimes have additional functionality, such as converting between single-mode and multimode fiber transmission. Because repeaters are relatively simple, they introduce very little delay in the signal. Repeaters are sometimes incorporated into more sophisticated devices, such as hubs connecting several local area networks; hubs are discussed in more detail in a later section.

An amplifier will amplify noise just as well as data. More sophisticated technologies are required to ensure the integrity of the bit stream. In the particular case of a digital signal regenerator, much of the noise can be eliminated using fairly standard techniques that involve synchronizing the data signal with a clock signal and using threshold methods; see, for example, Anttalainen's [1999] discussion of regenerative repeaters.

With the growth in popularity of wireless networks, new kinds of repeater mechanisms are coming into play; for example, "smart antennas" can selectively amplify some signals and inhibit others [Liberti and Rappaport, 1999].

24.3.3 Voice Modems

Modems are used when a digital signal must be encoded in analog fashion using modulation. This is often the case when home subscribers to an Internet service provider must use the analog lines of the public switched telephone network (PSTN) to go online. The word "modem" is a shortened form of modulator/demodulator.

Some manufacturers make a distinction between **hard modems**, which contain all of the necessary digital signal processing functions on the modem board, and the less-expensive **soft modems**, which off-load some of the signal processing responsibility to software on the host machine.

Modems designed for data transmission over telephone lines, also called voice-grade modems or simply telephone modems, use a combination of modulation techniques. QAM is used in all modems that adhere to ITU-T modem standards V.32 and higher.

The most recent ITU voice-grade modem recommendations as of this writing are V.90 and V.92, which specify an asymmetric data connection in which one direction (called "upstream") is analog and the other ("downstream") is digital. The maximum data rates for these are 33,000 to 48,000 bits per second for upstream and about 56,000 bits per second for downstream, respectively.

The difference in the upstream and downstream rates for V.90/V.92 modems is due to the fact that the digital-to-analog conversion used in the upstream connection incurs a penalty (the so-called "quantization noise"). The upstream data rate is nearly at the "Shannon limit" of roughly 38,000 bits per second, a theoretical upper bound on the achievable bit rate based on estimates of the best signal-to-noise ratio for analog telephone lines [Tanenbaum, 1996]. (Compression techniques used in V.92 permit some improvement here.) Lower signal-to-noise ratios result in slower data transfer rates. However, disregarding noise, both rates are limited to about 56,000 bits per second (56 Kbps) by the inherent properties of telephone voice transmission, which uses a very narrow bandwidth (about 3000 Hertz [3 kHz]), and a technique called pulse code modulation (PCM), which involves sampling the analog signal 8000 times per second. Each sample yields 8 bits, 7 of which contain data and 1 of which is used for control.

In the United States, the 56-Kbps downstream rate has, until recently, been unachievable owing to a Federal Communications Commission (FCC) restriction on power usage that effectively reduced the limit to 53K. In 2002, the FCC granted power to set such standards to a private industry committee, the Administrative Council for Terminal Attachments, which adopted new power limits permitting the 56-Kbps data rate for V.90 and V.92 modems [FCC, 2002; ACTA, 2002].

24.3.4 Cable Modems

The word "modem," once used almost exclusively to refer to voice-grade communication over telephone lines, has more recently been applied to other interfacing devices. A cable modem interfaces a computer to the cable television infrastructure, allowing digital signals to share bandwidth with the audio and visual broadcast content. Cable modem systems generally use coaxial cable or a combination of coaxial cable and fiber-optic cable (the latter are called hybrid fiber-coaxial, or HFC, networks). The much larger bandwidth of the cable TV connection permits faster data rates than telephone modems. As was the case with V.90 modems, the upstream and downstream data rates differ. With a cable modem, upstream data transfer rates can be millions of bits per second (Mbps), and downstream data rates can be in the tens of millions. The discrepancy between upstream and downstream rates is due, in part, to the fact that cable television systems were originally designed only for downstream data transfer from the cable provider to the television owner. A small portion of the bandwidth must be allocated for the upstream signal. (Cable companies offering cable modem service must upgrade their equipment by adding amplifiers for the upstream signal.)

Unlike telephone modem connections, which are dedicated point-to-point connections, cable modem connections are part of a broadcast network consisting of all the cable modem users in a particular neighborhood. When demand is high, the portion of bandwidth available to each user diminishes and data transfer rates decrease. In fact, a cable modem should be considered more as a layer 2, or data-link layer, device because it actually links a client computer to a local area network of cable modem users.

24.3.5 DSL

Sometimes, the word "modem" is used in a rather loose sense. For example, digital subscriber lines (DSL) use existing copper telephone wires and also require so-called digital modems. Technically, a digital modem is not a "modulator/demodulator," but the term has become standard. DSL exploits the fact that much more bandwidth is available than the narrow range (between about 300 and 3300 Hz) used for voice transmission. A portion of this bandwidth is reserved for ordinary voice transmission; the rest is divided into upstream and downstream bandwidth and uses digital rather than analog techniques. DSL comes in

several varieties, the most common of which is asymmetric DSL, or ADSL. In ADSL, the upstream data rate is generally no more than about 600 Kbps, and the downstream rate is generally no more than about 8 Mbps.

There are several other varieties of DSL — rate-adaptive DSL (RADSL), high bit-rate DSL (HDSL), very high bit-rate DSL (VDSL), etc. Peden and Young [2001] describe the various flavors of DSL and provide an overview of how DSL modems have evolved from voice-grade modems.

24.3.6 Hubs and Other LAN Switching Devices

The commonplace meaning of switch, a device for making and breaking connections in a circuit, encompasses a great many components in networks at several levels of the protocol stack: hubs, telephone switches (part of the communication infrastructure supporting wide area networks), bridges (devices for connecting local area networks to other networks), routers (devices used to direct packets along an appropriate path), and others.

The word "switch" is sometimes specialized to a particular class of devices normally associated with layer 2 of the OSI stack, in order to distinguish them from less-intelligent devices such as hubs. Although they clearly must communicate via the physical layer, they possess additional capabilities to do data-link-layer processing such as interpreting physical device addresses. When the word "switch" is used in the sequel, it will always be more precisely qualified.

A *hub* is used to connect several data lines into a single local area network; it simply takes incoming bits and broadcasts them to all the other lines to which it is connected. Such a hub is sometimes called a repeating hub. Early hubs did not even perform the amplification needed to regenerate the signal. These "passive hubs" were equivalent to wiring concentrators that are used to gather wires together into a central location; hubs are still sometimes referred to as concentrators.

Most LANs are broadcast networks; that is, all messages sent across the LAN are seen by all hosts in the network. As the number of hosts in a network grows, it becomes inefficient to force every host machine to see every message (more formally, we say that these hosts all share the same **collision domain** because two messages are said to collide if they are broadcast at the same time). Better segmentation is achieved using the protocols of the data-link layer and more sophisticated devices called **bridges**, which are discussed in the next section. Commercial vendors are increasingly using phrases such as "intelligent hub" to describe more bridge-like, even router-like, devices.

Some physical layer switching devices are specialized to particular situations. FDDI (Fiber Distributed Data Interface) networks consist of a number of hosts connected to two counter-rotating rings of fiber-optic cable (i.e., one ring transmits data clockwise, the other transmits it counterclockwise). One of these rings is designated the primary ring and is normally used for all data transfer; the secondary ring is idle. However, from time to time, one of the network hosts will fail, or one of the fibers may become damaged. In this case, a switch called a dual attachment concentrator "wraps" the primary and secondary rings together to form a single ring. A second type of switch, called an optical bypass switch, can also be used to handle host outages. Unlike the concentrator, the bypass switch optically "splices" each of the rings across the broken section, retaining the dual ring structure. (It is worth mentioning here that physically splicing fiber is an extremely precise operation; the optical switch uses mirrors to achieve the effect of an actual splice operation.) Downes et al. [1998] provide a good introduction to FDDI and its associated protocols.

Telephone switching has, of course, resulted in the design of many kinds of switches. One of the earliest types of switch, the **crossbar switch** (so named because the original electromechanical implementation actually used two layers of bars with each bar in one layer crossing every bar in the other), contains connection points for every combination of communication endpoints. Thus, if n ports are involved, there are n^2 switching points, of which only n can be closed at any given time (or $n/2$ if we remove the symmetry and use only the upper triangle of the connection point grid). Despite this quadratic growth, the simplicity of the crossbar design has made it a popular choice for switching, even in high-performance networks. For example, Stanford University's Tiny Tera packet switch uses a 32-port crossbar [Gupta and

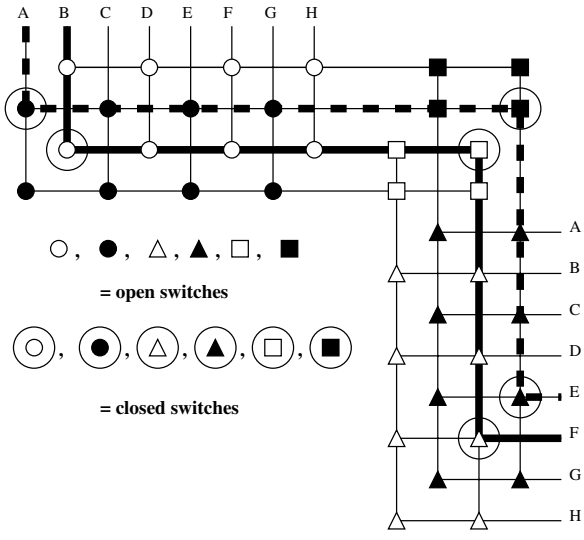


FIGURE 24.4 Space division switching.

McKeown, 1999]. Abel et al. [2003] describe a 256-port crossbar switch operating at data rates up to 4 terabits per second.

A necessary property of any switch is the ability to connect any pair of ports to one another. A desirable property is to permit a large number of simultaneous connections. A crossbar switch allows the maximum number of simultaneous connections, but in many cases it can be shown that fewer than the maximum will be required based on normal network traffic behavior. In such cases, it is possible to use small crossbar switches as modules for building larger multistage switches that allow any port pairs to be connected. This is called *space division switching*. Because there are fewer switches, some attempts to connect ports may "block" (blocking occurs when both endpoints are free but every path between them is unavailable due to conflicts with other connections). Figure 24.4 illustrates this. Any of the eight ports A,...,H can be connected to any other port. While a crossbar switch would require 64 connection points, this design uses a combination of four 4-by-2 crossbars (indicated by white and black circles and triangles) and two 2-by-2 crossbars (indicated by white and black rectangles) and has only 40 connection points. Connections are shown between A and E, and between B and F. Ports C and G are currently free, but it is not possible to connect them to each other because the wires needed to join them are already in use by other connections; an attempt to connect C to G would be blocked. In a full crossbar, this connection could be made.

One alternative to space division switching is time division switching. For this we use time division multiplexing to select one of the n input ports during each time slot, in round-robin fashion. Each input slice is placed into a buffer corresponding to its output port, so that after n time slices, all n buffers will be filled. On the output side, buffers are emptied in round-robin fashion and the contents of the ith buffer are sent to output port i. However, it requires n cycles to process one slice from each input port, so timing considerations come into play: the rate at which buffers are filled and emptied must be at least n times faster than the data rate for each port.

All the switching mechanisms described thus far have assumed a one-to-one pairing of input and output ports, an appropriate assumption when telephone conversations are being discussed. For data transfer, however, we often have multiple data streams directed to a single common destination. Switching now becomes just one component (the "switching fabric") of more complicated devices capable of dealing with such routing issues. These will be taken up once again when we describe switching functions in the higher layers of the protocol stack.

24.4 Best Practices: Data-Link Layer Examples

The data-link layer is sometimes divided into two sublayers. The higher of these is often called the logical link control layer (LLC); the lower sublayer is called the media access layer (MAC). The LLC is responsible for grouping bits together into units called frames and for handling things like flow and error control, while the MAC handles media-specific issues.

24.4.1 Network Interface Cards

Any networked device requires an interface to attach it to the network. A network interface card (NIC), also called an adapter, serves this function. It carries out the layer 2 protocols (framing, flow control, encoding/decoding, etc.) and the layer 1 protocols (transmitting a bit stream onto the network medium). There are two popular driver architectures for network adapters: 3Com/Microsoft's NDIS (Network Device Interface Specification) and Novell's ODI (Open Datalink Interface).

24.4.2 An Example: Ethernet NIC

Due to the many different data-link layer protocols in existence, it would be impractical to try to describe every variation in network adapter behavior. As an example of the type of processing required of an NIC, we describe the main features of the carrier-sense multiple-access protocol with collision detection (CSMA/CD) as described in IEEE [2002] and implemented by a traditional 10-Mbps Ethernet LAN. (Strictly speaking, Ethernet is not identical to IEEE 802.3; but we ignore the differences in what follows.)

Every Ethernet NIC has a 6-byte address, called its MAC address or Ethernet address, that uniquely distinguishes it from every other Ethernet adapter (IEEE controls the allocation of MAC addresses and enforces this uniqueness). Data packets received from the upper layers of the protocol stack are packaged by the data-link layer into frames of sizes ranging from 64 to 1518 bytes. These include the data (which might have been padded with extra bits in order to achieve minimum frame size), the MAC addresses of the sender and the intended recipient, and several other fields. The physical layer prepends an 8-byte preamble used for synchronization and timing purposes, and then transmits the resulting frame using the Manchester encoding described in a previous section.

When a host is ready to transmit, it sets a counter k to zero (this counter keeps track of the number of re-trys made by the sender). It then senses the wire to see if anyone else is transmitting. When the line is free, it waits an additional 96 bit-times (the time needed to transmit 1 bit) — about 9.6 microseconds for 10-Mbps Ethernet — and starts sending data. If, during the transmission, another host were also to begin transmitting, the resulting collision will be detectable by both senders. At this point, the CSMA/CD protocol requires both hosts to stop transmitting and to emit a 32-bit "jam signal." The sender then increments k and chooses a random number r in the range 0 through 2^{k-1} and waits $512 \times r$ bit-times before trying to retransmit. This is called *exponential backoff*.

The reason for requiring a minimum frame size has to do with collision detection. In the slowest Ethernet environment, 10 Mbps, a signal can travel no more than 2500 meters (the maximum allowed by the Ethernet standard). In this environment, conservatively assuming a propagation speed of one third the speed of light, or 10^8 meters per second (propagation in copper wire is usally twice this amount), a single bit could require as many as 25 microseconds (25 μs) to reach any host. If a signal were to travel this far just as a distant host began transmitting, it would take another 25 μs for news of the collision to return to the sender of the initial transmission. Therefore, a host might need to wait at least 50 μs to learn of a collision. It is important that the transmission last at least this long so that it can be terminated and restarted. About 500 bits (over 60 bytes) can be sent during this time at a speed of 10 Mbps, so 64 bytes is a conservative lower bound to enforce on the size of an Ethernet frame.

The IEEE 802 standards specify similarly precise behaviors for interfacing to Token Ring LANs, wireless LANs (in this case, the NIC is a wireless transceiver, often in the form of a PC card), and others. Most of these are described in nearly any published standard introduction to networks.

24.4.3 Bridges and Other Data-Link Layer Switches

A bridge connects several networks, forwarding data based upon the physical device addresses of the destination host as encoded in the headers added by the link layer protocol (unlike hubs, which indiscriminately forward bits to everyone, and unlike routers, which use logical addresses, such as IP addresses, rather than physical addresses). The word "switch" is sometimes used interchangeably with "bridge," but the primary difference between them is that LAN switches implement the switching function in hardware, while a bridge implements the switching function in software. For this reason, layer 2 switches can handle more traffic at higher data rates.

Some LAN switches have an additional capability; they are capable of switching a data stream immediately after determining the physical address of the destination (rather than waiting for the entire packet to arrive before forwarding it). These are called **cut-through switches**.

Bridges and switches are described in greater detail in Chapter 73; an excellent treatment of the subject can be found in Perlman's book [1999].

24.5 Best Practices: Network Layer Examples

Layer 3 of the OSI Reference Model deals mainly with routing. Protocols at this level are responsible for determining where to direct data based upon a *logical address*. In the current incarnation of the Internet protocol, IP version 4 (IPv4), the 32-bit IP address plays this role. However, the proposed IPv6 addressing scheme [Hinden and Deering, 2003] expands the size of IP addresses to 128 bits. For a description of the evolution of IP addressing and the impact of IPv6, see, for example, Kurose and Ross [2003].

The primary technology associated with layer 3 of the OSI stack is the router, which contains protocols from the lowest three layers of the OSI model. Unlike bridges, which use physical addresses to direct data, routers use logical addresses (such as IP addresses) to determine packet destinations. Routers fall into one of two categories: (1) edge routers, which are responsible for routing data between WANs and individual hosts or LANs; and (2) core routers, which perform routing between locations within the WAN.

Chapter 73 deals with routing algorithms (shortest path algorithms, multicast routing, distance vector vs. link state methods, etc.). The discussion here focuses more on the switching process used inside the router.

A router consists of a number of input ports, a number of output ports, and a switching fabric connecting them. We will assume that packets are all the same size (e.g., ATM cells). The small, fixed-size cells used in ATM make switching design much simpler, so a large amount of research has gone into designing high-speed ATM switches. In fact, this is one reason why ATM is a suitable technology for WANs — it lends itself to the design of high-performance switches. Several examples are described here.

Even assuming that all input ports and output ports have the same data rate and that the switching fabric is fast enough to handle all non-conflicting switching requests, it will eventually happen that data packets arriving at two different input ports will be destined for the same output port. The switch must place one or the other packet into a queue.

When queuing occurs at the output ports, there is always a danger of the queue becoming full, in which case routing software must make a decision about which packet to discard. A special kind of crossbar switch, called a *knockout switch* [Yeh, Hluchyj, and Acampora, 1987], uses a series of "competitions" between randomly chosen pairs of packets to eliminate some of them from entering the queue. For descriptions of more sophisticated switching fabrics, see Oie et al. [1990] and Robertazzi [1998].

An alternative to queueing the packet at the output port is to wait until the destination is free before forwarding the second data item. In this case, queueing occurs at the input ports. It is possible that a packet waiting in the input queue for its destination port, which is blocked, may prevent a packet further back in line, whose destination port is available, from being transmitted. This is called *head-of-the-line blocking*. Most switching fabric designs try to avoid input queueing.

A complete and thorough treatment of switching technologies and the associated scheduling and queueing algorithms is beyond the scope of this chapter. Although it has been in print for over a decade, the

IEEE Computer Society's collection of papers on switching is still a useful source of background information on switching technologies [Dhas, Konangi, and Sreetharan, 1991]. Turner and Yamanaka [1998] provide an overview of large-scale switching technologies (with particular reference to ATM networks). Li [2001] provides a modern mathematical treatment of switches with particular reference to networking applications.

24.6 Research Issues and Summary

The biggest challenges facing the next generation of networking devices appear to be in the areas of infrastructure, multimedia, and wireless networking. Most long-distance data transmission now takes place over fast media such as fiber-optic cable. However, replacing copper wire with fiber at the bottleneck points where most consumers access the Internet — the so-called "last mile" — is expensive and time-consuming. DSL and cable modem technologies provide some measure of broadband access, but they have reached the limits imposed by the copper wire medium. A number of alternatives to fiber have been proposed, most of them involving wireless technologies. In recent years, Multichannel Multipoint Distribution Service (MMDS, also sometimes called Multipoint Microwave Distribution System), originally a one-way microwave broadcast technology for subscription television, has been adopted for use in last-mile Internet connections [Ortiz, 2000]. Free-space optics involving infrared lasers could be used in a network of laser diodes to boost last-mile data rates into the gigabit per second range. Acampora and Krishnamurthy [1999] provide a description of a possible wireless broadband access network; and Acampora [2002] gives a nontechnical introduction to these technologies.

Infrastructure includes not only the hardware but also the protocols. TCP/IP has proven to be remarkably robust, to the embarrassment of experts who have occasionally predicted its collapse [Metcalfe, 1996], but address space considerations alone will soon force the issue of making the transition to IPv6.

Multimedia is one of the driving economic forces in the networking industry; services such as video-on-demand [Ma and Shin, 2002], Voice-over-IP [Rosenberg and Schulzrinne, 2000], and others require not only large amounts of bandwidth, but more attention to issues such as real-time networking, multicasting, and quality-of-service. These, in turn, will influence the design of not only the software protocols, but also the switches and routers that must implement them at the hardware level. *Convergence* is the word often used to describe the goal of integrating voice, data, audio, and video into a single network architecture. B-ISDN and ATM were designed with this in mind, but there are still many hurdles to overcome.

Defining Terms

Bridge: A data-link-layer device that connects LANs (usually of the same type) to one another and performs routing based upon physical addresses.

Circuit switching: A transmission method that uses a dedicated path over which the data bit stream is sent.

Cut-through switch: A switch that bypasses the store-and-forward approach of older packet switches and begins immediately to forward data as soon as the destination port has been determined.

IEEE 802: A collection of local area networking standards developed by the Institute of Electrical and Electronics Engineers, including 802.2 (Token Ring), 802.3 (CSMA/CD), and 802.11 (wireless OFDM and CSMA protocols).

Hub: A device used to join several LANs together; broadcasts all bits arriving from an attached network to every other attached network.

LAN switch: A device much like a bridge, but with switching functions implemented in hardware rather than software.

Modem (modulator/demodulator): A device for converting back and forth between digital data and an analog transmission medium.

Modulation: The use of a signal, usually digital, to modify an underlying carrier signal by varying the carrier's amplitude, frequency, phase, or some combination of these properties.

Multiplexing: The process of merging several signals into one; can be done in any of several domains: time, frequency, wavelength, etc. The inverse is called *demultiplexing*.

Packet switching: A transmission method involving the segmentation of a message into units called packets, each of which is separately and independently sent.

Protocol: A set of conventions and procedures agreed upon by two communicating parties.

Repeater (regenerator): A device used to simply retransmit the data that it receives; used to extend the range of transmission medium.

Router: A network layer device that forwards packets based upon their logical destination address.

Virtual circuit: A transmission method in which packets or cells are transmitted upon a path that has been reserved in advance; may be permanent or switched.

References

Abel, F., Minkenberg, C., Luijten, R., Gusat, P., and Iliadis, I. 2003. A four-terabit single-stage packet switch supporting long round-trip times. *IEEE Micro* 23(1): 10–24.

Acampora, A.S. 2002. Last mile by laser. *Scientific American*, 287(2):48–53.

Acampora, A.S. and Krishnamurthy, S.V. 1999. A broadband wireless access network based on mesh-connected free-space optical links. *IEEE Personal Communications*, 6(5):62–65.

ACTA (Administrative Council for Terminal Attachments), 2002. Telecommunications — telephone terminal equipment — technical requirements for connection of terminal equipment to the telephone network. Document TIA-968-A.

Anttalainen, Tarmo. 1999. *Introduction to Telecommunications Network Engineering.* Artech House, Inc., Boston, MA.

Benini, L. and De Micheli, G. 2002. Networks on chips: a new SoC paradigm. *Computer*. 35(1):70–78.

Braden, R., Faber, T., and Handley, M. 2002. From protocol stack to protocol heap — role-based architecture. *Comp. Commun. Rev.*, 33(1):17–22.

Dhas, C., Konangi, V.K., and Sreetharan, M. 1991. *Broadband Switching: Architecture, Protocols, Design, and Analysis*. IEEE Computer Society Press, Washington, D.C.

Downes, K., Ford, M., Lew, H.K., Spanier, S., and Stevenson, T. 1998. *Internetworking Technologies Handbook,* 2nd ed., Macmillan Technical Publishing, Indianapolis, IN.

FCC (Federal Communications Commission), 2002. Order on reconsideration in CC docket no. 99-216 and order terminating proceeding in CC docket no. 98-163. FCC document number 02-103, sec. III.G.1.

Forouzan, B.A. 2003. *Local Area Networks*. McGraw-Hill, Boston, MA.

Gallo, M.A. and Hancock, W.M. 2002. *Computer Communications and Networking Technologies*. Brooks/Cole, Pacific Grove, CA.

Gupta, P. and McKeown, N. 1999. Designing and implementing a fast crossbar scheduler. *IEEE Micro*, 20–28.

Hinden, R. and Deering, S. 2003. Internet Protocol Version 6 (IPv6) Addressing Architecture. Internet protocol version 6 (IPv6) addressing architecture (Network Working Group Request for Comments 3513). Internet Engineering Task Force (http://www.ietf.org/).

IEEE (Institute of Electrical and Electronics Engineers). 1999a. *IEEE Standard for Information technology — Telecommunications and information exchange between systems — Local and metropolitan area networks — Specific requirements — Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications — Amendment 1: High-Speed Physical Layer in the 5 GHz Band*. IEEE Standard: IEEE 802.11a-1999 (8802-11:1999/Amd 1:2000(E)). (http://standards.ieee.org/getieee802/802.11.html).

IEEE. 1999b. *IEEE 802.11b-1999 Supplement to 802.11-1999, Wireless LAN MAC and PHY Specifications: Higher Speed Physical Layer (PHY) Extension in the 2.4 GHz Band*. IEEE Standard: IEEE 802.11b-1999. (http://standards.ieee.org/getieee802/802.11.html).

IEEE. 2002. *IEEE Standard for Information technology — Telecommunications and Information Exchange between Systems — Local and Metropolitan Area Networks — Specific Requirements — Part 3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications*. IEEE Standard: IEEE 802.3-2002. (http://standards.ieee.org/getieee802/802.3.html).

IEEE. 2003. *IEEE Standard for IT — Telecommunications and Information Exchange between Systems LAN/MAN — Part II: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications Amendment 4: Further Higher Data Rate Extension in the 2.4 GHz Band*. IEEE Standard: 802.11G-2003.

ITU (International Telecommunications Union). 1998. Synchronous frame structures used at 1544, 6312, 2048, 8448 and 44 736 kbit/s hierarchical levels. ITU-T Recommendation G.704 (10/98).

Klensin, J.C. 2002. A policy look at IPv6: a tutorial paper. International Telecommunications Union Tutorial Workshop on IPv6, Workshop Document WS IPv6-3. (http://www.itu.int/itudoc/itu-t/workshop/ipv6/003.html).

Krishnamurthy, S. V., Acampora, A.S., and Zorzi, M. 2001. Polling based medium access control protocols for use with smart adaptive array antennas. *IEEE/ACM Transactions on Networking*, 9(2):148–161.

Kurose, J.F. and Ross, K.W. 2003. *Computer Networking: A Top-Down Approach Featuring the Internet*, 2nd ed. Addison-Wesley, Boston, MA.

Lawson, S. 2003. IEEE approves 802.11g standard. *Computerworld*, June 12, 2003. (http://www.computerworld.com/hardwaretopics/hardware/story/0,10801,82068,00.html).

Li, S.-Y.R. 2001. *Algebraic Switching Theory and Broadband Applications*. Academic Press, San Diego, CA.

Liberti, J.C. and Rappaport, T.S. 1999. *Smart Antennas for Wireless Communications: IS-95 and Third Generation CDMA Applications*. Prentice Hall PTR, Upper Saddle River, NJ.

Ma, H. and Shin, K.G. 2002. Multicast video-on-demand services. *ACM SIGCOMM Computer Communication Review*, 32(1):31–43.

Metcalfe, B. 1996. From the Ether: The Internet is collapsing; the question is who's going to be caught in the fall. *InfoWorld*, November 18, 1996.

Meyer, E.E., Jr. 1970. ARPA network protocol notes (Network Working Group Request for Comments 46). Internet Engineering Task Force (http://www.ietf.org/).

Oie, Y., Suda, T., Murata, M., and Miyahara, H. 1990. Survey of switching techniques in high-speed networks and their performance. In *Proceedings IEEE INFOCOM '93, Volume 3*. IEEE Press, Picataway, NJ.

Ortiz, S. 2000. Broadband fixed wireless travels the last mile. *IEEE Computer*, 33(7):18–21.

Peden, M. and Young, G. 2001. From voice-band modems to DSL technologies. *International Journal of Network Management*, 11(5):265–276.

Perlman, R. 1999. *Interconnections: Bridges, Routers, Switches, and Internetworking Protocols*, 2nd ed., Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

Peterson, L.L. and Davie, B.S. 2000. *Computer Networks: A Systems Approach,* 2nd ed., Morgan Kaufmann Publishers, San Francisco, CA.

Robertazzi, T.G. (Ed.). 1998. *Performance Evaluation and High Speed Switching Fabrics and Networks: ATM, Broadband ISDN, and MAN Technology*. Wiley-IEEE Press.

Rosenberg, J. and Schulzrinne, H. 2000. A framework for telephony routing over IP (Network Working Group Request for Comments 2871). Internet Engineering Task Force (http://www.ietf.org/).

Stallings, W. 2000. *Data and Computer Communications,* 6th ed., Prentice Hall, Upper Saddle River, NJ.

Stein, Jonathan (Y). 2000. *Digital Signal Processing: A Computer Science Perspective*. John Wiley & Sons, Inc., New York.

Tanenbaum, A.S. 1996. *Computer Networks*, 3rd ed., Prentice Hall, Upper Saddle River, NJ.

Turner, J. and Yamanaka, N. 1998. Architectural choices in large scale ATM switches. *IEICE Transactions on Communications*, E81-B(2):120–137.

Walrand, J. and Varaiya, P. 2000. *High-Performance Communication Networks,* 2nd ed., Morgan Kaufmann Publishers, San Francisco, CA.

Yeh, Y., Hluchyj, M.G., and Acampora, A.S. 1987. The knockout switch: a simple, modular architecture for high-performance packet switching. *IEEE J. Sel. Areas Commun.*, 5(8):1274–1283.

Further Information

A number of recent textbooks provide excellent surveys of the architecture of modern networks. In addition to those already mentioned, the books by Stallings [2000], and Peterson and Davie [2000] are worth examining.

It is nearly impossible to study computer networking without examining the telecommunications industry, because so much of networking, particular wide area networking, depends on the existing communication infrastructure. The book by Anttalainen [1999] gives a good overview of communications technologies, and concludes with chapters on data communication networks. Walrand and Varaiya [2000] also achieve a very smooth integration of networking and communication concepts.

A large number of professional organizations sponsor sites on the World Wide Web devoted to networking technologies. The IEEE's pilot "Get IEEE 802" program makes the 802 LAN standards documents freely available electronically (http://standards.ieee.org/getieee802/). In addition, the IEEE's Communications Society publishes a number of useful magazines and journals, including *IEEE Transactions on Communications*, *IEEE/ACM Transactions on Networking*, and *IEEE Transaction on Wireless Communications*.

The ATM Forum (http://www.atmforum.org) is a particularly well-organized and useful site, offering free standards documents, tutorials, and White Papers in all areas of Asynchronous Transfer Mode and broadband technology.

25

Fault Tolerance

Edward J. McCluskey
Stanford University

Subhasish Mitra
Stanford University

25.1 Introduction

Fault tolerance is the ability of a system to continue correct operation after the occurrence of hardware or software failures or operator errors. The intended system application is what determines the system reliability requirement. Since computers are used in a vast variety of applications, reliability requirements differ tremendously. For very low-cost systems, such as digital watches, calculators, games, or cell phones, there are minimal requirements: the products must work initially and should continue to operate for a reasonable time after purchase. Failures of these systems are easily discovered by the user. Any repair may be uneconomical. At the opposite extreme are systems in which errors can cause loss of human life. Examples are nuclear power plants and active control systems for civilian aircraft. The reliability requirement for the computer system on an aircraft is specified to be a probability of error less than 10^{-9} per hour [hissa.nist.gov/chissa/SEI_Framework/framework_7.html].

More typical reliability requirements are those associated with commercial computer installations. For such systems, the emphasis is on designing system features to permit rapid and easy recovery from failures. Major factors influencing this design philosophy are the reduced cost of commercial off-the-shelf (COTS) hardware and software components, the increasing cost and difficulty of obtaining skilled maintenance personnel, and applications of computers in banking, on-line reservations, networking, and also in harsh environments such as automobiles, industrial environments with noise sources, nuclear power plants, medical facilities, and space applications. For applications such as banking, on-line reservations, or e-commerce, the economic impact of computer system outages is significant. For applications such as space missions or satellites, computer failures can have a huge economic impact and cause missed opportunities

to record valuable data that may be available for only a short period of time. Computer failures in industrial environments, automobiles, and nuclear power plants can cause serious health hazards or loss of human life.

Fault tolerance generally includes detection of a system malfunction followed by identification of the faulty unit or units and recovery of the system from the failure. In some cases, especially applications with very short mission times or real-time applications (e.g., control systems used during aircraft landing), a fault-tolerant system requires correct outputs during the short period of mission time. After the mission is completed, the failed part of the system is identified and the system is repaired. Failures that cause a system to stop or crash are much easier to detect than failures that corrupt the data being processed by the system without any apparent irregularities in the system behavior. Techniques to recover a system from failure include system reboot, reloading the correct system state, and repairing or replacing a faulty module in the system.

This discussion is restricted mainly to techniques to tolerate hardware failures. The applicability of the described techniques to software failures will also be discussed.

25.2 Failures, Errors, and Faults

25.2.1 Failures

Any deviation from the expected behavior is a *failure*. Failures can happen due to incorrect functions of one or more physical components, incorrect hardware or software design, or incorrect human interaction.

Physical failures can be either permanent or temporary. A *permanent failure* is a failure that is always present. Permanent failures are caused by a component that breaks due to a mechanical rupture or some wearout mechanism, such as metal electromigration, oxide defects, corrosion, time-dependent dielectric breakdown, or hot carriers [Ohring 98, Blish 00]. Usually, permanent failures are localized. The occurrence of permanent failures can be minimized by careful design, reliability verification, careful manufacture, and screening tests. They cannot be eliminated completely.

A *temporary failure* is a failure that is not present all the time for all operating conditions. Temporary failures can be either transient or intermittent. Examples of causes of transient failures include externally induced signal perturbation (usually due to electromagnetic interference), power-supply disturbances [Cortes 86], and radiation due to alpha particles from packaging material and neutrons from cosmic rays [Ziegler 96, Blish 00].

An *intermittent failure* causes a part to produce incorrect outputs under certain operating conditions and occurs when there is a weak component in the system. For example, some circuit parameter may degrade so that the resistance of a wire increases or drive capability decreases. Incorrect signals occur when particular patterns occur at internal leads. Intermittent failures are generally sensitive to the temperature, voltage, and frequency at which the part operates. Often, intermittent failures cause the part to produce incorrect outputs at the rated frequency for a particular operating condition but to produce correct outputs when operated at a lower frequency of operation. Not all intermittent failures may be due to inaccuracies in manufacture. Intermittent failures can be caused by design practices leading to incorrect or marginal designs. This category includes cross-talk failures caused by capacitive coupling between signal lines and failures caused by excessive power-supply voltage drop. The occurrence of intermittent failures is minimized by careful design, reliability verification, and stress testing of chips to eliminate weak parts.

Major causes of software failures include incorrect software design (referred to as *design bugs*) and incorrect resource utilization such as references to undefined memory locations in data structures, inappropriate allocation of memory resoures and incorrect management of data structures, especially those involving dynamic structures such as pointers. Bugs are also common in hardware designs, and can involve millions of dollars when big semiconductor giants are involved [PC World 99, Bentley 01].

Failures due to human error involve maintenance personnel and operators and are caused by incorrect inputs through operator–machine interfaces [Toy 86]. Incorrect documentation in specification documents and user's manuals, and complex and confusing interfaces are some potential causes of human errors.

25.2.2 Errors

The function of a computing system is to produce data or control signals. An *error* is said to have occurred when incorrect data or control signals are produced. When a failure occurs in a system, the effect may be to cause an error or to make operation of the system impossible. In some cases, the failure may be benign and have no effect on the system operation.

25.2.3 Faults

A *fault model* is the representation of the effect of a failure by means of the change produced in the system signals. The usefulness of a fault model is determined by the following factors:

Effectiveness in detecting failures
Accuracy with which the model represents the effects of failures
Tractability of design tools that use the fault model

Fault models often represent compromises between these frequently conflicting objectives of accuracy and tractability. A variety of models are used. The choice of a model or models depends on the failures expected for the particular technology and the system design; it also depends on design tools available for the various models.

The most common fault model is the *single stuck-at fault*. Exactly one of the signal lines in a circuit described as a network of elementary gates (AND, OR, NAND, NOR, and NOT gates) is assumed to have its value fixed at either a logical 1 or a logical 0, independent of the logical values on the other signal lines. The single stuck-at fault model has gained wide acceptance in connection with manufacturing test. It has been shown that although the single stuck-at fault model is not a very accurate representation of manufacturing defects, test patterns generated using this fault model are very effective in detecting defective chips [McCluskey 00]. Radiation from cosmic rays that cause transient errors on signal lines have effects similar to a transient stuck-at fault which persists for one clock cycle.

Another variation of the stuck-at fault model is the *unidirectional fault model*. In this model, it is assumed that one or more stuck-at faults may be present, but all the stuck signals have the same logical value, either all 0 or all 1. This model is used in connection with storage media whose failures are appropriately represented by such a fault. Special error-detecting codes [Rao 89] are used in such situations.

More complex fault models are the *multiple stuck-at* and *bridging fault* models. In a *bridging fault* model, two or more distinct signal lines in a logic network are unintentionally shorted together to create wired logic [Mei 74]. It is shown in McCluskey [00] that a manufacturing defect can convert a combinational circuit into a sequential circuit. This can happen due to either *stuck-open faults* created by any failure that leaves one or more of the transistors of CMOS logic gates in a nonconducting state, or *feedback bridging faults* in which two signal lines, one dependent on the other, are shorted so that wired logic occurs [Abramovici 90].

The previous fault models all involve signals having incorrect logic values. A different class of fault occurs when a signal does not assume an incorrect value, but instead fails to change value soon enough. These faults are called *delay faults*. Some of the delay fault models are the transition fault model [Eichelberger 91], gate delay fault model [Hsieh 77], and path delay fault model [Shedletsky 78a, Smith 85].

The previously discussed fault models also have the convenient property that their effects are independent of the signals present in the circuit. Not all failure modes are adequately modeled by such faults. For example, *pattern sensitive faults* are used in the context of random–access memory (RAM) testing, in which the effect of a fault depends on the contents of RAM cells in the vicinity of the RAM cell to be tested. Failures occurring due to cross-talk effects also belong to this category [Breuer 96].

The fault models described until now generally involve signal lines or gates in a logic network. Many situations occur in which a less detailed fault model at a higher level of abstraction is the most effective choice. In designing a fault-tolerant system, the most useful fault model may be one in which it is

assumed that any single module can fail in an arbitrary fashion. This is called a *single module fault*. The only restriction placed on the fault is the assumption that at most one module will be bad at any given time.

There have been several attempts to develop fault models for software failures [Siewiorek 98]. However, there is no clear consensus about the effectiveness of any of these models — or even whether fault models can be developed for software failures at all.

25.3 Metrics and Evaluation

25.3.1 Metrics

Several metrics are used to quantify how reliable a system is. These are reliability, availability, safety, maintainability, performability, and testability.

The *reliability* of a system at time t is the probability that the system will produce correct outputs up to time t, provided it was producing correct outputs to start with. The concept of reliability directly applies situations in which the system must produce correct outputs for a given period of time. Examples include aircraft control, fly-by-wire systems, automobiles, nuclear reactors, and military and space applications.

The *availability* of a system at time t is defined as the probability that the system is operational at time t. Unlike a reliable system, a highly available system may not be operational for a very short while but must be quickly repaired so that it becomes operational again very quickly. Examples include servers and mainframes, network equipment, and telephone systems. For telephone networks, the availability requirement is 99.999% [http://www.iec.org/tutorials/five-nines/topic01.html], which roughly translates to 3 to 5 minutes of downtime per year.

The *safety* of a system at time t is the probability that the system either will be operating correctly or will fail in a "safe" manner. The way a system can fail in a "safe" manner depends on the actual application. For example, a safe failing state of a traffic light is when it is stays red and keeps blinking.

The *performability* of a system at time t is the probability that the system is operating correctly at full throughput or operating at a reduced throughput greater than or equal to a given value. The concept of performability, first introduced in Beudry [77] in terms of performance-related reliability measures, is useful in the context of systems with graceful degradation. As a simple example, consider a network of computers in which one or more network links or computers may not be working. In that case, some of the links and computers will have more traffic and higher workloads. As a result, the system throughput will be less than the full system throughput with all working network links and computers. Depending on the application, such a performance degradation may or may not be acceptable.

The *maintainability* of a system, denoted by $M(t)$, is the probability that it takes t units of time to restore a failed system to normal operation. Depending on the situation, a simple reload of the system state may be sufficient, or the faulty unit in the system (also referred to as the *field replaceable unit* or *FRU*) must be identified to be replaced or repaired. Maintainability is important; if it takes a long time to bring a failed system back to normal state, then the downtime increases and system availability suffers.

The *testability* of a system is the ease with which the system can be tested — this includes the ease with which test patterns can be generated and applied to the system. Test pattern generation cost depends on either the computer time required to run the test pattern generation program plus the (prorated) capital cost of developing the program, or the number of man-hours required for a person to write the test patterns plus the increase in system development time caused by the time taken to develop tests. Test application cost is determined by the cost of the test equipment plus the time required applying the test (sometimes called *socket time*). Attempts to understand circuit attributes that influence testability have produced the concepts of observability (visibility) and controllability (control). *Observability* refers to the ease with which the state of internal signals can be determined at the circuit outputs. *Controllability* refers to the ease of producing a specific internal signal value by applying signals to the circuit inputs [McCluskey 86].

25.3.2 Evaluation

Suppose that we want to estimate the reliability of a component (which can be a system itself) over time. An experimental approach to measure reliability is to take a large number N of these components and test them continuously. Suppose that at any time t, $G(t)$ is the number of components that are operating correctly up to time t and $F(t)$ is the number of components that failed from the time the experiment started up to time t. Note that $G(t)+F(t)=N$. Thus, the reliability $R(t)$ at time t is estimated to be $(G(t)/N)$, assuming that all components had equal opportunity to fail. The rate at which components fail is $(dF(t)/dt)$. The *failure rate* per component at time t, $\lambda(t)$, is $(1/G(t))(dF(t)/dt)$. Substituting $F(t)=N-G(t)$ and $R(t)=(G(t)/N)$, we obtain $\lambda(t)=-(1/R(t))(dR(t)/dt)$. When the failure rate is constant over time, represented by a constant failure rate λ, reliability $R(t)$ can be derived to be equal to $e^{-\lambda t}$ by solving the previous differential equation. This model of constant failure rate and exponential relationship between reliability and failure rate is widely used. Several other distributions that are used in the context of evaluation of reliable systems are hypoexponential, hyperexponential, gamma, Weibull, and lognormal distributions [Klinger 90, Trivedi 02].

Another metric very closely related to reliability is the *mean time to failure* (MTTF), which is the expected time that a component will operate correctly before failing. The MTTF is equal to $\int_{-\infty}^{\infty} tR(t)dt$. For a system with constant failure rate λ, the MTTF is equal to $\int_{-\infty}^{\infty} te^{-\lambda t}dt=(1/\lambda)$.

The failure rate is generally estimated from field data. For hardware systems, accelerated life testing is also useful in failure rate estimation [Klinger 90]. Figure 25.1 shows how the failure rate of a system varies with time. To start with, the failure rate is high for integrated circuits (ICs) for the first 20 weeks or so; during this time, weak parts that were not identified as defective during manufacturing testing typically fail in the system. After that, the failure rate stays more or less constant over time (in fact, it decreases a little bit), until the system lifetime is reached. After that, the failure rate increases again due to wearout. This dependence of failure rate on time is often referred to as the *bathtub curve*.

When a system fails, it must be repaired before it can be put into operation again. Depending on the extent of the failure, the time required to repair a system may vary. In general, it is assumed that the repair rate of the system is constant (generally represented by μ) and the *mean time to repair* (MTTR) is equal to $(1/\mu)$. For real situations, the assumption of constant repair rate may not be justified; however, this assumption makes the associated mathematics simple and manageable.

Once we know the system failure rate and the system repair rate, the system availability can be calculated using a simple Markov chain [Trivedi 02]. The average availability over a long period of time is given by $(MTTF/MTTF+MTTR)$. The *mean time between failures* (MTBF) is equal to MTTF + MTTR. In general, MTTR should be very small compared to MTTF so that the average availability is very close to 1.

As a hypothetical example, consider system A, with MTTF = 10 hours and MTTR = 1 hour, and system B, with MTTF = 10 days and MTTR = 1 day. The average availability over a long period of time is the same for both systems. While MTTR should be brought down as much as possible, having a system with MTTF = 10 hours may not be acceptable because of frequent disruptions and system outage. Thus, average availability does not model all aspects of system availability. Other availability metrics include instantaneous availability and interval availability [Klinger 90].

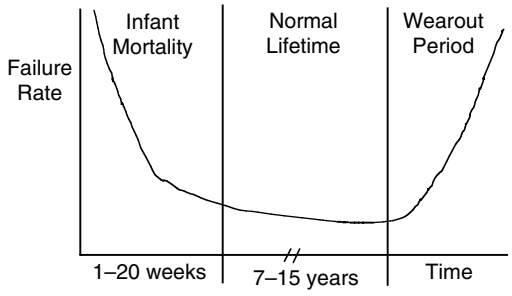


FIGURE 25.1 Failure rate vs. time: bathtub curve. The numbers on the time axis are relevant for integrated circuits.

Several techniques are used to evaluate the previous metrics for reliable system design [Trivedi 02, Iyer 96]. These include analytical techniques based on combinatorial methods and Markov modeling, simulation of faulty behaviors using fault injection, and experimental evaluation.

25.4 System Failure Response

The major reason for introducing fault tolerance into a computer system is to limit the effects of malfunctions on the system outputs. The various possibilities for system responses to failures are listed in Table 25.1.

25.4.1 Error on Output

With no fault tolerance present, failures can cause system outputs to be in error. For many systems, this is acceptable and the cost of introducing fault tolerance is not justified. Such systems are generally small and are used in noncritical applications such as computer games, word processing, etc. The important reliability attribute of such a system is the MTTF. This is an example of a product for which the best approach to reliability is to design it in such a way that it is highly unlikely that it does not fail — a technique called *fault avoidance*. For hardware systems, fault avoidance techniques include

- Robust design (e.g., special circuit design techniques, radiation hardened designs, shielding, etc.) [Kang 86, Calin 96, Wang 99]
- Design validation verification (e.g., simulation, emulation, fault injection, and formal verification)
- Reliability verification (to evaluate the impact of cross-talk, electromigration, hot carriers, careful manufacturing)
- Thorough production testing [Crouch 99, McClusley 86, Needham 91, Perry 95]

At present, it is not possible to manufacture quantities of ICs that are all free of defects. Defective chips are eliminated by testing all chips during production. Some parts may produce correct outputs only for certain operating conditions but may not be detected because production testing is not perfect. It is not economically feasible to test each part for all operating conditions. In addition, there are weak parts that may produce correct outputs for all specified operating conditions right after manufacture but will fail early in the field (within a few weeks) — much earlier than the other parts. Reliability screens are used to detect these parts. Techniques include burn-in [Jensen 82, Hnatek 87, Ohring 98], very low voltage (VLV) testing [Hao 93], and SHOrt Voltage Elevation (SHOVE) tests [Chang 97], and Iddq testing [Gulati 93]. (The applicability of Iddq testing is questionable for deep-submicron technologies.) The cost of IC testing is a significant portion of the manufacturing cost.

Depending on the application, fault avoidance techniques may be very expensive. For example, radiation hardening of hardware for space applications has a very limited market and is very expensive. Moreover, the radiation hardened designs are usually several generations behind the highest-performance designs.

Fault avoidance techniques for software systems include techniques to ensure correct specifications, validation, and testing. Fault avoidance techniques for human errors include adequate training, proper review of user documents, and development of user-friendly interfaces.

TABLE 25.1 System Output Response to Failure

Error on output	Acceptable in noncritical applications (e.g., games, etc.)
Errors masked	Outputs correct even when fault from specific class occurs. Required in critical control applications (e.g., flight control during aircraft takeoff and landing, fly-by-wire systems, etc.)
Fault secure or data integrity	Output correct or error indication if output incorrect. Recovery from failure required. Useful for critical situations in which recovery and retry is adequate (e.g., banking, telephony, networking, transaction processing, etc.)
Fail safe	Output correct or at "safe value." Useful for situations in which one class of output error is acceptable (e.g., flashing red for faulty traffic control light)

TABLE 25.2 Masking Techniques

Hardware	Voted logic — Each module is replicated. The outputs of all copies of a module are connected to a voter.
	Error correcting codes — Information has extra bits added. Some errors are corrected. Used in RAM and serial memory devices.
	Quadded (interwoven) logic — Each gate is replaced by four gates. Faults are automatically corrected by the interconnection pattern used.
	Fail safe — Output correct or at "safe value." Useful for situations in which one class of output error is acceptable (e.g., flashing red for faulty traffic control light).
Software	*N*-version programs — Execute a number of different programs written independently to implement the same specification. Vote on the final result.
	Recovery block — Execute acceptance test program upon completion of procedure. Execute alternate procedure if acceptance test is failed.

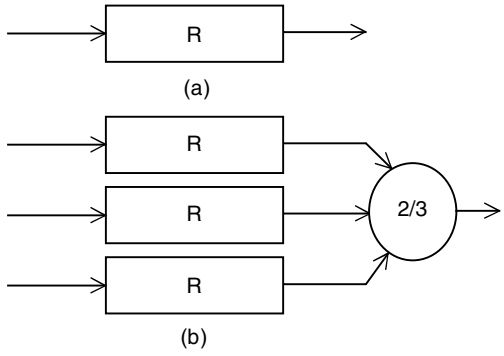


FIGURE 25.2 Voted logic. (a) Simplex, (b) TMR.

25.4.2 Error Masking

Several systems require that no errors are produced at their outputs. This is referred to as *error masking*. Such systems are usually real-time control systems in which outputs cause direct actions. Examples include air or spacecraft control surfaces, control for automobiles, medical systems, and nuclear and manufacturing controls. For such systems, the appropriate reliability parameter is the *fault tolerance*: the total number of failed elements that can be present without causing output errors. For some designs, it is important to consider both the total number of failed elements and the number of simultaneously failing elements that can be tolerated. Major error masking techniques are listed in Table 25.2.

Triple modular redundancy (TMR) is a widely used error working technique. Figure 25.2 illustrates this scheme, in which each module is replaced by three modules whose outputs are passed through a voting network before being transmitted as outputs. The three modules may be identical or different implementations of the same logic function. The concept of TMR was first developed in Von Neumann [56], in the context of models of neurons in the human brain.

One major advantage of TMR is its flexibility in choosing the module that forms the basic unit of replication. This can be as small as a single gate or as large as an entire computer. Any fault that affects only a single module will be masked. A direct extension of the TMR technique is *N-modular redundancy* (NMR), in which each module is replaced by N modules and voting is performed on the outputs of all modules. Several approaches to voting in TMR are discussed in Johnson [96], and Mitra [00a].

Suppose the reliability of each individual module of a TMR system is R; this is also referred to as the reliability of a simplex system or *simplex reliability*. The reliability of a TMR system is equal to $R^3 + 3R^2(1 - R) = 3R^2 - 2R^3$. Here, we assume that the voter reliability is 1, which is not generally the case. In that case, the TMR reliability must be multiplied by voter reliability. Figure 25.3 shows plots of simplex reliability and TMR reliability. It is assumed that the failure rate is constant, in which case R is equal to $e^{-(t/MTTF)}$, where MTTF is the mean time to failure of a simplex system.

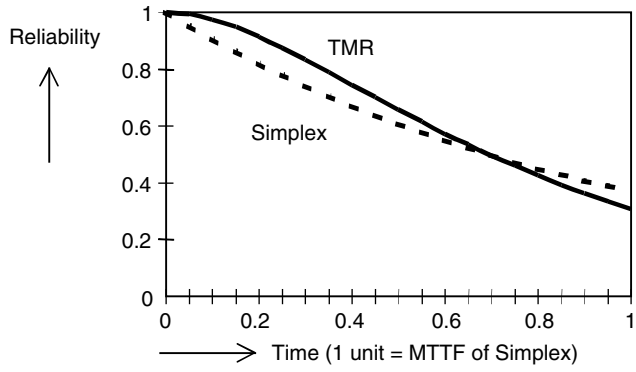


FIGURE 25.3 TMR reliability plot.

The following interesting observations can be made from Figure 25.3. The TMR reliability is greater than simplex reliability until time t equal to roughly seven-tenths (more precisely $log_e 2$) of the simplex MTTF (by solving for t when $R = 3R^2 - 2R^3$ and $R = e^{-(t/MTTF)}$). In fact, after that point TMR reliability is less than simplex reliability. Thus, TMR systems are effective only for very short mission times, beyond which their reliability is worse than simplex reliability.

As an example, a TMR system may be used during an aircraft landing when the system must produce correct outputs for that short time. This is true for NMR systems in general. For a system with a constant failure rate, the MTTF of a TMR system is roughly eight-tenths the MTTF of a simplex system. Thus, if MTTF is used as the only measure of reliability, then it will seem that the TMR is less reliable than a simplex system. However, this is not true for very short mission times (Figure 25.3). A TMR–simplex system [Siewiorek 98], which switches from a TMR to a simplex system, can be used to overcome the problem related to reliability of TMR for longer mission times. There are several examples of commercial systems using TMR [Riter 95, http://www.resilience.com]. Another example of hardware error masking is interwoven redundancy based on the concept of quadded logic [Tryon 62].

Error correcting codes (ECCs) are commonly used for error masking in RAMs and disks. Additional bits are appended to information stored or transmitted. Any faulty bit patterns within the capability of the used code are corrected, so that only correct information is provided at the outputs [Rao 89]. It relies on error correction circuitry to change faulty information bits and is thus effective when this circuitry is fault-free.

Two major software techniques for fault masking are N-version programming [Chen 78] and recovery blocks [Randall 75]. *N-version programming* requires that several versions of a program be written independently. Each program is run on the same data, and the outputs are obtained by voting on the outputs from the individual programs. This technique is claimed to be effective in detecting failures in writing a program.

The other software method, *recovery blocks*, also requires that several programs be written. However, the extra programs are run only when an error has been detected. Upon completion of a procedure, an acceptance test is run to check that no errors are present. If an error is detected, an alternate program for the procedure is run and checked by the acceptance test. One of the difficulties with this technique is the determination of suitable acceptance tests. A classification of acceptance tests into accounting checks, reasonableness tests, and run-time checks is discussed in Hecht [96].

25.4.3 Fault Secure Techniques

Fault secure techniques ensure that the output is correct unless an error indication is present. Errors must be detected but need not be corrected. Thus, the major objective is to preserve system data integrity so that any data corruption due to faults from a given class is detected. Error detection during normal system

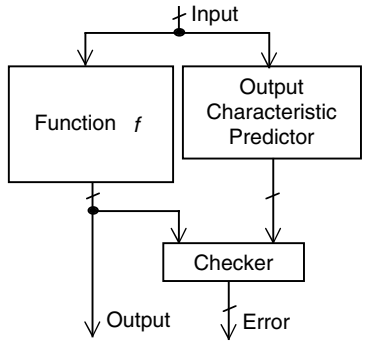


FIGURE 25.4 General architecture of a CED scheme.

operation is also referred to as *concurrent error detection* (CED) or *on-line checking*. CED techniques can be classified mainly into two major groups:

Hardware redundancy
Time redundancy

The basic principle behind CED based on hardware redundancy is shown in Figure 25.4 [Mitra 00b]. The system realizes function f and produces output sequence $f(i)$ for any input sequence i. A CED scheme generally contains another unit which independently predicts some special characteristic of the system output $f(i)$ for every input sequence i. Finally, a checker unit checks whether the special characteristic of the actual output produced by the system in response to sequence i matches the predicted characteristic and signals *error* when a mismatch occurs. Some examples of characteristics of $f(i)$ used for error detection purposes are $f(i)$ itself (duplication), its parity (parity prediction), 1s count, 0s count, transition count, etc. Coding schemes using parity and cyclic redundancy check (CRC) [Rao 89] are generally used for error detection in memories and data packets transmitted through computer networks.

A CED scheme is characterized by the class of failures in the presence of which data integrity is guaranteed. For the CED example in Figure 25.4, if the error signal is stuck-at-0, then an error will never be signaled. Thus, the CED system must be able to detect errors in the checking circuitry. An overview of self-checking checker designs can be obtained from Wakerly [78], and McCluskey [90]. Several CED techniques have been developed and used commercially for designing reliable systems [Sellers 68, Hsiao 81, Kraft 81, Chen 92, Webb 97, Spainhower 99]. Some of the important CED techniques are described next.

25.4.3.1 Duplication

Duplication is an example of a classic CED scheme in which there are two implementations, identical or different, of the same logic function. A self-checking comparator based on two-rail checkers is used to check whether the outputs from the two implementations agree. Duplication guarantees data integrity when one of the two implementations produces errors (single-module faults). Duplication has been used in several commercial systems [Kraft 81, Pradhan 96, Siewiorek 98, Webb 97, Spainhower 99].

25.4.3.2 Parity Prediction

The odd parity function indicates whether the number of 1s in a set of binary digits is odd. CED techniques for datapath circuits [Nicolaidis 97] and general combinational and sequential circuits based on parity prediction have been described in De [94], Touba [97], and Zeng [99]. For a design with parity prediction using a single parity bit, the circuit is designed in such a way that there is no sharing among the logic cones generating the circuit outputs. Thus, if a single fault affects a logic cone, at most one output will be erroneous, so that the error is detected by a parity checker. A self-checking parity checker design is given in McCluskey [90]. The restriction of no logic sharing among different logic cones may result in large

area overhead for circuits with a single parity bit. Synthesis of circuits with multiple parity bits and some sharing among logic cones is described in Touba [97]. It is shown in Mitra [00b] that the area overhead of parity prediction is comparable to duplication, sometimes marginally better, for general combinational circuits. Unlike duplication, which guarantees detection of all failures that cause nonidentical errors at the outputs of the two modules, parity prediction detects failures that cause errors at an odd number of outputs.

For datapath logic circuits, the area overhead of parity prediction becomes very high, because it is difficult to calculate the parity of the result of an addition or multiplication operation from the individual parities of the operands. Residue codes, described next, are used for error detection in these cases [Langdon 70, Avizienis 71].

25.4.3.3 Residue Codes

Given an n-bit binary word, which is the binary representation of the decimal number x, the *residue* modulo b is defined as equal to $y = x \bmod b$. The recommended value of b is of the form $2^m - 1$ for high error detection coverage and low cost of implementation. For example, when $b = 3$, we need two bits to represent y. Suppose we add two numbers x_1 and x_2 and we have the residues of these two numbers for error detection along the datapath: that is, $y_1 = x_1 \bmod b$, and $y_2 = x_2 \bmod b$. The residue of the sum $x_3 = x_1 + x_2$ is given by $y_3 = (y_1 + y_2) \bmod b$; the residue of the product $x_4 = x_1 \times x_2$ is given by $y_4 = (y_1 \times y_2) \bmod b$. Hence, addition and multiplication operations are said to preserve the residues of its operands. The overall structure of error detection using residue codes is shown in Figure 25.5.

25.4.3.4 Application-Specific Techniques

The overhead of error detection can be reduced significantly if the specific characteristics of an application are utilized. For example, for some functions it is very easy to compute the inverse of that function; that is, given the output it is very easy to compute the input. For such functions, a cost-effective error detection scheme is to compute the inverse of a particular output and match the computed inverse with the actual input. The LZ-based compression algorithm is an example of such a function, where computation of the inverse is much easier than computation of the actual function. Compression is a fairly complex process involving string matching; however, decompression is much simpler, involving a memory lookup. Hence, for error detection in compression hardware, the compressed word can be decompressed to check whether it matches the original word. This makes error detection for compression hardware very simple [Huang 00]. Other examples of application-specific techniques for error detection with low hardware overhead are presented in Jou [88] and Austin [99].

Error detection based on time redundancy uses some form of repetition of actual operations on a piece of hardware. For detection of temporary errors, simple repetition may be enough. For detection of permanent faults, it must be ensured that the same faulty hardware is not used in exactly the same way during repetition. Techniques such as alternate data retry [Shedletsky 78b], RESO [Patel 82], and ED^{4}I

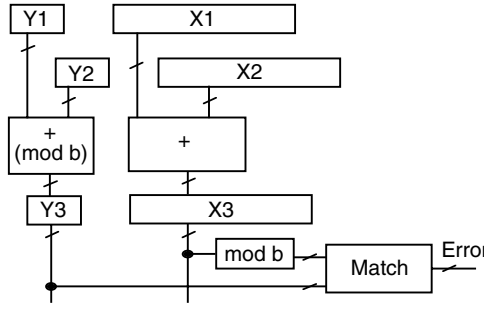


FIGURE 25.5 Error detection using residue codes.

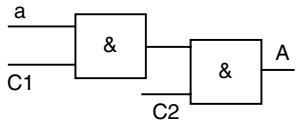


FIGURE 25.6 A simple fail-safe network.

[Oh 02a] are based on this idea. The performance overhead of error detection based on time redundancy may not be acceptable. Techniques such as multithreading [Saxena 00, Rotenberg 99, Reinhardt 00] or proper scheduling of instructions [Oh 02b] may be used reduce this overhead.

Some error detection techniques rely on a combination of hardware and software resources. Techniques include hardware facilities to check whether a memory access is to an area for which the executing program has authorization and also to verify that the type of access is allowed. Examples of other exception checks are given in Toy [86] and Siewiorek [98]. In a multiprocessor system, it is possible to run the same program on two different processors and compare the results. A more economical technique is to execute a reduced version of the main program on a separate device called a *watchdog processor* [Lu 82, Mahmood 88]. The watchdog processor is used to check the correct control flow of the program executed on the main processor, a technique called *control flow checking*. When the separate device is a simple counter, it is called a *watchdog timer*. Only faults that change the execution time of the main program are detected. Heartbeat signals that indicate the progress of a system are often used to ensure that a system is alive.

For signal processing applications involving matrix operations in a multiprocessor system, the number of processors required to check the errors in results can be significantly fewer than the number of processors required for the application itself, using a technique called *algorithm-based fault tolerance* (ABFT) [Huang 84].

Error detection using only software is very common and effective, but it must be developed for each individual program. Whenever some property of the outputs that ensures correctness can be identified, it is possible to execute a program that checks the property to detect errors. A simple example of checking output properties is the use of daily balance checks in the banking industry. The checked properties are also known as *assertions* [Leveson 83, Mahmood 83, Mahmood 85, Boley 95]. Automated software techniques for error detection without any hardware redundancy are described in Shirvani [00], Lovellette [02], Oh [02a], Oh [02b], and Oh [02c].

25.4.4 Fail-Safe Techniques

Fail-safe mechanisms cause the output state to remain in or revert to a safe state when a fault occurs. They typically involve replicated control signals and cause authorization to be denied when the control signals disagree. A simple example is a check that requires multiple signatures. Any missing or invalid signature causes the check not to be honored.

A simple fail-safe network is shown in Figure 25.6 Signals C1 and C2 have the same value in a fault-free situation. When they are both 0, the output is 0. If a fault causes them to disagree, the output is held at the safe value 0. This type of network is used in the fault-tolerant multiprocessor to control access to buses. No single fault can cause an incorrect bus access [Hopkins 78].

25.5 System Recovery

The rate of temporary failures is generally much higher than the rate of permanent faults [Lin 88]. Hence, when an error is detected, it is usually assumed that the error is due to some temporary failure. This temporary failure is expected to be gone when the operation is retried. If the failure persists even after a predetermined number of retries, it is assumed that the failure is a permanent one and repair routines are invoked.

Before a retry is performed, the system must start from some correct state. One solution is to initialize the entire system by rebooting it. However, a considerable amount of data may be lost as a result. A more systematic method of system recovery to prevent this data loss is checkpointingand rollback. The system state is stored in stable storage protected by ECC and battery backup at regular, predetermined intervals of time. This is referred to as *checkpointing*. Upon error detection, the system state is restored to the last checkpoint and the operation is retried; this is referred to as *rollback*. A particular application determines what system data must be checkpointed. For example, in a microprocessor checkpointing may be performed by copying all register and cache contents modified since the last checkpoint.

Techniques for roll-forward recovery in distributed systems that reduce the performance overhead of rollback recovery are described in Pradhan [94]. In a real-time system, rollback recovery may not be feasible because of the strict timing constraints. In that scenario, roll-forward recovery using TMR-based systems is applicable [Yu 01]. This technique also enables the use of TMR systems for temporary failures in long mission times.

Another useful recovery technique is called *scrubbing*. If the system memory is not used very frequently, then an error affecting a memory bit may not be corrected, because that memory word was not read out. As a result, errors may start accumulating, and used ECC codes may not be able to detect or correct the errors when the memory word is actually read out. Scrubbing techniques can be used to read the memory contents, check errors, and write back correct contents, even when the memory is not used. Scrubbing may be implemented in hardware or as a software process scheduled during idle cycles [Shirvani 00].

25.6 Repair Techniques

Unless a system is to be discarded when failures cause improper operation, failed components must be removed, replaced, or repaired. System repair may be manual or automatic. In the case of component removal, the system is generally reconfigured to run with fewer components, called *failed soft* or *graceful degradation*. If the faulty function can be automatically replaced or repaired, the system is called *self-repairing*. Self-repair techniques are useful for unmanned environments where manual repair is impossible or extremely costly. Examples include satellites, space missions, and remote or dangerous locations.

25.6.1 Built-in Self-Test and Diagnosis

The first error due to a fault typically occurs after a delay. This time is called the *error latency* [Shedletsky 76]. It is present because the output depends on the fault site logic value only for a subset of the possible input values. The error latency can be very long — so long that another fault (e.g., a temporary fault) occurs before the first fault is detected. Because of the limitations of the error-detection circuitry, the double error may be undetectable. In order to avoid this situation, provision is often made to test explicitly for faults that have not caused an output error. This technique is called *built-in self-test*. Built-in self-test is also useful in identifying the faulty unit after error detection.

Built-in self-test is generally executed periodically by interrupting the normal operation or when the system is idle. It is usually implemented by using test routines or diagnostic programs for programmed systems or by using built-in test hardware to apply a sequence of test vectors to the functional circuitry and checking its response [Bardell 87, Mitra 00c].

25.6.2 Fail-Soft Techniques

The most common form of automatic failure management involves disconnecting a failed component from the rest of the system and reconfiguring to continue operation until the bad device can be replaced. This can happen for as small a component as a portion of memory or for an entire processor, in the case of a multiprocessor system. The failed component is generally identified by means of built-in testing. Because no attempt is made to replace the disconnected component automatically, system operation may be degraded. This technique is sometimes called *graceful degradation*.

25.6.3 Self-Repair Techniques

Systems that must operate unattended for long periods require that the faulty components be replaced after being configured out of the system.

25.6.3.1 Standby Spares

One of the earliest self-repair techniques included a number of unused modules, *standby spares*, which could be used to replace failed components. The possibility of not applying power to a spare module until it is connected to the system is attractive for situations in which power is a critical resource or when the failure rate is much higher for powered than for unpowered circuits. Extensive hardware facilities are required to implement this technique, which is sometimes called *dynamic redundancy*. There are several major issues related to switching in a spare module without interrupting system operation.

25.6.3.2 Hybrid Redundancy

Fault masking and self-repair are obtained by combining TMR with standby spares, as shown in Figure 25.7 [Siewiorek 98]. Initially, the three top modules are connected through the switch to the voter inputs. The disagreement detector compares the voter output with each of the module outputs. If a module output disagrees with the voter output, the switch is reconfigured to disconnect the failed module from the voter and to connect a standby module to the voter instead. As long as at least two modules have correct outputs, the voter output will be correct. A failed module will automatically be replaced by a good module, as long as there are good modules available and the reconfiguration circuitry is fault-free. Thus, this system will continue to operate correctly until all spares are exhausted, the reconfiguration circuitry fails, or more than one of the on-line modules fail.

An advantage of this technique is the possibility of not applying power to a spare until it is connected to the voter. A disadvantage is the complexity of the reconfiguration circuitry. The complexity increases the cost of the system and limits its reliability [Ogus 75].

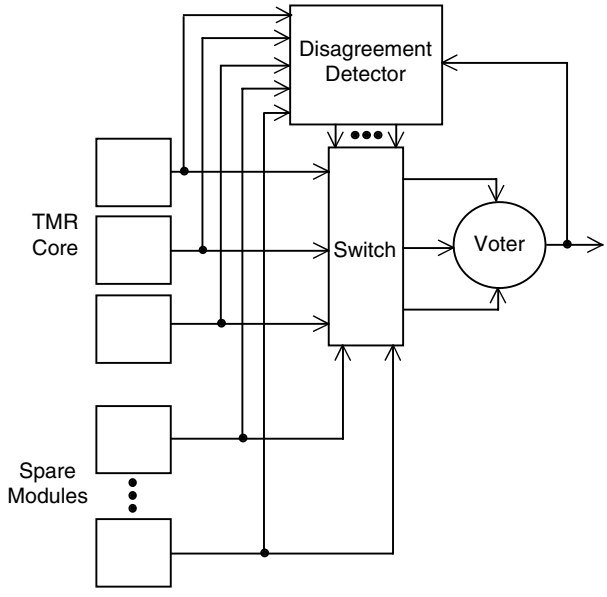


FIGURE 25.7 Hybrid redundancy with a TMR core and standby spares.

25.6.3.3 Self-Purging Redundancy

A self-purging system provides fault masking and self-repair with less complexity than hybrid redundancy [Losq 76]. It requires that all modules are powered until disconnected from the output due to a failure. Initially, all modules have their outputs connected to the voter inputs. The voter is a threshold network whose output agrees with the majority of voter inputs. Whenever one of the module outputs disagrees with the voter output, the module output is disconnected from the voter.

25.6.3.4 Reconfigurable Systems Self-Repair

In a reconfigurable system, such as a system with field programmable gate arrays (FPGAs), no spare modules are required for system self-repair. These systems can be configured multiple times by loading an appropriate configuration which implements logic functions in programmable logic blocks and interconnections among the logic blocks, controlled by switches implanted using pass transistors. After error detection, the failing portion (which can be a programmable logic block or a failing switch) is identified, and a configuration that does not use the failed resource is loaded. For practical purposes, such an approach does not cause any significant reliability or performance degradation of the repaired system. Self-repair and fault isolation techniques for reconfigurable systems are described in Lach [98], Abramovici [99], Huang [01a], Huang [01b], and Mitra [04a, 04b].

25.7 Common-Mode Failures and Design Diversity

Most fault-tolerance techniques are based on a single fault assumption. For example, for a TMR system or for error detection based on duplication, it is assumed that one of the modules will be faulty. However, in an actual scenario, several failure sources may affect more than one module of a redundant system at the same time, generally due to a common cause. These failures are referred to as *common-mode failures* (CMFs) [Lala 94]. Some examples of CMFs include design faults, power-supply disturbances, a single source of radiation creating multiple upsets, etc. For a redundant system with identical modules, it is likely that a CMF will have identical error effects in the individual modules [Tamir 84]. For example, if a CMF affects both modules of a system using duplicated modules for error detection, then data integrity is not guaranteed — both modules can produce identical errors. CMFs are surveyed in Lala [94] and Mitra [00d].

Design diversity is useful in protecting redundant systems against CMFs. Design diversity is an approach in which hardware and software elements of a redundant system are not just replicated; they are independently generated to meet a system's requirements [Avizienis 84]. The basic idea is that, with different implementations generated independently, the effects of a CMF will be different, so error detection is possible. *N*-version programming [Chen 78] is a diversity technique for software systems. Examples of hardware design diversity include flight control systems from Boeing [Riter 95], the space shuttle, Airbus 320 [Briere 93], and many other commercial systems. The conventional notion of diversity is qualitative. A quantitative metric for design diversity is presented in Mitra [02], and techniques for synthesizing diverse implementations of the same logic function are described in Mitra [00e] and Mitra [01]. Use of diversity during high-level synthesis of digital systems is described in Wu [01].

25.8 Fault Injection

Fault injection is generally used to evaluate and validate fault tolerance techniques by injecting faults into a prototype or through simulation of a software model of a system. For example, fault injection may be used to estimate the capability of a concurrent error detection technique to detect errors; on the other hand, fault injection may also be used to validate whether the control state machines designed to perform recovery are correctly designed. Fault injection is useful because of the flexibility in injecting faults at the areas of interest in a controlled fashion, unlike in an actual environment. Also, faults can be injected at a much higher frequency than in an actual scenario, where the failure rate may be very low. Thus, statistical

information about the capabilities of fault-tolerance techniques can be obtained in a much shorter time than in a real environment, where it may take several years to collect the same information.

However, it is questionable whether the injected faults represent actual failures. For example, results from test chip experiments demonstrate that very few manufacturing defects actually behave as single stuck-at faults [McCluskey 00]. Hence, experiments must be performed to validate the effectiveness of fault models used during fault injection. We briefly describe some of the commonly used fault injection techniques [Iyer 96, Shirvani 98].

Disturb signals on the pins of an IC — The signal values at the pins of an IC can be changed at random, or there may be some sequence and timing dependence between error events on the pins [Scheutte 87]. The problem with this approach is that there is no control over errors at the internal nodes of a chip not directly accessible through the pins.

Radiation experiments — Soft errors due to cosmic rays in the internal nodes of a chip can be created by putting the chip under heavy-ion radiation or a high-energy proton beam in radiation facilities. The angle of incidence and the amount of radiation can be controlled very precisely. This technique is used widely in industry.

Power-supply disturbance — Errors can be caused in the system by disturbing the power supply [Cortes 86, Miremadi 95]. This technique models errors due to disturbances in the power supply and in industrial environments.

Simulation techniques — Errors can be introduced at selected nodes of a design, and then the system can be simulated to understand its behavior in the presence of these errors. This technique is applicable at all levels of abstraction: transistor level, gate–level, or even a high-level description language level of the system. The choice of appropriate fault models is very important in this context. One of the drawbacks of simulation-based fault injection is the slow speed of simulation compared to the actual operation speed of a circuit. Several fault injection tools based on simulation have been developed, such as Kanawati [95] and Goswami [97].

25.9 Conclusion

Computing and communication systems constitute an inseparable part of our everyday lives — home appliances, communication, transportation, banking and finance management, embedded computing in space applications, and biomedical applications, including automated remote surgery, ICs implanted in human bodies (the pacemaker being a very simple example), and health monitoring systems. Malfunctions of computing and communication systems have also become an inseparable part of our daily lives. Such malfunctions can cause massive financial losses or loss of human lives for critical applications, such as servers, automobiles, avionics, space vehicles, and biomedical equipment. Security violations due to lapses in security protocol designs or unprecedented attacks by external agents are already a major cause of concern that will significantly grow with our increased reliance on computers and networks.

There is general consensus in the industry that these failure sources will become more pronounced. We need robust system designs ensuring high system availability, maintainability, quality of service, security, and efficient recovery from failures. These systems must be available even during bursts of heaviest demands, must detect errors in processed transactions, and must be able to repair themselves from failures very quickly. They also must be transparent to the user, through retry, reconfiguration, or reorganization through graceful degradation, for example. Hence, design techniques for self-organizing, self-repairing, and self-maintaining adaptive systems are required.

This chapter has surveyed techniques for evaluating and designing such robust systems. We have demonstrated that the issues of error control and repair are distinct and appropriate specifications for each must be matched to the intended applications. We also have presented techniques for satisfying these aspects of the system specifications and discussed fault injection techniques to evaluate the effectiveness of these techniques and validate their design. Some of these techniques are applicable in the context of designing robust, adaptive, self-repairing architectures for molecular and quantum computing systems.

25.10 Further Reading

The reader is encouraged to study the following topics related to this chapter:

Security and survivability [http://www.iaands.org/iaands2002/index.html]
Autonomic computing [http://www.research.ibm.com/autonomic]
Software rejuvenation [http://www.software-rejuvenation.com]
Efforts related to dependability benchmarking [http://www.ece.cmu.edu/~koopman/ifip_wg_10_4_sigdeb]
Defect and fault tolerance in molecular computing systems [http://www.hpl.hp.com/research/qsr]

References

[Abramovici 90] Abramovici, M., M.A. Breuer, and A.D. Friedman, *Digital Systems Testing and Testable Design*, IEEE Press, 1990.

[Abramovici 99] Abramovici, M., C. Stroud, C. Hamilton, C. Wijesuriya, and V. Verma, "Using Roving Stars for On-line Testing and Diagnosis of FPGAs," *Proc. Intl. Test Conf*., pp. 973–982, 1999.

[Austin 99] Austin, T., "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design," *Micro-32*, 1999.

[Avizienis 71] Avizienis, A., "Arithmetic Error Codes: Cost and Effectiveness Studies for Applications in Digital System Design," *IEEE Trans. Computers*, Vol. C-20, No. 11, pp. 1322–1331, Nov. 1971.

[Avizienis 84] Avizienis, A., and J.P.J. Kelly, "Fault Tolerance by Design Diversity: Concepts and Experiments," *IEEE Computer*, pp. 67–80, Aug. 1984.

[Bardell 87] Bardell, P.H., W.H. McAnney, and J. Savir, *Built-in Test for VLSI: Pseudorandom Techniques*, John Wiley & Sons, 1987.

[Bentley 01] Bentley, B., "Validating the Intel Pentium 4 Microprocessor," *Proc. Intl. Conf. Dependable Systems and Networks*, pp. 493–498, 2001.

[Beudry 77] Beudry, M.D., "Performance Related Reliability Measures for Computing Systems," *Proc. Intl. Symp. Fault-Tolerant Computing*, pp. 16–21, 1977.

[Blish 00] Blish, R., and N. Durrant, Eds., "Semiconductor Device Reliability Failure Models," *Tech. Transfer #00053955A-XFR, International Sematech*, May 31, 2000.

[Boley 95] Boley, D., G.H. Golub, S. Makar, N. Saxena, and E.J. McCluskey, "Floating Point Fault-Tolerance with Backward Error Assertions," *IEEE Trans. Computers*, *Special Issue on Fault-Tolerant Computing*, pp. 302–311, Feb. 1995.

[Briere 93] Briere, D., and P. Traverse, "Airbus A320/A330/A340 Electrical Flight Controls: A Family of Fault-Tolerant Systems," *Proc. Intl. Conf. Fault Tolerant Computing*, pp. 616–623, 1993.

[Breuer 96] Breuer, M.A., and S.K. Gupta, "Process Aggravated Noise (PAN): New Validation and Test Problems," *Proc. Intl. Test Conf.*, pp. 914–923, 1996.

[Calin 96] Calin, T., M. Nicolaidis, and R. Velazco, "Upset Harderened Memory Design for Submicron CMOS Technology," *IEEE Trans. Nuclear Science*, Vol. 43, No. 6, pp. 2874–2878, Dec. 1996.

[Chang 97] Chang, T.Y.J., and E.J. McCluskey, "SHOrt Voltage Elevation (SHOVE) Test for Weak CMOS ICs," *Proc. IEEE VLSI Test Symp*., pp. 446–451, 1997.

[Chen 78] Chen, L., and A. Avizienis, "N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation," *Proc. Intl. Symp. Fault-Tolerant Computing*, pp. 3–9, 1978.

[Chen 92] Chen, C.L., et al., "Fault-Tolerance Design of the IBM Enterprise System/9000 Type 9021 Processors," *IBM Journal Res. And Dev.*, Vol. 36, No. 4, pp. 765–779, July 1992.

[Cortes 86] Cortes, M., and E.J. McCluskey, "Modeling Power-Supply Disturbances in Digital Circuits," *Intl. Solid State Circuits Conf.*, pp. 164–165, 1986.

[Crouch 99] Crouch, A. L., *Design-for-Test for Digital ICs and Embedded Core Systems*, Prentice Hall, 1999.

[De 94] De, K., C. Natarajan, D. Nair, and P. Banerjee, "RSYN: A System for Automated Synthesis of Reliable Multilevel Circuits," *IEEE Trans. VLSI*, Vol. 2, pp. 186–195, June 1994.

[Eichelberger 91] Eichelberger, E.B., E. Lindbloom, J.A. Waicukauski, and T.W. Williams, *Structured Logic Testing*, Prentice Hall, Englewood Cliffs, NJ, 1991.

[Goswami 97] Goswami, K.K., R.K. Iyer, and L.Y. Young, "DEPEND: A Simulation-Based Environment for System Level Dependability Analysis," *IEEE Trans. Computers*, Vol. 46, No. 1, pp. 60–74, Jan. 1997.

[Gulati 93] Gulati, R.K., and C.F. Hawkins, Eds., *Iddq Testing of VLSI Circuits*, Kluwer Academic Publishers, 1993.

[Hao 93] Hao, H., and E.J. McCluskey, "Very-Low-Voltage Testing for Weak CMOS Logic ICs," *Proc. Intl. Test Conf.*, pp. 275–284, 1993.

[Hecht 96] Hecht, H., and M. Hecht, "Fault Tolerance in Software," *Fault-Tolerant Computer System Design*, D.K. Pradhan, Ed., pp. 428–477, Prentice Hall, Upper Saddle River, NJ, 1996.

[Hnatek 87] Hnatek, E. R., *Integrated Circuit Quality and Reliability*, Marcel Dekker Inc., 1987.

[Hopkins 78] Hopkins, A.L., T.B. Smith, and J.H. Lala, "FTMP — A Highly Fault-Tolerant Multiprocessor for Aircraft," *Proc. IEEE*, Vol. 66, No. 10, pp. 1221–1239, Oct. 1978.

[Hsiao 81] Hsiao, M-Y., W.C. Carter, J.W. Thomas, and W.R. Stringfellow, "Reliability, Availability and Serviceability of IBM Computer Systems: A Quarter Century of Progress," *IBM Journal Res. And Dev.*, Vol. 25, No. 5, pp. 453–469, Sept. 1981.

[Hsieh 77] Hsieh, E.P., R.A. Rasmussen, L.J. Vidunas, and W.T. Davis, "Delay Test Generation," *Proc. Design Automation Conf.*, pp. 486–491, 1977.

[Huang 84] Huang, K.H., and J.A. Abraham, "Algorithm Based Fault Tolerance for Matrix Operations," *IEEE Trans. Computers*, Vol. C-33, No. 6, pp. 518–528, June 1984.

[Huang 00] Huang, W.J., N.R. Saxena, and E.J. McCluskey, "A Reliable LZ Data Compressor on Reconfigurable Coprocessors," *Proc. IEEE Field Programmable Custom Computing Machines (FCCM)*, 2000.

[Huang 01a] Huang, W.J., and E.J. McCluskey "Column-Based Precompiled Configuration Techniques for FPGA Fault Tolerance," *Proc. IEEE Field Programmable Custom Computing Machines (FCCM)*, 2001.

[Huang 01b] Huang, W.J., S. Mitra, and E.J. McCluskey, "Fast Run-Time Fault Location for Dependable FPGA-based Applications," *Proc. Intl. Symp. Defect and Fault-Tolerance of VLSI Systems*, pp. 206–214, 2001.

[IRPS 02] Special Workshop on "Soft Errors," *Intl. Reliability Physics Symp.*, 2002.

[Iyer 96] Iyer, R.K., and D. Tang, "Experimental Analysis of Computer System Dependability," *Fault-Tolerant Computer System Design*, D.K. Pradhan, Ed., pp. 282–392, Prentice Hall, Upper Saddle River, NJ, 1996.

[Jensen 82] Jensen, F. and N.E. Petersen, *Burn-In: On Engineering Approach to the Design and Analysis of Burn-In Procedurese*, Wiley, 1982.

[Johnson 96] Johnson, B.W., "An Introduction to the Design and Analysis of Fault-Tolerant Systems," *Fault-Tolerant Computer System Design*, D.K. Pradhan, Ed., pp. 1–88, Prentice Hall, Upper Saddle River, NJ, 1996.

[Jou 88] Jou, J.Y., and J.A. Abraham, "Fault-Tolerant FFT Networks," *IEEE Trans. Computers*, Vol. 37, No. 5, pp. 548–561, May 1988.

[Kanawati 95] Kanawati, G.A., N.A. Kanawati, and J.A. Abraham, "FERRARI: A Flexible Software-Based Fault and Error Injection System," *IEEE Trans. Computers*, Vol. 44, No. 2, pp. 248–260, Feb. 1995.

[Kang 86] Kang, S.M., and D. Chu, "CMOS Circuit Design for Prevention of Single Event Upset," *Proc. Intl. Conf. Computer Design*, pp. 385–388, 1986.

[Klinger 90] Klinger, D.J., Y. Nakada, and M.A. Menendez, Eds., *AT&T Reliability Manual*, Van Nostrand Reinhold, New York, 1990.

[Kraft 81] Kraft, G.D., and W.N. Toy, *Microprogrammed Control and Reliable Design of Small Computers*, Prentice Hall, Englewood Cliffs, NJ, 1981.

[Lach 98] Lach, J., W.H. Manglone-Smith, and M. Potkonjak, "Efficiently Supporting Fault-Tolerance in FPGAs," *Proc. Intl. Symp. FPGAs*, pp.105–115, 1998.

[Lala 94] Lala, J.H., and R.E. Harper, "Architectural Principles for Safety-Critical Real-Time Applications," *Proc. IEEE*, Vol. 82, No. 1, pp. 25–40, Jan. 1994.

[Langdon 70] Langdon, G.G., and C.K. Tang, "Concurrent Error Detection for Group Look-ahead Binary Adders," *IBM Journal Res. and Dev.*, pp. 563–573, Sept. 1970.

[Leveson 83] Leveson, N.G., and P.R. Harvey, "Analyzing Software Safety," *IEEE Trans. Software Eng.*, SE-9(5), pp. 569–579, Sept. 1983.

[Lin 88] Lin, T.Y. and D.P. Siewiorek, "Error Log Analysis: Statistical Modeling and Heuristic Trend Analysis," *IEEE Trans. Reliability* Vol. 39, No. 4, pp. 419–432, Oct. 1990.

[Losq 76] Losq, J., "A Highly Efficient Redundancy Scheme: Self-Purging Redundancy," *IEEE Trans. Computers*, Vol. 12, No. 6, pp. 569–578, June 1976.

[Lovellette 02] Lovellette, M.N., et al., "Strategies for Fault-Tolerant, Space-Based Computing: Lessons Learned from the ARGOS Testbed," *Proc. IEEE Aerospace Conf.*, Vol. 5, pp. 2109–2119, 2002.

[Lu 82] Lu, D.J., "Watchdog Processors and Structural Integrity Checking," *IEEE Trans. Computers*, Vol. C-31, No. 7, pp. 681–685, July 1982.

[Mahmood 83] Mahmood, A., E.J. McCluskey, and D.J. Lu, "Concurrent Fault Detection Using a Watchdog Processor and Assertions," *Proc. Intl. Test Conf.*, pp. 622–628, 1983.

[Mahmood 85] Mahmood, A., E. Ersoz, and E.J. McCluskey, "Concurrent System-Level Error Detection Using a Watchdog Processor," *Proc. Intl. Test Conf.*, pp. 145–152, 1985.

[Mahmood 88] Mahmood, A., and E.J. McCluskey, "Concurrent Error Detection Using Watchdog Processors — A Survey," *IEEE Trans. Computers*, Vol. 37, No. 2, pp. 160–174, Feb. 1988.

[McCluskey 86] McCluskey, E.J., *Logic Design Principles with Emphasis on Testable Semicustom Circuits*, Prentice Hall, Englewood Cliffs, NJ, 1986.

[McCluskey 90] McCluskey, E.J., "Design Techniques for Testable Embedded Error Checkers," *IEEE Computer*, Vol. 23, No. 7, pp. 84–88, July 1990.

[McCluskey 00] McCluskey, E.J., and C.W. Tseng, "Stuck-Fault Tests vs. Actual Defects," *Proc. Intl. Test Conf*., pp. 336–343, 2000.

[Mei 74] Mei, K.C.Y., "Bridging and Stuck-at Faults," *IEEE Trans. Computers*, C-23, No. 7, pp. 720–727, July 1974.

[Miremadi 95] Miremadi, G., and J. Torin, "Evaluating Processor-Behavior and Three Error-Detection Mechanisms Using Physical Fault-Injection," *IEEE Trans. Reliability*, Vol. 44, No. 3, pp. 441–453, Sept. 1995.

[Mitra 00a] Mitra, S., and E.J. McCluskey, "Word-Voter: A New Voter Design for Triple Modular Redundant Systems," *Proc. IEEE VLSI Test Symp.*, pp. 465–470, 2000.

[Mitra 00b] Mitra, S., and E.J. McCluskey, "Which Concurrent Error Detection Scheme to Choose?" *Proc. Intl. Test Conf.*, pp. 985–994, 2000.

[Mitra 00c] Mitra, S., and E.J. McCluskey, "Fault Escapes in Duplex Systems," *Proc. IEEE VLSI Test Symp.*, pp. 453–458, 2000.

[Mitra 00d] Mitra, S., N.R. Saxena, and E.J. McCluskey, "Common-Mode Failures in Redundant VLSI Systems: A Survey," *IEEE Trans. Reliability*, *Special Section on Fault-Tolerant VLSI Systems*, Vol. 49, No. 3, pp. 285–295, Sept. 2000.

[Mitra 00e] Mitra, S., and E.J. McCluskey, "Combinational Logic Synthesis for Diversity in Duplex Systems," *Proc. Intl. Test Conf.*, pp. 179–188, 2000.

[Mitra 01] Mitra, S., and E.J. McCluskey, "Design of Redundant Systems Protected Against Common-Mode Failures," *Proc. IEEE VLSI Test Symp.*, pp. 190–195, 2001.

[Mitra 02] Mitra, S., N.R. Saxena, and E.J. McCluskey, "A Design Diversity Metric and Analysis of Redundant Systems," *IEEE Trans. Computers*, Vol. 51, No. 5, pp. 498–510, May 2002.

[Mitra 04a] Mitra, S., W.J. Huang, N.R. Sexena, S.Y. Yu, and E.J. McCluskey, "Dependable Reconfigurable Computing: Reliability Obtained by Adaptive Reconfiguration," *ACM Trans. Embedded Computing Systems*, To appear.

[Mitra 04b] Mitra, S., W.J. Huang, N.R. Saxena, S.Y. Yu, and E.J. McCluskey, "Reconfigurable Architecture for Autonomous Self-Repair," *IEEE Design & Test of Computers*, Special Issue on Design for Yield and Reliability, May-June, 2004.

[Needham 91] Needham, W.M., *Designer's Guide to Testable ASIC Devices*, Van Nostrand Reinhold, 1991.

[Nicolaidis 97] Nicolaidis, M., R.O. Duarte, S. Manich, and J. Figueras, "Fault-Secure Parity Prediction Arithmetic Operators," *IEEE Design and Test of Computers*, Vol. 14, No. 2, pp. 60–71, 1997.

[Ogus 75] Ogus, R.C., "Reliability Analysis of Hybrid Redundant Systems with Nonperfect Switches," *Technical Report, Computer Systems Lab., Stanford University*, CSL-TR-65, 1975.

[Oh 02a] Oh, N., S. Mitra, and E.J. McCluskey, "ED^{4}I: Error Detection by Diverse Data and Duplicated Instructions," *IEEE Trans. Computers, Special Issue on Fault-Tolerant Embedded Systems*, Vol. 51, No. 2, pp. 180–199, Feb. 2002.

[Oh 02b] Oh, N., P.P. Shirvani, and E.J. McCluskey, "Error Detection by Duplicated Instructions in Super-Scalar Processors," *IEEE Trans. Reliability*, Vol. 51, No. 1, pp. 63–75, March 2002.

[Oh 02c] Oh, N., P.P. Shirvani, and E.J. McCluskey, "Control-Flow Checking by Software Signatures," *IEEE Trans. Reliability*, Vol. 51, No. 1, pp. 111–122, March 2002.

[Ohring 98] Ohring, M., *Reliability and Failure of Electronic Materials and Devices*, Academic Press, 1998.

[Patel 82] Patel, J.H., and L.Y. Fung, "Concurrent Error Detection in ALUs by Recomputing with Shifted Operands," *IEEE Trans. Computers*, Vol. C-31, No. 7, pp. 589–595, July 1982.

[PC World 99] "Vendor Settles Suit over Alleged Problems in Floppy Disk Drives," *PC World*, Nov. 10, 1999.

[Perry 95] Perry, 6., *Digital Testing Course*, 1995, (http://www.soft-test.com).

[Pradhan 94] Pradhan, D.K., and N.H. Vaidya, "Roll-Forward Checkpointing Scheme: A Novel Fault-Tolerant Architecture," *IEEE Trans. Computers*, Vol. 43, No. 10, pp. 1163–1174, Oct. 1994.

[Pradhan 96] Pradhan, D.K., *Fault-Tolerant Computer System Design*, Prentice Hall, Upper Saddle River, NJ, 1996.

[Randall 75] Randall, B., "System Structure for Software Fault Tolerance," *IEEE Trans. Software Engineering*, Vol. SE-1, No. 2, pp. 220–232, June 1975.

[Rao 89] Rao, T.R.N., and E. Fujiwara, *Error-Control Coding for Computer Systems*, Prentice Hall, 1989.

[Reinhardt 00] Reinhardt, S., and S. Mukherjee, "Transient Fault Detection via Simultaneous Multi-threading," *Intl. Symp. Computer Architecture*, pp. 25–36, 2000.

[Riter 95] Riter, R., "Modeling and Testing a Critical Fault-Tolerant Multi-Process System," *Proc. Intl. Symp. Fault-Tolerant Computing*, pp. 516–521, 1995.

[Rotenberg 99] Rotenberg, E., "AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessor," *Proc. Fault-Tolerant Computing Symposium*, 1999.

[Saxena 00] Saxena, N.R., S. Fernandez Gomez, W.J. Huang, S. Mitra, S.Y. Yu, and E.J. McCluskey, "Dependable Computing and On-line Testing in Adaptive and Reconfigurable Systems," *IEEE Design and Test of Computers*, pp. 29–41, Jan–Mar 2000.

[Scheutte 87] Scheutte, M.A., and J.P. Shen, "Processor Control Flow Monitoring Using Signatured Instruction Streams," *IEEE Trans. Computers,* Vol. C-36, No. 3, pp. 264–276, March 1987.

[Sellers 68] Sellers, F., M.Y. Hsiao, and L.W. Bearnson, *Error Detection Logic for Digital Computers*, McGraw-Hill, 1968.

[Shedletsky 76] Shedletsky, J.J., and E.J. McCluskey, "The Error Latency of a Fault in a Sequential Circuit, *IEEE Trans. Computers*, C-25, pp. 655–659, June 1976.

[Shedletsky 78a] Shedletsky, J.J., "Delay Testing LSI Logic," *Proc. Intl. Symp. Fault Tolerant Computing*, pp. 159–164, 1978.

[Shedletsky 78b] Shedletsky, J.J., "Error Correction by Alternate-Data Retry," *IEEE Trans. Computers*, Vol. C-27, No. 2, pp. 106–112, Feb. 1978.

[Shirvani 98] Shirvani, P.P., and E.J. McCluskey, "Fault-Tolerant Systems in a Space Environment: The CRC Argos Project," *Technical Report, Center for Reliable Computing, Stanford University, CRC-TR-98-2, CSL-TR-98-774*, Dec. 1998.

[Shirvani 00] Shirvani, P.P., N. Saxena, and E.J. McCluskey, "Software-Implemented EDAC Protection Against SEUs," *IEEE Trans. Reliability, Special Section on Fault-Tolerant VLSI Systems*, Vol. 49, No. 3, pp. 273–284, Sept. 2000.

[Siewiorek 98] Siewiorek, D.P., and R.S. Swarz, *Reliable Computer Systems Design and Evaluation 3rd Ed.*, A.K. Peters, Massachusetts, 1998.

[Smith 85] Smith, G.L., "Model for Delay Faults," *Proc. Intl. Test Conf.*, pp. 342–349, 1985.

[Spainhower 99] Spainhower, L., and T.A. Gregg, "S/390 Parallel Enterprise Server G5 Fault Tolerance," *IBM Journal Res. and Dev.*, Vol. 43, pp. 863–873, Sept.–Nov., 1999.

[Tamir 84] Tamir, Y., and C.H. Sequin, "Reducing Common Mode Failures in Duplicate Modules," *Proc. Intl. Conf. Computer Design*, pp. 302–307, 1984.

[Touba 97] Touba, N.A., and E.J. McCluskey, "Logic Synthesis of Multilevel Circuits with Concurrent Error Detection," *IEEE Trans. CAD*, Vol. 16, No. 7, pp. 783–789, July 1997.

[Toy 86] Toy, W., and B. Zee, *Computer Hardware/Software Architecture*, Prentice Hall, Englewood Cliffs, NJ, 1986.

[Trivedi 02] Trivedi, K.S., *Probability and Statistics with Reliability, Queuing and Computer Science Applications*, *2nd Ed.*, John Wiley & Sons, New York, 2002.

[Tryon 62] Tryon, J.G., "Quadded Logic," *Redundancy Techniques for Computing Systems*, Wicox and Mann, Eds., pp. 205–208, Spartan Books, Washington D.C., 1962.

[Von Neumann 56] Von Neumann, J., "Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components," *Annals of Mathematical Studies*, Vol. 34, Eds. C.E. Shannon and J. McCarthy, pp. 43–98, 1956.

[Wakerly 78] Wakerly, J.F., *Error Detecting Codes, Self-Checking Circuits and Applications*, Elsevier North-Holland, New York, 1978.

[Wang 99] Wang, J.J., et al., "SRAM-based Reprogrammable FPGA for Space Applications," *IEEE Trans. Nuclear Science*, Vol. 46, No. 6, pp. 1728–1735, Dec. 1999.

[Webb 97] Webb, C.F., and J.S. Liptay, "A High Frequency Custom S/390 Microprocessor," *IBM Journal Res. and Dev.*, Vol. 41, No. 4/5, pp. 463–474, 1997.

[Wu 01] Wu, K., and R. Karri, "Algorithm Level Recomputing with Allocation Diversity: A Register Transfer Level Time Redundancy Based Concurrent Error Detection Technique," *Proc. Intl. Test Conf.*, pp. 221–229, 2001.

[Yu 01] Yu, S.Y., and E.J. McCluskey, "On-line Testing and Recovery in TMR Systems for Real-Time Applications," *Proc. Intl. Test Conf.*, pp. 240–249, 2001.

[Zeng 99] Zeng, C., N.R. Saxena, and E.J. McCluskey, "Finite State Machine Synthesis with Concurrent Error Detection," *Proc. Intl. Test Conf.*, pp. 672–680, 1999.

[Ziegler 96] Ziegler, J.F., et al., "IBM Experiments in Soft Fails in Computer Electronics (1978–1994)," *IBM Journal Res. and Dev.*, No. 1, Jan. 1996.

III

Computational Science

Computational Science unites computational simulation, scientific experimentation, geometry, mathematical models, visualization, and high performance computing to address some of the "grand challenges" of computing in the sciences and engineering. Advanced graphics and parallel architectures enable scientists to analyze physical phenomena and processes, providing a virtual microscope for inquiry at an unprecedented level of detail. These chapters provide detailed illustrations of the computational paradigm as it is used in specific scientific and engineering fields, such as ocean modeling, chemistry, astrophysics, structural mechanics, and biology.

26

Geometry-Grid Generation

Bharat K. Soni
Mississippi State University

Nigel P. Weatherill
University of Wales Swansea

26.1 Introduction

With the advent and rapid development of supercomputers and high-performance workstations, computational field simulation (CFS) is rapidly emerging as an essential analysis tool for science and engineering problems. In particular, CFS has been extensively utilized in analyzing fluid mechanics, heat and mass transfer, biomedics, geophysics, electromagnetics, semiconductor devices, atmospheric and ocean science, hydrodynamics, solid mechanics, civil engineering related transport phenomena, and other physical field problems in many science and engineering firms and laboratories.

The basic equations governing the general physical field can be represented as a set of nonlinear partial differential equations pursuant to a particular set of boundary conditions. For computational simulation, the field is decomposed into a collection of elemental areas (2-D)/volumes (3-D). The governing equations associated with the field under consideration are then approximated by a set of algebraic equations on these elemental volumes and are numerically solved to get discrete values which approximate the solution of the pertinent governing equations over the field. This discretization of the field (domain, region) into finite-elemental areas/volumes is referred to as grid generation, and the collection of discretized elemental areas/volumes is called the grid.

The numerical solution process associated with general CFS applications first involves discretization of the integral or differential form of the governing set of partial differential equations (PDEs) formulated in continuum. The discretization of these equations is usually influenced by the grid strategy under consideration and the solution strategy to be employed. In general, the solution strategies are classified as: finite difference, finite volume, and finite element. In the case of finite differences, the derivatives in the PDEs are represented by algebraic difference expressions obtained by performing Taylor series expansions of the associated solution variables at several neighbors of the point of evaluation [Thompson and Mastin

1983]. The differential forms of the governing equations are utilized in this case. However, the integral forms of the governing equations are used in the cases of finite-element and finite-volume strategies. Here the solution process involves representation of the solution variables over the cell in terms of selected functions, and then these functions are integrated over the volume (in case of finite-element) or the associated fluxes through cell sides (edges) are balanced (in case of finite volume).

The finite-element approach itself comes in two basic forms — the *variational*, where the PDEs are replaced by a more fundamental integral variational principle (from which they arise through the calculus of variations), or the *weighted residual* (Galerkin) approach in which the PDEs are multiplied by certain functions and then integrated over the cell.

In the finite-volume approach the fluxes through the cell sides (which separate discontinuous solution values) are best calculated with a procedure which represents the dissolution of such a discontinuity during the time step (Riemann solver).

The finite-difference approach, using the discrete points, is associated by many with rectangular Cartesian grids since such a regular lattice structure provides easy identification of neighboring points to be used in the representation of derivatives, whereas the finite-element approach has always been, by the nature of its construction on discrete cells, considered well-suited for irregular regions since a network of cells can be made to fill any arbitrarily shaped region and each cell is an entity unto itself, the representation being on a cell, not across cells. In view of the discretization strategy employed, the grids can be classified as structured, unstructured, or hybrid.

26.1.1 Structured Grids

Let $\mathbf{r} = (x, y, z)$ and $\mathbf{\Omega} = (\xi, \eta, \zeta)$ denote the coordinates in the physical and computational space. The structured grid is presented by a network of lines of constant ξ, η, and ζ such that a one-to-one mapping can be established between physical and computational space. The computational space is made up of uniformly distributed points within a square in two dimensions or a cube in three dimensions as demonstrated in Figure 26.1. The structured grid involving identity transformation between physical and computational space (that is, $x = \xi, y = \eta, z = \zeta$) is called a Cartesian grid. However, the body-fitted grid generated by utilizing discrete/analytic arbitrary transformations between physical and computational space is classified as a curvilinear grid. A grid around a cylinder demonstrating the Cartesian grid and curvilinear two-dimensional grid demonstrating O, C, and H type strategies and their respective correspondence with the computational domain are displayed in Figure 26.2a through Figure 26.2d.

The curvilinear grid points conform to the solid surfaces/boundaries and hence provide the most economical and accurate way for specifying boundary conditions. For example, in the O-type grid the boundary of the cylinder is specified at $\eta = 1$ boundary, and $\xi = 1$ and $\xi = \xi_{max}$ boundaries represent the same physical boundary (commonly referred to as a cut line). In the C-type grid the cylinder boundary is mapped into only a part of the ξ boundary, as shown in the Figure 26.2c. Here, the boundary segment in the front of the airfoil in the computational domain and at the tail of the airfoil represent the cut line

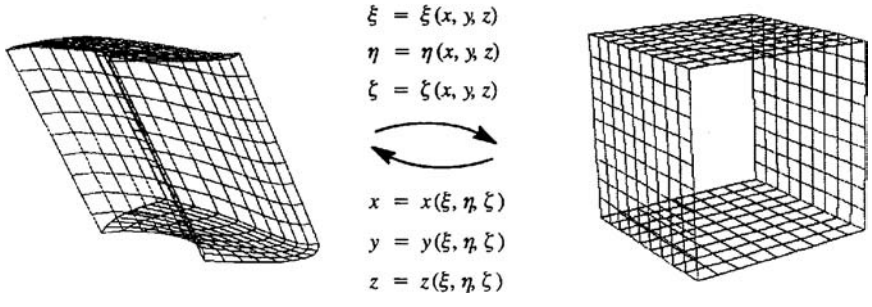


FIGURE 26.1 Physical to computational space mapping.

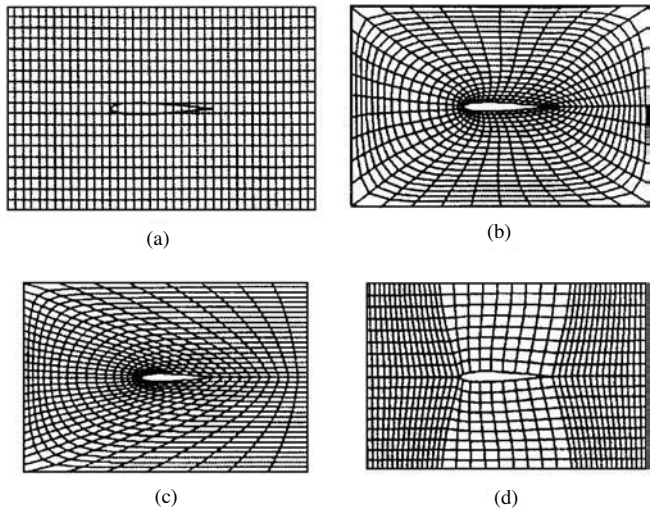


FIGURE 26.2 (a) Cartesian grid, (b) O-type grid, (c) C-type grid, and (d) H-type grid.

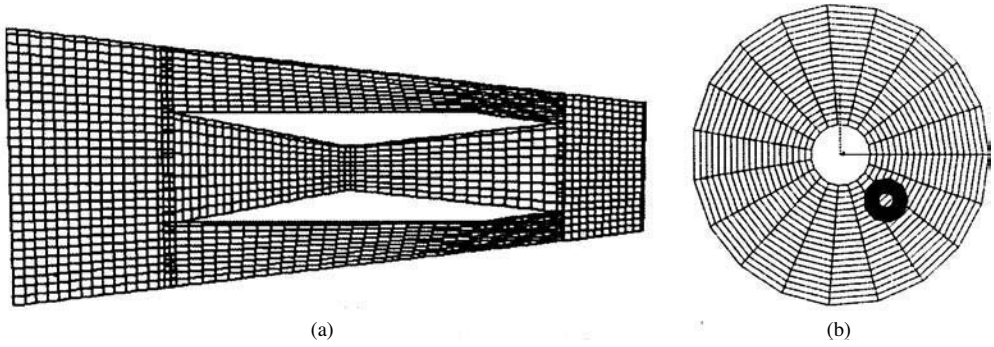


FIGURE 26.3 (a) Multiblock grid showing matching, nonmatching, and overlapping strategies and (b) overlaid chimera grid.

boundary. However, in the H-type grid an airfoil is mapped into the middle of the computational domain as shown in Figure 26.2d.

For complicated geometrical configurations, the physical region is divided into subregions, within each of which a structured grid is generated. These subgrids are commonly referred to as composite grids, and the generation process is referred to as domain decomposition [Thompson 1987b]. The resulting subgrids may be patched together at common interfaces, may be overlapped, or may be overlaid. Self-explanatory pictorial views of domain decomposition strategies allowing patched blocks, overlapping blocks, and overlaid blocks are presented in Figure 26.3a and Figure 26.3b. Overlaid grids are also called chimera grids after the composite monster of Greek mythology.

The use of composite grids allowing patched and overlapped blocks is very common in computational fluid dynamics where geometrical configurations representative of the physical space are extremely complex and/or distinct sets of underlying partial differential equations are to be simulated in different regions. An application of a chimera (overlaid) grid simplifies an overall grid generation and solution process when temporally deforming/moving geometrical components are involved in the simulation. The transfer of solution information at the block interface is very critical for successful simulation. In case of chimera and nonmatching blocks, the interface treatment involves interpolation, and these interpolated physical

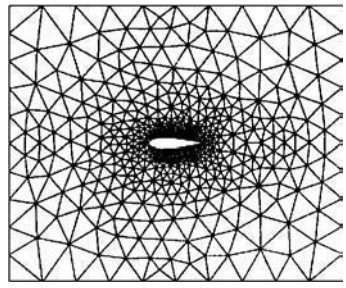


FIGURE 26.4 Unstructured grid.

variables may not satisfy the conservation requirement associated with the overall simulation process, resulting in spurious oscillations in the vicinity of the interface.

In general, the composite grids are presented as

$$\mathbf{r}_{ijkl}(i = 1, \ldots, I\text{MAX}(l), \quad j = 1, \ldots, J\text{MAX}(l), \quad k = 1, \ldots, K\text{MAX}(l), \quad l = 1, \ldots, N\text{BLKS})$$

where i, j, and k identify three curvilinear coordinates, $I\text{MAX}(l)$, $J\text{MAX}(l)$, $K\text{MAX}(l)$ denote the grid dimensions in each direction of block l. NBLKS represents the total number of blocks and vector $\mathbf{r}$ contains the physical variables in x, y, and z direction. In structured grid generation the connections between points are automatically defined from the given (i, j, k) ordering. Such ordering (structure) does not exist in unstructured grids. Unstructured grids are composed of triangles in two dimensions and tetrahedrons in three dimensions. Each elemental volume is called a cell. The grid information is presented by a set of coordinates (nodes) and the connectivity between the nodes. The connectivity table specifies connections between nodes and cells. A pictorial view of a simple unstructured grid is presented in Figure 26.4.

26.1.2 Unstructured Grids

26.1.2.1 Hybrid Grid

A grid formed by a combination of structured–unstructured grids and/or allowing polygonal cells with different numbers of sides is called a hybrid or generalized grid. An usual practice is to generate structured grids near solid components up to desired distance and fill in remaining regions with unstructured (triangular/tetrahedron) grids. A typical hybrid grid is displayed in Figure 26.5.

26.1.3 Generation Process

Regardless of which grid strategy is being considered, creation of a computational grid requires the following:

1. *Computational mapping:* Establishing an appropriate mapping from physical to computational space allowing proper multiblock strategies in case of structured and hybrid grids or establishing an ordering of nodes in case of unstructured grids and hybrid grids.
2. *Geometry generation:* Defining an accurate numerical description of all solid components (surfaces) in conjunction with associated computational mapping criteria and a desired distribution of points.
3. *Computational modeling:* Generating an *appropriate* grid around these surfaces according to some criteria, usually with a specified multiblock strategy, point distribution, smoothness, and orthogonality in case of structured grids and desired background mesh representative of point distribution in case of unstructured grids.

The relationship of geometry to the grid generation process is analogous to the relationship between boundary conditions and the solution of the governing fluid flow equations. An accurate construction of

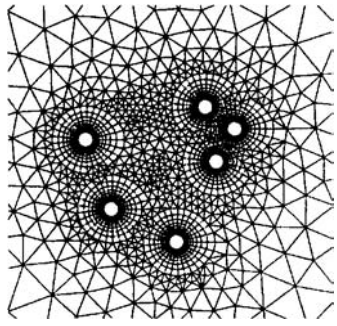


FIGURE 26.5 Hybrid grid.

the geometry with the proper distribution of points usually consumes 85 to 90 percent of the total time spent on the grid generation process. The geometry specification associated with grid generation involves the following:

1. Determine the desired distribution of grid points. This depends upon the expected flow characteristics.
2. Evaluate boundary segments and surface patches to be defined in order to resolve an accurate mathematical description of the geometry in question.
3. Select the geometry tools to be utilized to define these boundary segments/surface patches.
4. Follow an appropriate logical path to blend the aforementioned tasks obtaining the desired discretized mathematical description of the geometry with properly distributed points.

The accuracy of the numerical algorithm depends not only on the formal order of approximations but also on the distribution of grid points in the computational domain. The grid employed can have a profound influence on the quality and convergence rate of the solution. Grid adaption offers the use of excessively fine, computationally expensive, grids by directing the grid points to congregate so that a functional relationship on these points can represent the physical solution with sufficient accuracy.

Underlying principles and methodologies for grid generation and adaption follow.

26.2 Underlying Principles

26.2.1 Terminology and Grid Characteristics

The differencing and solution techniques involving Cartesian (regular) grids are well developed and well understood. The use of structured curvilinear grids (nonorthogonal in most cases) in the numerical solution of PDEs is not, in principle, any more difficult than using Cartesian grids. This is accomplished by transforming the pertinent PDEs from the physical space to computational space. The following notations will be utilized in the development of structured grids and these transformations; detailed mathematical analysis can be found in Thompson et al. [1985]:

$$\begin{aligned}
\mathbf{r} &= (x, y, z) \equiv (x_1, x_2, x_3) && = \text{physical space}\\
\boldsymbol{\Omega} &= (\xi, \eta, \zeta) \equiv (\xi^1, \xi^2, \xi^3) && = \text{computational space}\\
\mathbf{a}_i &= \mathbf{r}_{\xi^i},\ i = 1, 2, 3 && = \text{covariant base vectors}\\
\mathbf{a}^i &= \nabla^{\xi^i},\ i = 1, 2, 3 && = \text{contravariant base vectors}\\
g_{ij} &= \mathbf{a}_i \cdot \mathbf{a}_j\ (i = 1, 2, 3),\ (j = 1, 2, 3) && = \text{covariant metric tensor components}\\
g^{ij} &= \mathbf{a}^i \cdot \mathbf{a}^j\ (i = 1, 2, 3),\ (j = 1, 2, 3) && = \text{contravariant metric tensor components}\\
\sqrt{g} &= a_1 \cdot (a_2 \times a_3) && = \text{Jacobian of transformation}
\end{aligned}$$

Also it can be shown that for any tensor **u**

$$\nabla \cdot \mathbf{u} = \frac{1}{\sqrt{g}} \sum_{i=1}^{3} [(\mathbf{a}_j \cdot \mathbf{a}_k) \cdot \mathbf{u}]_{\xi^i}, \quad (i, j, k) \text{ cyclic} \tag{26.1}$$

and

$$\nabla \cdot \mathbf{u} = \frac{1}{\sqrt{g}} \sum_{i=1}^{3} (\mathbf{a}_j \cdot \mathbf{a}_k) \cdot \mathbf{u}_{\xi^i}, \quad (i, j, k) \text{ cyclic} \tag{26.2}$$

Although Equation 26.1 and Equation 26.2 are equivalent expressions, the numerical representations of these two forms may not be equivalent. The form given by Equation 26.1 is called conservative form and that of Equation 26.2 is called the nonconservative form. Equation 26.1 or Equation 26.2 is utilized for transforming derivatives from physical to computational space. With moving grids the time derivatives also must be transformed using the following equation:

$$\left(\frac{\partial \mathbf{u}}{\partial t}\right)_{\mathbf{r}} = \left(\frac{\partial \mathbf{u}}{\partial t}\right)_{\boldsymbol{\Omega}} - \dot{\mathbf{r}} \cdot \nabla \mathbf{u} \tag{26.3}$$

where $\dot{\mathbf{r}}$ indicates the associated grid speed.

The discretization process associated with the finite-volume scheme employed in unstructured or hybrid grids is also very straightforward. Here, the integral equations are utilized. The edge-based data structure is used for connectivity information and can be easily utilized to compute areas of cells and associated fluxes. The area, for example, of a region bounded by a two-dimensional boundary $\partial\Omega$ is

$$A = \int_{\partial\Omega} x \, \mathrm{d}y \tag{26.4}$$

which can be approximated to

$$A = \sum_{\text{edges}} x \Delta y \tag{26.5}$$

where x and Δy are interpreted as edge quantities. Also, the governing equations are discretized in similar fashion. For example, consider an integral form of the Navier-Stokes equation without body force as follows:

$$\frac{\partial}{\partial t} \int_{v} \mathbf{Q} \, \mathrm{d}v + \oint_{\partial\Omega} \mathbf{F}(Q) \cdot \mathbf{n} \, \mathrm{d}s = \oint_{\partial\Omega} \mathbf{F}^{v}(Q) \cdot \mathbf{n} \, \mathrm{d}s \tag{26.6}$$

where **n** is the outward normal to the control volume with components n_x and n_y in the x and y directions. The domain of interest is discretized into a set of nonoverlapping polygons (unstructured or hybrid grid), and the cell-averaged variables are stored at the cell center. For each cell the semidiscretized form of the governing Equation 26.6 can be written as

$$V_i \frac{\partial Q}{\partial t} = -\sum_{j=1}^{k} F_{ij} \cdot n_j \, \mathrm{d}s + \sum_{j=1}^{k} F_{ij}^{v} \cdot n_j \, \mathrm{d}s \tag{26.7}$$

where j varies over the cell faces, and F_{ij} and F_{ij}^{v} are the convective and viscous part of the numerical flux at jth edge of cell i. Detailed analysis and description of this discretization process can be found in Weatherill [1990] and Koomullil et al. [1996b].

On structured grids the definition of an order of a difference representation is integrally tied to point distribution functions, commonly referred to as stretching functions. The order is determined by the error behavior as the spacing varies with the points fixed in a certain distribution, either by increasing the number of points or by changing a parameter in the distribution. Actually, a global order is not meaningful in case of nonuniform structured grids. The order is relevant only locally in regions where the spacing does in fact

decrease as the point distribution changes. Looking at the truncation error analysis involving nonuniform structured curvilinear grids, the following grid requirements can be outlined: (1) The structured grid system must be either a right-handed system [$\sqrt{g} > 0$ for all (i, j, k)] or a left-handed system [$\sqrt{g} < 0$ for all (i, j, k)]. (2) Application of the distribution function with bounds on higher order derivatives does not change the formal order of approximation. The following functions involving exponential and hyperbolic tangent or hyperbolic sine stretching functions are widely utilized as distribution functions for distributing N points:

$$x(\xi) = \frac{e^{\alpha(s-1)} - 1}{e^{\alpha} - 1} \qquad \text{or} \qquad x(\xi) = 1 - \frac{\tanh(\alpha(1-s))}{\tanh(\alpha)} \tag{26.8}$$

where

$$s = (\xi - 1)/(N - 1) \quad 1 \le \xi \le N.$$

(3) Numerical derivative evaluation of the distribution function is preferred, i.e., instead of analytical definition of x_ξ, where $x_\xi = [(x_{i+1} - x_{i-1})/2]$. (4) The truncation error is inversely proportional to the sine of the angle between grid lines. This in turn indicates that the mildly skewed grid does not increase truncation error significantly. In fact, as a rule of thumb, nonorthogonality resulting in an angle between grid lines not less than 45° does not increase truncation errors significantly.

26.2.2 Geometry Preparation

The geometry preparation is the most critical and labor intensive part of the overall grid generation process. Most of the geometrical configurations of interest to practical engineering problems are designed in the computer-aided design/computer-aided manufacturing (CAD/CAM) system. There are numerous geometry output formats which require the designer to spend a great deal of time manipulating geometrical entities in order to achieve a useful sculptured geometrical description with appropriate distribution for grid points. Also, there is a danger of loosing fidelity of the underlying geometry in this process. The desired point distribution on the boundary segment/surface patch is achieved by computing a concentration array of unit length. The concentration array is computed using specified spacing and by selecting an exponential or hyperbolic tangent stretching function. The geometry preparation involves the discrete-sculptured definitions of all outer boundaries/surfaces associated with the domain of interest. In case of unstructured or hybrid grids, the geometry preparation involves the definition of discrete points on the boundaries associated with the domain. The following definitions will be utilized in this development.

Definition 26.1 Given a set of points on a curve with physical Cartesian coordinates $(x_i, y_i, z_i), i = 1i, i1 + 1, \ldots, i2$. A number sequence $r = (r_1, r_2, \ldots, r_n)$, with $0 \le r_j \le 1$, $r_1 = 0, r_n = 1, r_i \le r_j$ for all $i < j$ represents the distribution of points, such that there exists a one-to-one correspondence between the element r_j of r and the triplet (x_j, y_j, z_j). This number sequence r is called a curve distribution. For example, the normalized chord length with $r_{i1} = 0$ and

$$r_i = \frac{\sum_{u=i1+1}^{i} \sqrt{(x_u - x_{u-1})^2 + (y_u - y_{u-1})^2 + (z_u - z_{u-1})^2}}{\sum_{u=i1+1}^{i2} \sqrt{(x_u - x_{u-1})^2 + (y_u - y_{u-1})^2 + (z_u - z_{u-1})^2}} \tag{26.9}$$

where $i = i1 + 1, \ldots, i2$ satisfies the definition of a curve distribution.

Definition 26.2 Given a set of points on a surface with physical Cartesian coordinates $(x_{ij}, y_{ij}, z_{ij}), i = 1i, i1 + 1, \ldots, i2;\ j = j1, j1 + 1, \ldots, j2$. A mesh (s_{ij}, t_{ij}), $i = 1i, i1 + 1, \ldots, i2;\ j = j1, j1 + 1, \ldots, j2$, is called a surface distribution mesh if $(s_{i1j}, \ldots, s_{i2j})$ for $j = j1, \ldots, j2$ represents the curve distribution for the curve $((x_{ij}, y_{ij}, z_{ij}),\ i = i1, i1 + 1, \ldots, i2)$ for $j = j1, j1, \ldots, j2)$ and $(t_{ij1}, \ldots, t_{ij2})$ for $i = i1, \ldots, i2$ represents the curve distribution for the curve $((x_{ij}, y_{ij}, z_{ij}),\ j = j1, j1 + 1, \ldots, j2)$, for $i = i1, i1 + 1, \ldots, i2)$.

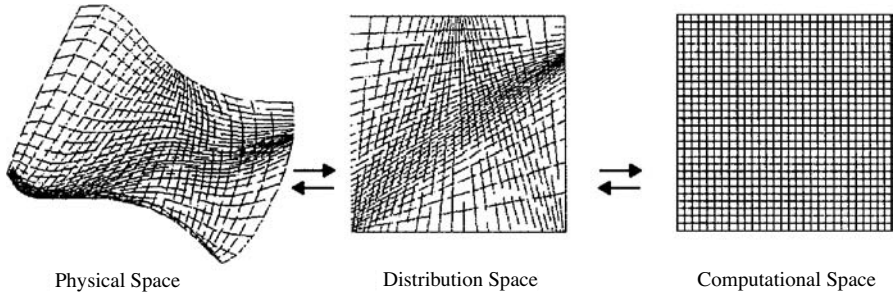


FIGURE 26.6 Relationship between physical space, distribution mesh, and computational space.

Also, there exists a one-to-one correspondence between the physical domain and the surface distribution mesh and between the surface distribution mesh and the computational domain. These relations are demonstrated in Figure 26.6.

Definition 26.3 Let $(X_{ijk}, Y_{ijk}, Z_{ijk})$, $i = 1, 2, \ldots, N$; $j = 1, 2, \ldots, M$; $k = 1, 2, \ldots, L$ be a single block three-dimensional structured grid. A mesh $(s_{ijk}, t_{ijk}, q_{ijk})$, $i = 1, 2, \ldots, N$; $j = 1, 2, \ldots, M$; $k = 1, 2, \ldots, L$ is called a volume distribution mesh if $(\overline{s}_{ij\overline{k}}, \overline{t}_{ij\overline{k}})$ $i = 1, 2, \ldots, N$; $j = 1, 2, \ldots, M$ represents the surface distribution mesh for the surface $(\overline{X}_{ij\overline{k}}, Y_{ij\overline{k}}, Z_{ij\overline{k}})$, $\overline{k} = 1, 2, \ldots, L$, $(t_{\overline{i}jk}, q_{\overline{i}jk})$ represents the surface distribution mesh for the surface $(X_{\overline{i}jk}, \overline{Y}_{\overline{i}jk}, \overline{Z}_{\overline{i}jk})$, for all $\overline{i}$ and $(s_{\overline{i}jk}, q_{\overline{i}jk})$ represents the surface distribution mesh for the surface $(\overline{X}_{i\overline{j}k}, \overline{Y}_{i\overline{j}k}, \overline{Z}_{i\overline{j}k})$, for all $\overline{j}$.

26.2.3 Structured Grid Generation

26.2.3.1 Algebraic Generation Methods

An algebraic 3-D generation system based on transfinite interpolation (using either Lagrange or Hermite interpolation) is widely utilized for grid generation [Gordon and Thiel 1982, Soni 1992b]. The interpolation, in general complete transfinite interpolation from all boundaries, can be restricted to any combination of directions or lesser degrees of interpolation, and the form (Lagrange, Hermite, or incomplete Hermite) can be different in different directions or in different blocks. The blending functions can be linear or, more appropriately, based on the distribution surface/volume mesh. Hermite interpolation, based on cubic blending functions, allows orthogonality at the boundary. Incomplete Hermite uses quadratic functions and, hence, can give orthogonality at any one of two opposing boundaries, whereas Lagrange, with its linear functions, does not give orthogonality.

The transfinite interpolation is accomplished by the appropriate combination of 1-D projectors F for the type of interpolation specified. (Each projector is simply the 1-D interpolation in the direction indicated.) For interpolation from all sides of the section, if all three directions are indicated and the section is a volume, this interpolation is from all six sides, and the combination of projectors is the Boolean sum of the three projectors,

$$F_1 \oplus F_2 \oplus F_3 \equiv F_1 + F_2 + F_3 - F_1F_2 - F_2F_3 - F_3F_1 + F_1F_2F_3 \tag{26.10}$$

With interpolation in only the two directions j and k, or if the section is a surface on which ξ^i is constant, the combination is the Boolean sum of F_j and F_k

$$F_j \oplus F_k \equiv F_j + F_k - F_jF_k, \quad (i, j, k) \text{ cyclic} \tag{26.11}$$

With interpolation in only a single direction i, or if the section is a line on which ξ^i varies, the interpolation is between the two sides on which ξ^i is constant using only the single projector F_i.

Blocks can be divided into subblocks for the purpose of generation of the algebraic grid. Point distributions on the sides of the subblocks can either be specified or automatically generated by transfinite interpolation from the edge of the side. This allows additional control over the grid in general configurations and is particularly useful in cases where point distributions need to be specified in the interior of a block or to prevent grid overlap in highly curved regions. This also allows points in the interior of the field to be excluded if desired, e.g., to represent holes in the field.

26.2.3.2 Elliptic Generation Method

An elliptic grid generation system [Thompson 1987b] commonly used is

$$\sum_{i=1}^{3}\sum_{l=1}^{3} g^{il} r_{\xi^i \xi^l} + \sum_{l=1}^{3} g^{ll} P_l r_{\xi^l} = 0 \tag{26.12}$$

where the g^{il}, the elements of the contravariant metric tensor, can be evaluated as

$$g^{il} = \frac{1}{g}(g_{jm}g_{kn} - g_{jn}g_{km}), \quad (i = 1,2,3), \quad (l = 1,2,3) \tag{26.13}$$
$$(i, j, k) \text{ cyclic}, \quad (l, m, n) \text{ cyclic}$$

where g is the square of the Jacobian.

The P_n are the *control functions*, which serve to control the spacing and orientation of the grid lines in the field. The first and second coordinate derivatives are normally calculated using second-order central differences. One-sided differences dependent on the sign of the control function P_n (backward for $P_n < 0$ and forward for $P_n > 0$) are useful to enhance convergence with very strong control functions. The control functions are evaluated either directly from the initial algebraic grid and then smoothed or by interpolation from the boundary point distributions.

The three components of the elliptic grid generation system provide a set of three equations that can be solved simultaneously at each point for the three control functions $P_n (n = 1, 2, 3)$, with the derivatives here represented by central differences.

The elliptic generation system is solved by point successive over relaxation (SOR) iteration using a field of locally optimum acceleration parameters. These optimum parameters make the solution robust and capable of convergence with strong control functions.

Control functions can also be evaluated on the boundaries using the specified boundary point distribution in the generation system, with certain necessary assumptions (orthogonality at the boundary) to eliminate some terms, then they can be interpolated from the boundaries into the field. More general regions can, however, be treated by interpolating elements of the control functions separately. Thus, control functions on a line on which ξ^n varies can be expressed as

$$P_n = A_n + \frac{S_n}{\varrho_n} \tag{26.14}$$

where A_n is the logarithmic derivative of the arc length, S_n is the arc length spacing, and ϱ_n is the radius of curvature of the surface on which ξ^n is constant.

A second-order elliptic generation system allows either the point locations on the boundary or the coordinate line slope at the boundary to be specified, but not both. It is possible, however, to iteratively adjust the control functions in the generation system until not only a specified line slope, but also the spacing of the first coordinate surface off the boundary is achieved, with the point locations on the boundary specified. The extent of the orthogonality into the field can also be controlled. This orthogonality feature is also applicable on specified grid surfaces within the field, allowing grid surfaces in the field to be kept fixed while retaining continuity of slope of the grid line crossing the surface. This is quite useful in controlling the skewness of grid lines in some troublesome areas. Alternatively, boundary orthogonality can be achieved through Neumann boundary conditions, which allow the boundary points to move over a surface spline.

The boundary locations are located by Newton iteration on the spline to be at the foot of normals to the adjacent field points.

26.2.3.3 Hyperbolic Generation Method

It is also possible to base a grid generation system on hyperbolic PDEs rather than elliptic equations. In this case, the grid is generated by numerically solving a hyperbolic system [Steger and Chaussee 1980], marching in the direction of one curvilinear coordinate between two boundary curves in two dimensions or between two boundary surfaces in three dimensions. The hyperbolic system, however, allows only one boundary to be specified and is therefore of interest only for use in calculation on physically unbounded regions where the precise location of a computational outer boundary is not important. The hyperbolic grid generation system has the advantage of being generally faster than elliptic generation systems, but, as just noted, is applicable only to certain configurations. Hyperbolic generation systems can be used to generate orthogonal grids. In two dimensions, the condition of orthogonality is simply $g_{12} = 0$.

If either the cell area $\sqrt{g}$ or the cell diagonal length (squared) $g_{11} + g_{22}$ is a specified function of the curvilinear coordinates, i.e.,

$$\sqrt{g} = F(\xi, \eta) \qquad \text{or} \qquad g_{11} + g_{22} = F(\xi, \eta) \tag{26.15}$$

then the system consisting of $g_{12} = 0$ and either of the preceding two equations is hyperbolic. Since the system is hyperbolic, a noniterative marching solution can be constructed proceeding in one coordinate direction away from a specified boundary.

26.2.3.4 Multiblock Systems

Although in principle it is possible to establish a correspondence between any physical region and a single empty logically rectangular block for general 3-D configurations, the resulting grid is likely to be much too skewed and irregular to be usable when the boundary geometry is complicated. A better approach with complicated physical boundaries is to segment the physical region into contiguous subregions, each bounded by six curved sides (four in 2-D) and each of which transforms to a logically rectangular block in the computational region. Each subregion has its own curvilinear coordinate system, irrespective of that in the adjacent subregions (see Figure 26.7). This then allows both the grid generation and numerical solutions on the grid to be constructed to operate in a logically rectangular computational region, regardless of the shape or complexity of the full physical region. The full region is treated by performing the solution

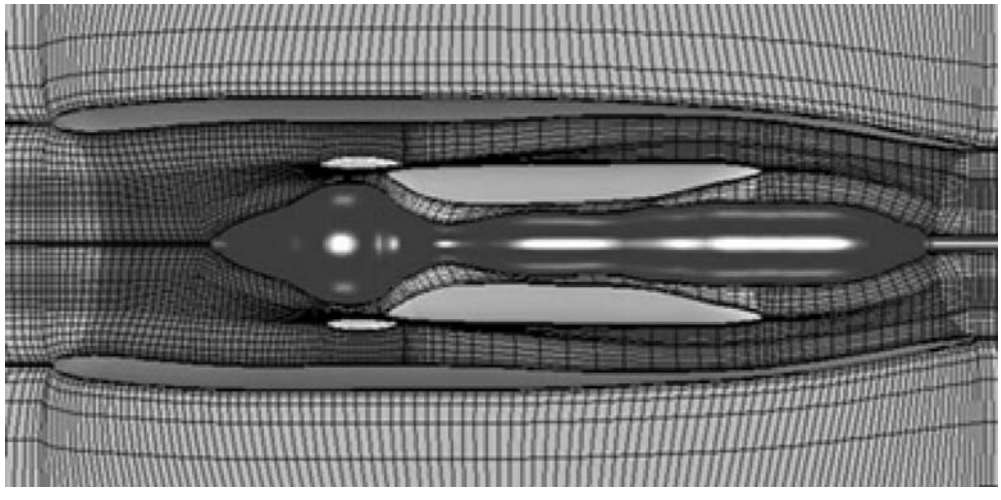


FIGURE 26.7 Multiblock grid.

operations in all of the logically rectangular computational blocks. With the composite framework, PDE solution procedures written to operate on logically rectangular regions can be incorporated into a code for general configurations in a straightforward manner, since the code only needs to treat a rectangular block. The entire physical field then can be treated in a loop over all the blocks. Transformation relations for PDEs are covered in detail in Thompson et al. [1985]. Discretization error related to the grid is covered in Thompson and Mastin [1983]. The evaluation and control of grid quality is an ongoing area of active research [Gatlin et al. 1991].

Grid lines at the interfaces may meet with complete continuity, with or without slope continuity, or may not meet at all. Complete continuity of grid lines across the interface requires that the interface [Thompson 1987a] be treated as a branch cut on which the generation system is solved just as it is in the interior of blocks. The interface locations are then not fixed, but are determined by the grid generation system. This is most easily handled in coding by providing an extra layer of points surrounding each block. Here, the grid points on an interface of one block are coincident in physical space with those of another interface of the same or another block, and also the grid points on the surrounding layer outside the first interface are coincident with those just inside the other interface, and vice versa. This coincidence can be maintained during the course of an iterative solution of an elliptic generation system by setting the values on the surrounding layers equal to those at the corresponding interior points after each iteration. All of the blocks are thus iterated to convergence, so that the entire composite grid is generated at once. The same procedure is followed by PDE solution codes on the block-structured grid.

26.2.3.5 Chimera Grids

The *chimera* (overlaid) grids [Belk 1995, Meakin 1991, Benek et al. 1985] are composed of completely independent component grids which may even overlap other component boundary elements, creating holes in the component grids. This requires flagging procedures to locate grid points that lie out of the field of computation, but such holes can be handled even in tridiagonal solvers by placing 1s at the corresponding positions on the matrix diagonal and all 0s off the diagonal. These overlaid grids also require interpolation to transfer data between grids, and that subject is the principal focus of effort with regard to the use of this type of composite grid.

26.2.3.6 Adaptive Grid Generation

With structured grids, the adaptive strategy based on redistribution is by far the most simple to implement, requiring only the regeneration of the grid and interpolation of flow properties at the new grid points at each adaptive stage without modification of the flow solver unless time accuracy is desired. Time accuracy can be achieved, as far as the grid is concerned, by simply transforming the time derivatives, thus adding convectivelike terms that do not alter the basic conservation form of the PDEs.

Adaptive redistribution of points traces its roots to the principle of equidistribution of error [Brackbill 1993, Soni et al. 1993] by which a point distribution is set so as to make the product of the spacing and a weight function constant over the points

$$w\Delta x = \text{const}$$

With the point distribution defined by a function ξ_i, where ξ varies by a unit increment between points, the equidistribution principle can be expressed as

$$w x_\xi = \text{const} \tag{26.16}$$

This one-dimensional equation can be applied in each direction in an alternating fashion. A direct extension to multiple dimensions using algebraic, variational, and elliptic systems has been developed.

The weight function is usually formulated by utilizing scaled gradients and curvatures of the solution variables considered for adaption.

The control of the characteristics and distribution of a grid system can be achieved by varying the values of the control functions P_l in Equation 26.12. The application of the one-dimensional form of

Equation 26.12 with Equation 26.16 results in the definition of control functions in three dimensions,

$$P_i = \frac{W_{\xi^i}}{W} \quad (i = 1, 2, 3) \tag{26.17}$$

These control functions were generalized by Eiseman [1985] as

$$P_i = \sum_{j=1}^{3} \frac{g^{ij}}{g^{ii}} \frac{(W_i)_{\xi^i}}{W_i} \quad (i = 1, 2, 3) \tag{26.18}$$

In order to conserve the geometrical characteristics of the existing grid the definition of control functions is extended as

$$P_i = (P_{\text{initial geometry}})_i + c_i(P_{wt}) \quad (i = 1, 2, 3) \tag{26.19}$$

where $(P_{\text{initial geometry}})$ is the control function based on initial grid geometry, P_{wt} is the control function based on gradient of flow parameter, and c_i is the constant weight factors.

These control functions are evaluated based on the current grid at the adaptation step. This can be formulated as

$$P_i^{(n)} = P_i^{(n-1)} + c_i(P_{wt})^{(n-1)} \quad (i = 1, 2, 3) \tag{26.20}$$

where

$$P_i^{(1)} = (P_{\text{initial geometry}})_i^{(0)} + c_i(P_{wt})^{(0)} \quad (i = 1, 2, 3) \tag{26.21}$$

26.2.4 Unstructured Grid Generation

26.2.4.1 The Delaunay Triangulation

Dirichlet, in 1850, first proposed a method whereby a domain could be systematically decomposed into a set of packed convex polyhedra. For a given set of points in space, $\{P_k\}, k = 1, \ldots, K$, the regions $\{V_k\}, k = 1, \ldots, K$, are the territories which can be assigned to each point P_k, such that V_k represents the space closer to P_k than to any other point in the set. Clearly, these regions satisfy

$$V_k = \{P_i : |p - P_i| < |p - P_j|\} \; \forall \; j \neq i \tag{26.22}$$

This geometrical construction of tiles is known as the Dirichlet tessellation or Voronoi [1908] diagram. This tessellation of a closed domain results in a set of nonoverlapping convex polyhedra, called Voronoi regions, covering the entire domain. If all point pairs which have some segment of a Voronoi boundary in common are joined, the result is a triangulation of the convex hull of the set of points $\{P_k\}$. This triangulation is known as the *Delaunay triangulation* [Baker 1990, George and Hermeline 1992]. The definition is valid for n-dimensional space.

From the preceding discussion, it is apparent that in two dimensions a line segment of the Voronoi diagram is equidistant from the two points it separates. Hence, the vertices of the Voronoi diagram must be equidistant from each of the three nodes which form the Delaunay triangles. Clearly, it is possible to construct a circle, centered at a Voronoi vertex, which passes through the three points, which form a triangle. Furthermore, it is evident that, given the definition of Voronoi line segments and regions, no circle can contain any point. This latter condition is referred to as the in-circle criterion.

26.2.4.2 Advancing Front Procedure

The advancing front procedure is based on the method originally proposed in Peraire et al. [1987] for two dimensions and then extended to three dimensions in [Peraire et al. 1988, 1990]. The advocated approach is regarded as a generalization of the advancing front technique [George 1971, Lo 1985] with the distinctive feature that elements, i.e., triangles or tetrahedra, and points are generated simultaneously. This enables the

generation of elements of variable size and stretching and differs from the approach followed in tetrahedral generators which are based on Delaunay concepts [Baker 1990, Cavendish et al. 1985], which generally connect grid points which have already been distributed in space.

The generation problem consists of subdividing an arbitrarily complex domain into a consistent assembly of elements. The consistency of the generated mesh is guaranteed if the generated elements cover the entire domain and the intersection between elements occurs only on common points, sides, or triangular faces in the three-dimensional case. The final mesh is constructed in a bottom-up manner. The process starts by discretising each boundary curve. Nodes are placed on the boundary curve components and then contiguous nodes are joined with straight line segments. In later stages of the generation process, these segments will become sides of some triangles. The length of these segments must therefore, be consistent with the desired local distribution of mesh size. This operation is repeated for each boundary curve in turn.

The next stage consists of generating triangular planar faces. For each two-dimensional region or surface to be discretized, all of the edges produced when discretizing its boundary curves are assembled into the so-called initial front. The relative orientation of the curve components with respect to the surface must be taken into account in order to give the correct orientation to the sides in the initial front. The front is a dynamic data structure which changes continuously during the generation process. At any given time, the front contains the set of all of the sides which are currently available to form a triangular face. A side is selected from the front and a triangular element is generated. This may involve creating a new node or simply connecting to an existing one. After the triangle has been generated, the front is updated and the generation proceeds until the front is empty. The size and shape of the generated triangles must be consistent with the local desired size and shape of the final mesh. In the three-dimensional case, these triangles will become faces of the tetrahedra to be generated later.

The geometrical characteristics of a general mesh are locally defined in terms of certain mesh parameters. If $N = (2 \text{ or } 3)$ is the number of dimensions, then the parameters used are a set of N mutually orthogonal directions $\boldsymbol{\alpha}_i, i = 1, \ldots, N$, and N associated element sizes $\delta_i, i = 1, \ldots, N$. Thus, at a certain point, if all N element sizes are equal, the mesh in the vicinity of that point will consist of approximately equilateral elements. To aid the mesh generation procedure, a transformation T which is a function of $\boldsymbol{\alpha}_i$ and δ_i is defined. This transformation is represented by a symmetric $N \times N$ matrix and maps the physical space onto a space in which elements, in the neighborhood of the point being considered, will be approximately equilateral with unit average size. This new space will be referred to as the normalized space. For a general mesh this transformation will be a function of position. The transformation T is the result of superimposing N scaling operations with factors $1/\delta_i$ in each $\boldsymbol{\alpha}_i$ direction. Thus,

$$T(\boldsymbol{\alpha}_i, \delta_i) = \sum_{i=1}^{N} \frac{1}{\delta_i} \boldsymbol{\alpha}_i \otimes \boldsymbol{\alpha}_i \tag{26.23}$$

where $\otimes$ denotes the tensor product of two vectors.

26.2.4.3 Grid Adaption Methods

For the solution adaptive grid generation procedure, an error indicator is required that detects and locates appropriate features in the flowfield. In order to provide flexibility in isolating varying features, multiple error indicators are used. Each can isolate a particular type of feature. The error indicators are usually set to the negative and positive components of the gradient in the direction of the velocity vector as given by

$$\begin{aligned} e_1 &= \min[\mathbf{V} \cdot \nabla(\mathbf{u}), 0] \\ e_2 &= \max[\mathbf{V} \cdot \nabla(\mathbf{u}), 0] \end{aligned} \tag{26.24}$$

and the magnitude of the gradient in all directions normal to the velocity vector as given by

$$e_3 = |\nabla \mathbf{u} - \mathbf{V}(\mathbf{V} \cdot \nabla \mathbf{u})/\mathbf{V} \cdot \mathbf{V}| \tag{26.25}$$

Where $\mathbf{V}$ is the velocity vector and $\mathbf{u}$ is any suitable flow property. Typically, density is used as the basis for the error indicator. The first two error indicators represent expansions and compressions in the flow

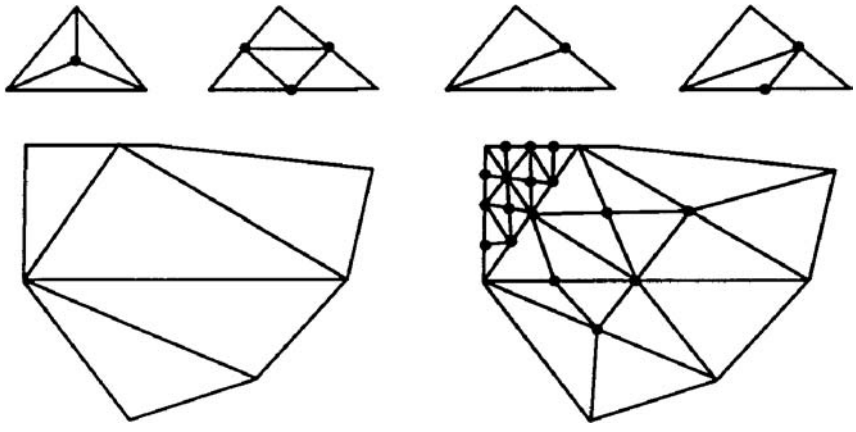


FIGURE 26.8 Types of h-refinement in two dimensions.

direction and the third represents gradients normal to the flow direction. The indicators can be scaled by the relative element size. Length scaling can improve detection of weak features on a coarse grid with the present procedure. Each error indicator is treated independently, allowing particular features in the flowfield to be isolated. For each error indicator, an error is determined from

$$e_{\text{lim}} = e_m + c_{\text{lim}} \cdot e_s \tag{26.26}$$

where e_{lim} is the error limit, e_m is the mean of the error indicator, e_s is the standard deviation of the error indicator, and c_{lim} is a constant. Typically a value near 1 is used for the constant. The error indicators are used to control the local reduction in relative element size during grid generation.

One of the advantages of an unstructured grid is that it provides a natural environment for grid adaptation using h-refinement or mesh enrichment. Points can be added to the mesh with the consequence that new elements are formed and only local modifications to the connectivity matrix need to be made. In addition, no modification or special treatments are required within the solution algorithm provided that on enriching the mesh the distribution of elements and points remains smooth. Once the regions for enrichment are determined and individual elements are identified there are a number of strategies for adding points. The most suitable methods attempt to ensure smoothness of the enriched mesh, and in this respect local refinement strategies can prove to be useful. Some examples of point enrichment are given in Figure 26.8.

In addition to h-refinement, node movement has been found to be necessary for an efficient implementation of grid adaptation. Node movement can be applied in the form

$$\mathbf{r}_0^{n+1} = \mathbf{r}_0^n + \omega_i \frac{\sum_{i=1}^{M} C_{i0} \left(\mathbf{r}_i^n - \mathbf{r}_0^{ni}\right)}{\sum_{i=1}^{M} c_{i0}} \tag{26.27}$$

where $\mathbf{r} = (x, y)$, $\mathbf{r}_0^{n+1}$ is the position of node 0 at relaxation level $n+1$, C_{i0} is the adaptive weight function between nodes i and 0, and ω is the relaxation parameter. An adaptive weight function C_{i0} is used which takes the form

$$C_{i0} = k_1 + k_2 \left| \frac{\phi_i - \phi_0}{\phi_i + \phi_0} \right| \tag{26.28}$$

where ϕ is the driving variable e.g., pressure, density, Mach number, etc.; k_1 and k_2 are constants, k_1 acts to damp out noise; and k_2 amplifies the gradients along edges. In practice, this is implemented in a form which guarantees positive area cells after movement, even in regions close to a wall, which for viscous grids can have very small volumes.

26.3 Best Practices

In the last few years, numerical grid generation has evolved as an essential tool in obtaining the numerical solution of the partial differential equations of fluid mechanics. A multitude of techniques and computer codes have been developed to support multiblock structured and unstructured grid generation associated with complex configurations. Structured grid generation methodologies can be grouped in two main categories: direct methods, where algebraic interpolation techniques are utilized, and indirect methods, where a set of partial differential equations is solved. Both of these techniques are utilized either separately or in combination, to efficiently generate grids in the aforementioned codes.

In algebraic methods the most widely used technique is transfinite interpolation. Historically, application of algebraic methods in grid generation has progressed as follows. In the 1970s (and early 1980s) the algebraic methods based on Lagrange and Hermite (mostly cubic) interpolation methods in tensor product form or Coon's patching, commonly referred to as transfinite interpolation technique form, and parametric spline (mostly cubic natural splines) were utilized to construct an initial grid for the iterative grid evaluation associated with a set of partial differential equations (mostly elliptic equations). In the 1980s (and 1990–1991), the development of high-powered graphics workstations along with the application of Bezier B-spline curve/surface definition in a control point form revolutionized the grid generation process with graphically interactive generation strategies and grid quality (smoothness-orthogonality with precise distribution improvements) with fast and efficient parametric curve/surface description based on basis splines. The parametric based nonuniform rational B-spline (NURBS) is a widely utilized representation for geometrical entities in computer-aided geometric design (CAGD) and CAD/CAM systems [Yu 1995]. The convex hull, local support, shape preserving forms, and variation diminishing properties of NURBS are extremely attractive in engineering design applications and in geometry-grid generation. In fact, the NURBS representation is becoming the de facto standard for the geometry description in most of the modern grid generation systems. Recently, the research concentration in algebraic grid generation is placed on utilizing CAGD techniques for efficient and accurate geometric modeling (boundary/surface grid generation). The development of NASA initial graphics exchange specification (IGES) standard [Blake and Chou 1992]; NURBS data structure in the National Grid Project [Thompson 1993]; DTNURBS library and its implementation in various grid systems; the grid systems NGP, ICEM, GRIDGEN, IGG, GENIE++; and computer aided grid interface (CAGI) system is just a partial list of the outcome of this research concentration.

The best practice in grid generation is to transform all geometrical entities associated with the complex configuration under consideration into parametric control points based on NURBS representation allowing standard data structure. The grid generation algorithms are then tailored to exhibit NURBS representation in the generation process. An overall generation process is usually based on utilizing the best features of direct and indirect methods in case of structured grids and Delaunay triangulation and advancing front methods in case of unstructured grids.

26.3.1 Structured Grid Generation

The following observations and evaluations on the structured grid generation methodologies are important to consider before the development of overall grid generation process: algebraic systems are fast and economical; precise spacing control (a well-distributed grid) is always achieved with algebraic systems; grid generation by elliptic systems is always smooth; algebraic systems may cause grids to overlap; however, elliptic systems resist grid line overlapping; the control functions can be formulated to achieve boundary orthogonality and spacing control (near solid boundary surfaces) by elliptic generation systems; the control functions can be formulated to accomplish field orthogonality in a given computational direction (ξ, ς, or η) and spacing control by elliptic generation systems by iteratively updating various terms in the generation system. This is very time consuming especially in three-dimensional problems. Algebraic systems require a high degree of understanding and visual user interaction. However, elliptic systems can be readily adaptable for generalization. This is extremely useful in grid adaptations. The hyperbolic systems preserve

the orthogonality at the solid boundary and the point distribution in the field. However, their applicability is restricted to external flows where the accurate geometrical shape of the outer boundaries/surfaces is not important as long as their location is a certain distance away from the body. Also in three-dimensional applications of hyperbolic systems the grid quality is directly influenced by the characteristics of the surfaces associated with the computational domain.

26.3.1.1 Transfinite Interpolation Method

In general, the algebraic methods are based on utilizing tensor product form of interpolation (in case of surface generation) and transfinite interpolation (in case of 2-D or full 3-D volume grid generation). Define a one-dimensional interpolation projector as follows:

$$T[r(s)] = \sum_{k=0}^{q} \sum_{j=0}^{p} \phi_{j,k}(s) r_j^{(k)} \tag{26.29}$$

where the parameter s is such that $0 \leq s \leq 1$, $\phi_{j,k}$ are the blending functions, and $r_j(k)$ is the kth derivative of the variable r at parametric location $s_j (0 = s_0 \leq s_1 \leq s_2 \cdots \leq s_p = 1)$. The following example clarifies Equation 26.29.

Example 26.1

Let

$$\begin{aligned} q = 0, \qquad p = 1 \\ \phi_{0,0}(s) = (1 - s), \phi_{1,0}(s) = s \end{aligned}$$

then

$$T[r(s)] = \phi_{0,0}(s) r_0^0 + \phi_{1,0}(s) r_1^0 = (1 - s) r_0^0 + s r_1^0$$

defines a straight line between two points r_0^0 and r_1^0.

Example 26.2

Let

$$\begin{aligned} q = 1, \qquad p = 1 \\ \phi_{0,0}(s) &= 2s^3 - 3s^2 + 1 \\ \phi_{1,0}(s) &= -2s^3 + 3s^2 \\ \phi_{0,1}(s) &= s^3 - 2s^2 + 1 \\ \phi_{1,1}(s) &= s^3 - s^2 \\ T[r(s)] &= \phi_{0,0}(s) r_0^0 + \phi_{1,0}(s) r_1^0 + \phi_{0,1}(s) r_0^1 + \phi_{1,1}(s) r_1^1 \end{aligned} \tag{26.30}$$

defines a Hermite cubic polynomial between two points r_0^0 and r_1^0 with specified slopes r_0^1 and r_1^1 at the respective end points. The linear interpolation and Hermite cubic interpolation are widely applied in grid generation.

For 2-D grid generation the transfinite interpolation (TFI) method is defined as follows:

$$T[r(s_{ij}, t_{ij})] = T_I[r(s_{ij}, t_{ij})] \oplus T_J[r(s_{ij}, t_{ij})]$$

where T_I is a one-dimensional interpolation projector applied in the $i(\xi)$ direction keeping t_{ij} fixed and T_J is the one-dimensional interpolation projector applied in the $j(\eta)$ direction keeping s_{ij} fixed. $T[r] = (T_I + T_J - T_I T_J)[r]$, and (s_{ij}, t_{ij}) is the distribution mesh for the 2-D grid configuration under consideration. The $T_I T_J$ represents the tensor product of interpolation in both the I and J directions.

The transfinite interpolation method for 3-D grid generation can be defined as

$$T[r(s_{ijk}, t_{ijk}, q_{ijk})] = T_I[r(s_{ijk}, t_{ijk}, q_{ijk})] \oplus T_J[r(s_{ijk}, t_{ijk}, q_{ijk})] \oplus T_K[r(s_{ijk}, t_{ijk}, q_{ijk})] \tag{26.31}$$

where T_I is a one-dimensional interpolation projector applied in the $t(\xi)$ direction keeping t_{ijk} and q_{ijk} fixed, T_J and T_K are similarly defined. Here $(s_{ijk}, t_{ijk}, q_{ijk})$ represents volume distribution mesh associated with 3-D grid generation.

For example, if (s_{ij}, t_{ij}) represents an $N \times M$ size distribution mesh, $((X_{i1}, Y_{i1}),\ (X_{iM}, Y_{iM}))$, $i = 1, 2, \ldots, N$; and $((X_{1j}, Y_{1j}), (X_{Nj}, Y_{Nj}))$, $j = 1, 2 \ldots, M$ boundaries are known, T_I is selected as a linear interpolation projector and T_J is selected as a Hermite interpolation projector and then

$$T_I[r(s_{ij}, t_{ij})] = (1 - s_{ij})r_{1j} + s_{ij}r_{Nj} \tag{26.32}$$

and

$$T_J[r(s_{ij}, t_{ij})] = \Phi_{00}(t_{ij})r_{i1} + \Phi_{1,0}(t_{ij})r_{iM} + \Phi_{0,1}(t_{ij})\left.\frac{\partial r}{\partial \eta}\right|_{r_{i1}} + \Phi_{1,1}(t_{ij})\left.\frac{\partial r}{\partial \eta}\right|_{r_{iM}} \tag{26.33}$$

and the respective 2-D grid can be evaluated as

$$r_{ij} = T_I + T_J - T_I T_J$$

where

$$\begin{aligned} T_I T_J[r(s_{ij}, t_{ij})] = (1 - s_{ij}) &\left[\Phi_{0,0}(t_{ij})\gamma_{11} + \Phi_{1,0}(t_{ij})\gamma_{1M} + \Phi_{0,1}(t_{ij})\left.\frac{\partial r}{\partial \eta}\right|_{r_{11}} + \Phi_{1,1}(t_{ij})\left.\frac{\partial r}{\partial \eta}\right|_{1M}\right] \\ + &\left[s_{ij}\ \Phi_{0,0}(t_{ij})r_{N1} + \Phi_{1,0}(t_{ij})r_{NM} + \Phi_{0,0}(t_{ij})\left.\frac{\partial r}{\partial \eta}\right|_{r_{NM}} + \Phi_{1,1}(t_{ij})\left.\frac{\partial r}{\partial \eta}\right|_{NM}\right] \end{aligned} \tag{26.34}$$

An important factor in applying the Hermite interpolation projectors in TFI formulation is the evaluation of slopes and twist vectors (cross derivatives). The slope vector $\overline{\mathbf{r}}_\xi$ can be evaluated by solving

$$\begin{aligned} \mathbf{r}_\xi \cdot \mathbf{r}_\eta &= (\sqrt{g_{11}g_{22}})\cos(\theta_1) \\ \|\mathbf{r}_\xi \times \mathbf{r}_\zeta\| &= A \end{aligned} \tag{26.35}$$

where θ_1 is the desired angle between grid lines ξ and η, and A is the desired area of the cell. The metric terms g_{11} and g_{22} can be evaluated using the desired change in arc length or from the appropriate algebraic grid (precise spacing control property of the algebraic grid can be exploited here). The system (26.35) can be uniquely solved to evaluate $\mathbf{r}_\xi$ and $\mathbf{r}_\eta$, and $\mathbf{r}_\varsigma$ can be evaluated similarly. The twist vectors $\mathbf{r}_{\xi\eta}$ and $\mathbf{r}_{\xi\xi}$ can be evaluated by solving

$$\begin{aligned} (\mathbf{r}_\xi \cdot \mathbf{r}_\eta)_\eta &= [(\sqrt{g_{11}g_{22}})\cos(\theta_1)]_\eta \\ \|\mathbf{r}_\xi \times \mathbf{r}_\eta\|_\eta &= (A)_\eta \end{aligned} \tag{26.36}$$

and

$$\begin{aligned} (\mathbf{r}_\xi \cdot \mathbf{r}_\eta)_\xi &= [(\sqrt{g_{11}g_{22}})\cos(\theta_1)]_\xi \\ \|\mathbf{r}_\xi \times \mathbf{r}_\xi\|_\eta &= (A)_\xi \end{aligned} \tag{26.37}$$

respectively. The other cross derivatives can be evaluated similarly. Observe that if orthogonality is desired, that is, $\theta_1 = 90^\circ$, then the right-hand side will have zeros except for A, A_ξ, and A_η where A_ξ and A_η represent the change in desired volumes in all areas in the ξ and η directions. This concept can be easily extended to three-dimensional configurations.

26.3.1.2 Elliptic Grid Generation

A multitude of general purpose elliptic generation systems here appeared [Thompson 1987b]. Most of these algorithms are based on an iterative adjustment of control functions to achieve boundary orthogonality. The following analysis is provided to illustrate this development.

Consider

$$\begin{aligned}(g_{ij})_{\xi^k} &\equiv (\text{derivative of } g_{ij} \text{ with respect } \xi^k)\\ &\equiv \mathbf{r}_{\xi^i\xi^k}\cdot\mathbf{r}_{\xi^j}+\mathbf{r}_{\xi^i}\cdot\mathbf{r}_{\xi^j\xi^k}\\ i=1,2,3;\qquad j=1,2,3;\qquad &\text{and}\qquad k=1,2,3\end{aligned}\tag{26.38}$$

Using Equation 26.12, the following statement can be obtained:

$$\mathbf{r}_{\xi^i\xi^j}\cdot\mathbf{r}_{\xi^k}=\frac{(g_{ik})_{\xi^j}-(g_{ij})_{\xi^k}+(g_{jk})_{\xi^i}}{2}$$
$$i=1,2,3;\qquad j=1,2,3;\qquad k=1,2,3$$

The three-dimensional elliptic grid generation system presented in Eq. (26.12) can be rewritten by taking the dot product with r_{ξ^q}, $q=1,2,3$ as

$$\sum_{i=1}^{3}\sum_{j=1}^{3}g^{ij}\mathbf{r}_{\xi^i\xi^j}\cdot\mathbf{r}_{\xi^q}+\sum_{k=1}^{3}\Phi_k g^{kk}\mathbf{r}_{\xi^k}\cdot\mathbf{r}_{\xi^q}=0\tag{26.39}$$

This can be written in terms of metric terms and their derivatives as

$$\sum_{i=1}^{3}\sum_{j=1}^{3}g^{ij}\frac{((g_{iq})_{\xi^j}-(g_{ij})_{\xi^q}+(g_{jq})_{\xi^i})}{2}+\sum_{k=1}^{3}\Phi_k g^{kk}g_{kq}=0\quad q=1,2,3\tag{26.40}$$

Now

$$g_{ii}=\overline{\mathbf{r}}_{\xi^i}\cdot\overline{\mathbf{r}}_{\xi^i}=\|\overline{\mathbf{r}}_{\xi^i}\|^2$$

represents an increment of arc length on a coordinate line along which ξ^i varies and

$$g_{ij}=\overline{\mathbf{r}}_{\xi^i}\cdot\overline{\mathbf{r}}_{\xi^j}=|\overline{\mathbf{r}}_{\xi^i}|\cdot|\overline{\mathbf{r}}_{\xi^j}|\cdot\cos\theta\quad i\neq j$$

represents a measure of orthogonality between grid lines along which ξ^i and ξ^j varies. These quantities can be evaluated if the desired increment in the arc length and desired angles between grid lines are known. Looking at the precise control of spacing property of the algebraic grid [Soni 1992a, b], the quantities g_{ij} can be evaluated from the well-defined algebraic grid, and using

$$g_{ij}=(\sqrt{g_{ii}})(\sqrt{g_{jj}})\cos\theta\tag{26.41}$$

where θ is the desired angle between ξ^i,ξ^j grid lines, the quantities g_{ij}, $i\neq j$, can be evaluated. Once all g_{ij} are known, then Eq. (26.40) can be solved for the forcing functions Φ_k, $k=1,2,3$.

If orthogonality is enforced, i.e., $\theta=90^\circ$ or $g_{ij}=0$ for $i\neq j$, then Φ_k, $k=1,2,3$, can be formulated as

$$\hat{\Phi}_k=\frac{1}{2}\frac{\mathrm{d}}{\mathrm{d}\xi^k}\left(\ell n\frac{g_{kk}}{g_{ii}g_{jj}}\right),\qquad (i,j,k)\text{ cyclic}\quad k=1,2,3\tag{26.42}$$

and

$$\Phi_k=-\frac{(g_{ii})(g_{jj})\hat{\Phi}_k}{g},\qquad (i,j,k)\text{ cyclic}\quad k=1,2,3\tag{26.43}$$

A usual practice is to utilize Equation 26.43 in the following form:

$$\Phi_k = -\frac{\mathbf{r}_{\xi^k} \cdot \mathbf{r}_{\xi^k\xi^k}}{|\mathbf{r}_{\xi^k}|^2} \frac{\mathbf{r}_{\xi^k} \cdot \mathbf{r}_{\xi^i\xi^i}}{|\mathbf{r}_{\xi^i}|^2} \frac{\mathbf{r}_{\xi^k} \cdot \mathbf{r}_{\xi^j\xi^j}}{|\mathbf{r}_{\xi^j}|^2} \tag{26.44}$$

In fact, the firm term of the definition of Φ_k provides the distribution control and the remaining two terms contribute toward the curvature control.

To understand the iterative evaluation of the control function, consider a two-dimensional elliptic system:

$$g_{22}(\mathbf{r}_{\xi\xi} - \phi\mathbf{r}_\xi) - 2g_{12}\mathbf{r}_{\xi\eta} + g_{11}(\mathbf{r}_{\eta\eta} - \psi\mathbf{r}_\eta) = 0 \tag{26.45}$$

The control functions ϕ and ψ can be formulated as

$$\Phi = -\frac{\mathbf{r}_\xi \cdot \mathbf{r}_{\xi\xi}}{|\mathbf{r}_\xi|^2} - \frac{\mathbf{r}_\xi \cdot \mathbf{r}_{\eta\eta}}{|\mathbf{r}_\eta|^2} \tag{26.46}$$

and

$$\Psi = -\frac{\mathbf{r}_\eta \cdot \mathbf{r}_{\eta\eta}}{|\mathbf{r}_\eta|^2} - \frac{\mathbf{r}_\eta \cdot \mathbf{r}_{\xi\xi}}{|\mathbf{r}_\xi|^2} \tag{26.47}$$

During the evaluation of ϕ: (1) the quantities $\mathbf{r}_\xi$ and $\mathbf{r}_{\xi\xi}$ can be evaluated by utilizing appropriate finite-difference approximations, (2) the $\mathbf{r}_\eta$ are evaluated by solving

$$\mathbf{r}_\xi \cdot \mathbf{r}_\eta = 0 \qquad \text{and} \qquad \|\mathbf{r}_\xi \times \mathbf{r}_\eta\| = (\Delta A) \tag{26.48}$$

where (ΔA) is the desired cell area, and (3) the $\mathbf{r}_{\eta\eta}$ quantities are calculated using the finite-difference approximation on the current grid. These quantities are updated at every iteration. Another approach is to utilize well-distributed algebraic grid characteristics to solve the following equations in order to evaluate $\mathbf{r}_{\eta\eta}$:

$$(\mathbf{r}_\xi \cdot \mathbf{r}_\eta)_\xi = 0 \qquad \text{and} \qquad \|\mathbf{r}_\xi \times \mathbf{r}_\eta\|_\xi = (\Delta A)_\xi \tag{26.49}$$

and

$$(\mathbf{r}_\xi \cdot \mathbf{r}_\eta)_\eta = 0 \qquad \text{and} \qquad \|\mathbf{r}_\xi \times \mathbf{r}_\eta\|_\eta = (\Delta A)_\eta \tag{26.50}$$

where $(\Delta A)_\xi$ and $(\Delta A)_\eta$ represent the change of cell area in the ξ and η directions, respectively (they can be computed using finite-difference approximation from a well-distributed algebraic grid or by utilizing desired cell areas on the boundaries). The control functions are usually evaluated on the boundaries and then interpolated in the interior. The distribution mesh can be utilized as a parametric space for doing this interpolation.

26.3.2 The Delaunay Algorithm

The algorithm widely utilized to generate the Delaunay triangulation is based upon the work of Bowyer. The algorithm is based on the in-circle criterion, and is a sequential process with each point introduced into an existing Delaunay satisfying structure, which is broken and then reconnected to form a new Delaunay triangulation [Baker 1990, George and Hermeline 1992]. The algorithm is applicable in two- and three-dimensions and in step-by-step format as follows:

Algorithm I.

1. *Define a set of points which forms a convex hull within which all points will lie.*
2. *Introduce a new point anywhere within the convex hull.*
3. *Determine all vertices of the Voronoi diagram to be deleted. A point which lies within a circle (sphere), centered at a vertex of the Voronoi diagram and which passes through its three (four) forming points, results in the deletion of that vertex. This follows from the in-circle criterion of the Voronoi construction.*

4. *Find the forming points of all the deleted Voronoi vertices. These are the contiguous points to the new point.*
5. *Determine the neighboring Voronoi vertices to the deleted vertices which have not themselves been deleted.*
6. *Determine the forming points of the new Voronoi vertices. The forming points of new vertices must include the new point together with two (three) points which are contiguous to the new point and form an edge (face) of a neighboring Voronoi diagram data structure, overwriting the entries of the deleted vertices.*
7. *Repeat steps (2–6) for the next point.*

In the preceding algorithm, the interpretation for three dimensions is included in parentheses. This algorithm has been used for the construction of the triangulation in two and three dimensions. It does not differ in content from that used in earlier work, but its implementation has made use of highly efficient search procedures and, hence, the computational time is considerably less than that used in earlier work.

For grid generation purposes the boundary of the domain is defined by points and associated connectivities. It will be assumed that the grid points on the boundary reflect appropriate variations in geometrical slope and curvature. Ideally any method which automatically creates points should ensure that the boundary point distribution is extended into the domain in a spatially smooth manner. An algorithm which achieves this in both two and three dimensions is the following:

Algorithm II.

1. *Compute the point distribution function for each boundary point* $r_0 = (x, y, z)$ *(i.e., for point* 0*)*

$$dp_0 = \frac{1}{M} \sum_{i=1}^{M} |r_1 - r_0|$$

where $\|$ *is the Euclidean distance and it is assumed that point* 0 *is surrounded by M points,* $i = 1, M$.
2. *Generate the Delaunay triangulation of the boundary points.*
3. *Initialize the number of interior field points created,* $N = 0$.
4. *For all tetrahedra within the domain:*
   a. *Define a prospective point Q to be at the centroid of the tetrahedron.*
   b. *Derive the point distribution dp for the point Q, by interpolating the point distribution function from the nodes of the tetrahedron,* dp_m, $m = 1, \ldots, 4$.
   c. *Compute the distances* d_m, $m = 1, \ldots, 4$, *from the prospective point, Q, to each of the four points of the tetrahedron.*
      If $\{d_m < a dp_m\}$ *for any* $m = 1, \ldots, 4$ *then*
      reject the point :- Return to the beginning of step 4.
      If $\{d_m > a dp_m\}$ *for any* $m = 1, \ldots, 4$ *then*
      compute the distance s_j, $(j = 1, \ldots, N)$, *from the prospective point Q, to other points to be inserted,* P_j, $j = 1, N$.
      If $\{s_j < \beta dp_m\}$ *then*
      reject the point :- Return to the beginning of step 4.
      If $\{s_j > \beta dp_m\}$ *then*
      accept the point Q for insertion by the Delaunay triangulation algorithm and include Q in the list P_j, $j = 1, N$.
   d. *Assign the interpolated value of the point distribution function dp to the new node* P_N.
   e. *Next tetrahedra.*
5. *If* $N = 0$ *go to step 7.*
6. *Perform the Delaunay triangulation of the derived points,* P_j, $j = 1, N$. *Go to step 3.*
7. *Smooth the mesh.*

In the preceding algorithm, the term tetrahedron and triangle are interchangeable. The coefficient α controls the grid point density by changing the allowable shape of formed tetrahedra, whereas β has an influence on the regularity of the triangulation by not allowing points within a specified distance of each other to be inserted in the same sweep of the tetrahedra within the field. The effects of the parameters α and β are demonstrated in the following examples, which for convenience are presented for domains in two dimensions.

The interpolation of the boundary point distribution function is linear throughout the field. This can be modified to provide a weighting toward the boundaries so as to ensure greater point density in such regions. The implementation of such a procedure involves a scaling of the point distribution of the nodes, which form an element on the boundary. It should be noted that this point creation algorithm can be implemented very efficiently within the Delaunay triangulation procedure. In particular, if a point is accepted for insertion, then in the Delaunay algorithm a tetrahedron is known which contains this point, since by the very nature of the procedure the tetrahedron from which the point was created is known. However, after the insertion of one point the tetrahedron numbering can be changed, and if the tetrahedra formed from the inserted points overlap, then the tetrahedron numbers which have been flagged for each new point can be then incorrect. However, the exclusion zone, controlled by the parameter β, ensures that the points created from one sweep through the tetrahedra are sufficiently spatially separated that on the insertion of each point the resulting tetrahedra do not overlap and, hence, the original tetrahedron numbers associated with each new point are valid. Hence, in this way β improves the regularity of the tetrahedra and also ensures that no search is required to find a circle which includes the point.

The procedure outlined creates points consistent with the point distribution on the boundaries. Simple modifications provide greater flexibility.

26.3.2.1 Point Creation by the Use of Sources

In somewhat of an analogous way to point sources used as control functions with elliptic partial differential equations, it is possible to define line and point sources to provide grid control for unstructured meshes. Local point spacing, at position r, can be defined as

$$dp(r) = A_j e^{B_j|R_j - r|}$$

where A_j and B_j are the user specified amplification and decay parameters of the sources j, $j = 1, M$, and R_j is the position of each point source. Grid point creation is then performed as outlined in Algorithm II but in step 4b the appropriate point distribution function at the centroid is determined by Equation 26.2. Various forms of implementation of this can be devised. One simple modification is to define the point spacing as

$$dp(r) = \min\left(dp_{\text{boundary}}, A_j e^{B_j|R_j - r|}\right)$$

In this case dp_{boundary} is the point distribution from the boundary spacing. This then provides the desired point clustering in regions influenced by the sources but farther away the boundary point distribution has a dominant effect.

26.3.2.2 Point Creation Controlled by a Background Mesh

Another way to control the point spacing in the domain is to use a background mesh. A mesh is overlaid over the domain and at each node a point spacing is specified. To encompass this approach within the framework of Algorithm II, the point distribution function dp for a prospective point is obtained from the interpolated spacing from the background mesh. Within Algorithm II step 4b is replaced by the interpolation of dp from the background mesh. The effect is similar to that achieved by sources.

26.3.2.3 Boundary Integrity

This problem is widely recognized as a problem inherent to the generation of boundary conforming grids using the Delaunay triangulation. Several approaches to its solution have been proposed. The procedure

followed here is an extension of earlier work and is closely related to the work of George. Both these approaches involve the addition of points to *block* the penetration of tetrahedra through the boundary surface. In the former approach, points were added on the boundary, which were then connected to the surrounding points using the Delaunay triangulation procedure. In two dimensions and in some cases in three dimensions, this approach proves to be adequate. Hence, the approach used in two dimensions uses this technique. However, in three dimensions, for some severe cases, it proves to be difficult to completely ensure the reconstruction of surface triangles. Hence, a different approach has been devised for three dimensions, which also involves the addition of points, but these are connected directly to the tetrahedral construction rather than using the Delaunay triangulation. A finite number of direct connections can be formulated for all types of tetrahedral penetration and hence in the proposed procedure the recovery of the surface can be guaranteed.

The necessary and sufficient conditions for a face with nodes (P, Q, R) to be present in the triangulation $\{\tau_l\}, l = 1, T$ are as follows:

1. The nodes P, Q, and R exist in the tetrahedral construction $\{\tau_l\}, l = 1, T$.
2. The cyclic combinations of P, Q, and R that is, *PQ, QR, RP* occur in one of the tetrahedra in the construction $\{\tau_l\}, l = 1, T$.
3. The combination (P, Q, R) exists in one of the tetrahedra in $\{\tau_l\}$, $l = 1, T$.

Hence, to recover an arbitrary set of triangular faces, these three conditions must be met for each face. The first condition appears to be self-evident.

If a boundary face is not in τ, this is because edges and faces of the tetrahedra $\{\tau\}$ intersect the required face. Since a face is formed from edges, and it is assumed that the points P, Q and R are present, it is necessary to firstly recover edges *PQ, QR, RP* and then the face (P, Q, R). This is achieved, for a given face *PQR,* by firstly finding the tetrahedra which are intersected by the edges *PQ, QR,* and *RP*. These tetrahedra are then modified and new tetrahedra created so that the required edges are present. Once edges are present, a similar procedure follows to recover the face. If the edges *PQ, QR,* and *RP* exist but the face (P, Q, R) does not exist, then all tetrahedra which possess at least one edge which intersects the face (P, Q, R) are determined. These tetrahedra are then modified accordingly to recover the missing faces.

26.3.2.4 Edge Swapping

In circumstances where it is required to have a complete surface, although there is not a fixed constraint that a given set of faces is recovered, it is possible to ensure boundary faces are coincident with faces within the tetrahedral construction by swapping edges within the boundary surface triangulation. If in the tetrahedral construction faces *ABC* and *BCD* exist, but in the surface triangulation faces *ACD* and *ABD* exist, then the two can be made to agree if in the surface triangulation edge *AD* is replaced by *BC*. There are conditions under which such a transformation is not allowed and these must be checked.

Edge swapping can be incorporated as an option. However, if it can be used, it can greatly reduce the amount of work to be carried out in the edge and face recovery routines.

26.3.2.5 Boundary Edge Recovery

The procedure to recover a missing edge of a boundary face involves two steps. First, it is necessary to identify the faces, edges, and points of tetrahedra which the edge intersects. Second, local transformations involving tetrahedra are performed to recover the edge. The intersections of edges with tetrahedra can be readily computed. Once this has been performed, a set of transformations are used to recover edges.

26.3.2.6 Boundary Face Recovery

After the recovery of all boundary face edges, the boundary faces can then be recovered. Clearly, although the edges of all boundary faces are present, this does not imply that boundary faces are present, since other

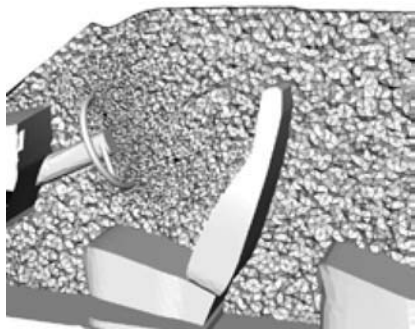


FIGURE 26.9 Volume grid of Ford Explorer.

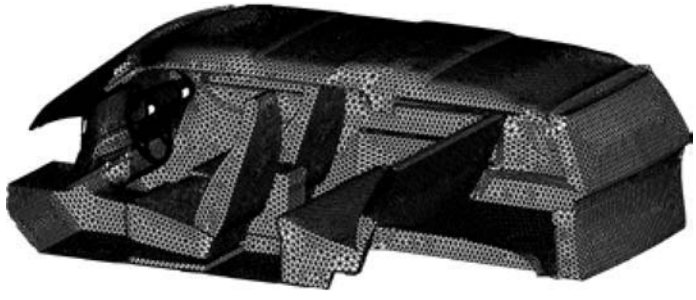


FIGURE 26.10 Surface grid of Ford Explorer interior.

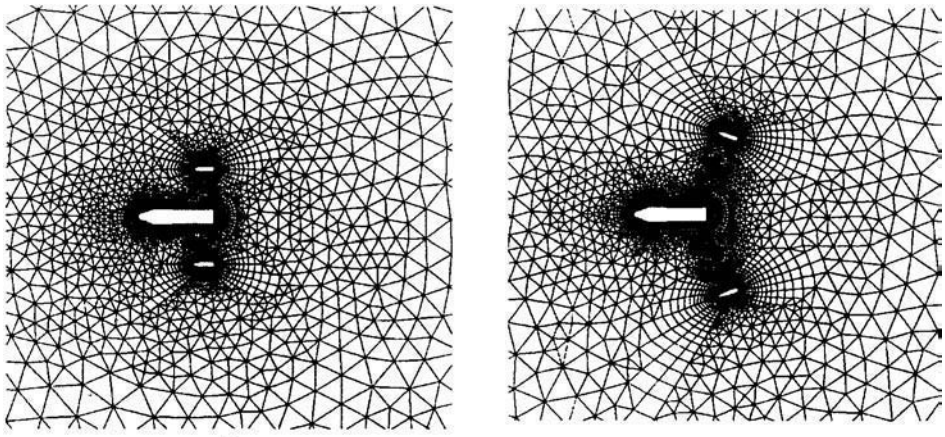
tetrahedra can penetrate the interior of a face but not the boundary face edges. If a boundary face is not present, it is necessary to determine all tetrahedra which intersect the face. One, two, three, or four edges and associated segments of a tetrahedra can intersect a face. Hence, for each missing face, all tetrahedra which have an edge or edges which intersect the face are determined and each of the tetrahedra are then classified accordingly.

26.3.2.7 Removal of Added Points

Most of the transformations used to recover the edges and faces in both 2-D and 3-D grids involve the creation of one or more points. These added surface points are used purely as part of the boundary recovery procedure and are removed after the boundary is complete. The mechanics of node removal involve taking each added point in turn and finding all elements connected to it. These elements are deleted leaving an empty polyhedron, which is then triangulated in a direct manner by finding point connections which lead to the optimum-shaped tetrahedral construction. This is a rapid process since this operation is performed locally for a relatively small number of points. A pictorial view of an unstructured grid for automotive application is presented in Figure 26.9 and Figure 26.10. The grid is generated by advancing front local reconnection algorithm [Marcum 1995].

26.3.3 Hybrid Grid Generation

A hybrid grid consists of structured grid in part of the physical domain and unstructured grid in rest of the field. In general, a hybrid grid can be defined as an agglomeration of cells having polygons with a different number of sides. This necessitates a hybrid grid generation process as a combination of structured and unstructured grid methodologies. The truncation error of a triangular cell is inversely proportional to



(a) (b)

FIGURE 26.11 Hybrid grid around a launch vehicle with boosters: (a) grid at an initial state and (b) grid after booster separation.

the sine of the minimum angle of the triangle. Therefore, to reduce the truncation error the number of triangles in the boundary layer has to increase and, consequently, it will increase the total number of cells in the field. This will lead to more memory and central processing unit (CPU) time for the flow solver. The structured grid in the boundary layer will help to overcome these difficulties.

The basic steps involved in hybrid grid generation are the following. The first step is to decompose the complex geometries into simple geometric entities. A structured grid is generated around these geometric entities using an advancing layer-type method based on the surface normals together with the application of a local elliptic solver [Koomullil et al. 1996a]. The second step involves the trimming of the overlapping structured grid from different solid bodies in the domain, by comparing the aspect ratio. Cells having aspect ratio less than unity are removed. In the third step the void in the physical domain after the trimming of the structured grid is filled with unstructured grid.

The hybrid grid for dynamic motion-type geometries can be generated quickly and efficiently using the approach present in Koomullil et al. [1996a]. An example of a hybrid grid for the moving geometry problem is illustrated in Figure 26.11. The strap-on separation from the main launch vehicle booster is shown in this figure. The relative position of the strap-ons and booster rockets at different times and the hybrid grid around this configuration is shown in Figures 26.11a and b.

26.3.3.1 Grid Adaption: Construction of Weight Functions

Application of the equidistribution law results in grid spacing inversely proportional to the weight function, and hence, the weight function determines the grid point distribution. Ideally, the weight would be the local truncation error ensuring a uniform distribution of error. Determination of this function is one of the most challenging areas of adaptive grid generation. The overall solution is only as accurate as the least accurate region. Thus, excessive resolution in certain regions does not increase the accuracy of the overall solution.

Evaluation of higher order derivatives from discrete data is progressively less accurate and subject to noise. However, lower order derivatives must be nonzero in regions of wide variations of higher order derivatives, and are proportional to the rate of variation. Therefore, it is possible to employ lower order derivatives as a proxy for the truncation error.

Analysis of the weight functions explored to date indicates that density or velocity derivatives are not independently sufficient to represent the different types and strengths of flow features. Density, or pressure for that manner, varies insufficiently in the boundary layer to be used to construct weight functions for representation of these features. Whereas velocity derivatives for viscous flows by themselves are dominated by the boundary layer, additional variables must be included to represent other flow features. The weight

function [Thornburg et al. 1996] consists of relative derivatives of density and the three conservative velocities,

$$W_{ijk}^{k} = 1.0 + \frac{|\hat{q}_{ijk}|}{|\hat{q}_{ijk} + e|_{\max}} + \frac{|(\hat{q}_{\xi^k})_{ijk}/\hat{q}_{ijk}|}{|(\hat{q}_{\xi^k})_{ijk}/\hat{q}_{ijk} + e|_{\max}} + \frac{|(\hat{q}_{\xi^k\xi^k})_{ijk}/\hat{q}_{ijk}|}{|(\hat{q}_{\xi^k\xi^k})_{ijk}/\hat{q}_{ijk} + e|_{\max}}$$

where $\hat{q} = (\varrho, \varrho u, \varrho v, \varrho w)$. The relative derivatives are necessary to detect features of varying intensity, so that weaker, but important structures such as vortices are accurately reflected in the weight function. One-sided differences are used at boundaries, and no-slip boundaries require special treatment since the velocity is zero. This case is handled in the same manner as zero velocity regions in the field. A small value, epsilon in the preceding equation, is added to all normalizing quantities. Also it appears that the Boolean sum construction method of Thornburg et al. [1998] would balance the weight functions more evenly, as several features are reflected in multiple variables, whereas some are reflected in only one.

26.4 Grid Systems

A multitude of general purpose grid generation codes to address complex three-dimensional structured–unstructured grid generation needs are newly available in the public domain or as proprietary commercial code. A brief description of the widely utilized candidate general-purpose codes follows.

The National Grid Project (NGP) system of Mississippi State University is an interactive geometry and grid generation system for block-structured and tetrahedral grids. The system reads CAD data via IGES and converts all surface patches to NURBS. A carpet, composed of interfacing NURBS patches, is then laid over the CAD patches to correct for gaps and overlaps. The system also has internal CAD capability for the construction or repair of surfaces. Surface grids are generated on the NURBS carpet, and can be projected onto the original CAD patches. Both the surface grids and the subsequent volume grid can be generated as block structured via elliptic, hyperbolic, or TFI methods, or as unstructured via Delaunay or advancing front procedure. A pictorial view demonstrating the graphical interface is displayed in Figure 26.12.

The ICEM–CFD [Akdag and Wulf 1992] system is a commercial code which offers block-structured grids, tetrahedral grids, and unstructured hexahedral grids. The system interfaces with numerous CAD systems and has been connected to a number of flow solvers.

The GRIDGEN [Steinbrenner and Chawner 1993] system is a graphically interactive block-structured commercial code. The user constructs curves, which are in turn used to build the topological surface and

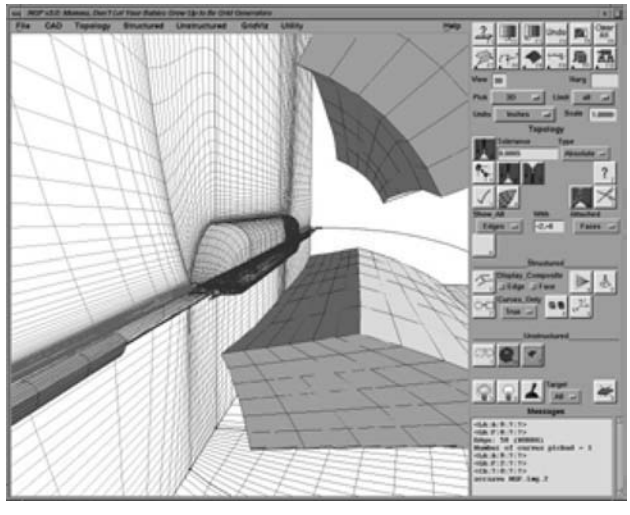


FIGURE 26.12 NGP code interface demonstrating multiblock grid.

volume components. The user then selects curves as the boundaries of surface grids, and finally surfaces as the boundaries of volume grids (blocks). With this system, grid generation is a user-in-the-loop task.

EAGLEView [Soni et al. 1992] is a graphical system that allows interactive construction of geometry and block-structured grids with journaling capability.

GridPro/az3000 [Eiseman 1995] is a commercial block-structured code topology input language (TIL) to define both the surface and the block-structured grid. The language includes components (objects) that can be invoked, and therefore admits the formation of element libraries.

CFD-GEOM [Hufford et al. 1995] is an interactive geometric modeling and grid generation system for block structured grids, tetrahedral (advancing front) grids, and hybrid grids. All elements are linked so that updates are propagated throughout the database. The geometry is NURBS based, reads IGES files, and has some internal CAD capabilities. The system also has macrolibrary capability.

The GEMS [Dener et al. 1994] block-structured grid generation system of SAMTEK-ITC in Turkey is based on object-oriented programming and C++ that uses case-based reasoning and reinforcement learning to capture CFD expertise. The system selects the case that is best suited for a particular geometry from among known ones.

The 3-DGRAPE/AL [Sorenson and Alta 1995] system of NASA Ames is a block-structured grid generator that now includes the specification of arbitrary intersection angles at boundary surfaces, as well as the orthogonality pioneered by Steger–Sorenson.

The GENIE++ [Soni et al. 1992] block-structured grid generation system of Mississippi State was also introduced in the late 1980s and has been continually enhanced over the years. This system uses TFI with elliptic smoothing and includes various splining methods.

VGRID [Parikh and Pirzadeh 1992] of NASA Langley is a tetrahedral grid generator which uses advancing front with a Cartesian background grid to control resolution.

TGrid of Fluent is a tetrahedral grid generator based on the Delaunay approach.

The first general-purpose-domain connectivity codes for chimera grids were the PEGSUS (from the Air Force Arnold Engineering Development Center) and CMPGRD (from IBM) codes in the late 1980s [Meakin 1995], which continue to be enhanced. Advances in CMPGRD are detailed in Henshaw et al. [1992]. Later codes are DCF3D of NASA Ames and Overset Methods (MEAKIN 1991) and BEGGAR of the Air Force Wright Laboratory at Eglin [Belk 1995, Maple and Belk 1994]. A detailed description of these codes and a comprehensive review of existing codes and technology can be found in Thompson [1996].

26.5 Research Issues and Summary

The first step of the CFS process is the construction of a discrete approximation of the general region of interest. This representation could be multiblock structured, unstructured, or hybrid. Only for the simplest of applications can a grid be generated quickly or easily. In fact, geometry-grid generation is by far the most time consuming aspect of the entire CFS process. Rapid turnaround, reliable accuracy, and affordability are the three key requirements to be addressed for CFS to play its rightful role in supporting multidisciplinary design environment. To this end, industry is targeting grid generation in 1 h for complex configuration, i.e., reliable 1-h grid generation turnaround for one-time geometries when run by designers. The system must include CAD-to-grid links, which resolve tolerance issues and produce grids with a quality good enough for the CFS solver. The designer has to feel that the grid generation process is under control and is predictable. The following critical barriers must be overcome to fulfill the aforestated industrial requirements.

The time-consuming aspects of grid generation are usually related to the geometry definition, i.e., the input of the geometry information into the grid system. Today, trimming the surfaces associated with their intersection is a significant barrier. The trimmed surfaces are a widely utilized entity in the construction of complex geometrical configurations in CAD/CAM systems. Algorithms which utilize surface triangulation techniques and solid modeling schemes need to be developed. Reliable methodologies for representing triangulated surfaces into a single surface represented by the NURBS are needed. The

CAD/CAM technology and methodology evolution based on solid modeling will also reduce this present barrier of addressing geometries undergoing design perturbations.

Automatic (noninteractive) algorithms for domain decomposition for the development of multiblock structured grids pose a barrier for addressing multidisciplinary design applications involving geometry optimization. The solution grid adaptive algorithms, at present, are limited to simple three-dimensional configurations. Techniques are needed to enhance the applicability of adaptive schemes pertaining to complex configurations. Parallel and distributed processing of grid generation algorithms is also essential for these multidisciplinary applications. Algorithms need to be developed to improve the quality of unstructured surface grids since they highly influence the quality of unstructured volume grids. Hybrid-generalized grid techniques are promising, especially for multidisciplinary CFS applications that include dynamic motion. This technology does not exist for full three-dimensional configurations.

References

Akdag, V. and Wulf, A. 1992. Integrated geometry and grid generation system for complex configurations, p. 161. In *Proc. Software Syst. Surface Modeling Grid Generation Workshop*. R. E. Smith, ed. NASA Conf. Pub. 3143, NASA Langley Research Center, Hampton, VA.

Baker, T. J. 1990. Unstructured mesh generation by a generalized Delaunay algorithm, pp. 20.1–20.10. In *Appl. Mesh Generation to Complex 3-D Configurations*. AGARD Conf. Proc. No. 464.

Belk, D. M. 1995. The role of overset grids in the development of the general purpose CFD code, p. 193. In *Proc. Surface Modeling, Grid Generation Related Issues Comput. Fluid Dyn. Workshop*, NASA Conf. Publ. 3291, NASA Lewis Research Center, Cleveland, OH, May.

Benek, J. A., Buning, P. G., and Steger, J. L. 1985. A 3-D chimera grid embedding technique. AIAA Paper 85–1523.

Blake, M. W. and Chou, J. J. 1992. The NASA-IGES geometry data exchange standard. *Proc. Workshop Sponsored by NASA*. Washington, DC, Langley Research Center, Hampton, VA, April.

Brackbill, J. U. 1993. An adaptive grid with directional control. *J. Comput. Physics*. 108:38.

Cavendish, J. C., Field, D. A., and Frey, W. H. 1985. An approach to automatic three dimensional finite element mesh generation. *Int. J. Num. Methods Eng.* 21:329–348.

Dener, C., Koc, E., and Sirin, I. 1994. Extentions to GEMS for automatic grid generation and intelligent topology definition. *Numerical Grid Generation in Computational Field Simulation and Related Fields*. N. P. Weatherill, P. R. Eiseman, J. Hauser, and J. F. Thompson, Eds., p. 453. Proc. 4th Int. Grid Con. Pineridge Press, Ltd. Swansea, Wales, UK.

Dirichlet, G. L. 1850. Uber die Reduction der positiven quadratischen formen mit drei understimmten, ganzen Zahlen. *J. Reine Angew. Math.* 40(3):209–227.

Eiseman, P. R. 1985. Grid generation for fluid mechanics computations. *Annu. Rev. Fluid Mech.* 17:487–522.

Eiseman, P. R. 1995. Multiblock grid generation with automatic zoning, p. 143. In *Proc. Surface Modeling, Grid Generation Related Issues Comput. Fluid Dyn. Workshop*, NASA Conf. Pub. 3291, NASA Lewis Research Center, Cleveland, OH, May.

Gatlin, B., Thompson, J. F., Yoon, Y.-H., Luong, P. V., Ganapathiraju, D., and Wolverton, M. K. 1991. Extensions to the EAGLE grid code for quality control and efficiency. *29th AIAA Aerospace Sci. Meeting*, AIAA Paper 91-0148, Reno, NV, Jan.

George, A. J. 1971. *Computer Implementation of the Finite Element Method*. Ph.D. Thesis, Stanford University, STAN-CS-71-208.

George, P. L. and Hermeline, F. 1992. Delaunay's mesh of a conven polyhedron in dimension D; application for arbitrary polyhedra. *Int. J. Num. Methods Eng.* 33:975–995.

Gordon, W. J. and Thiel, L. C. 1982. Transfinite mappings and their application to grid generation. In *Numerical Grid Generation*. J. F. Thompson, Ed. North Holland, Amsterdam.

Henshaw, W. D., Chessire, G., and Henderson, M. E. 1992. On constructing three-dimensional overlapping grids with CMPGRD, p. 415. In *Proc. Software Systems Surface Modeling Grid Generation Workshop*, R. E. Smith, Ed. NASA Conf. Pub. 3143, NASA Langley Research Center, Hampton, VA.

Hufford, G. S., Harrand, V. J., Patel, B. C., and Mitchell, C. R. 1995. Evaluation of grid generation technologies from an applied perspective, p. 401. In *Proc. Surface Modeling, Grid Generation Related Issues Comput. Fluid Dyn. Workshop*, NASA Conf. Pub. 3291, NASA Lewis Research Center, Cleveland, OH, May.

Koomullil, R. P., Soni, B. K., and Huang, C. 1996a. Flow simulations on generalized grids, pp. 527–536. In *5th Int. Conf. Num. Grid Generation Comput. Fluid Dyn. Related Fields.* B. K. Soni, J. F. Thompson, J. Hauser and P. Eiseman, Eds. Mississippi State University, April 1–5.

Koomullil, R. P., Soni, B. K., and Huang, C. 1996b. Navier–Stokes Simulation on Hybrid Grids. *34th Aerospace Sci. Meeting,* AIAA Paper 96-767, Reno, NV, Jan. 15–18.

Lo, S. H. 1985. A new mesh generation scheme for arbitrary planar domains. *Int. J. Num. Methods in Eng.* 21:1403–1426.

Lohner, R. and Parikh, P. 1988. Three-dimensional grid generation by the advancing-front method. *Int. J. Num. Methods Fluids* 8:1135–1149.

Maple, R. C. and Belk, D. M. 1994. Automated setup of blocked, patched and embedded grids in the beggar flow solver. In *Numerical Grid Generation in Computational Field Simulations and Related Fields*, N. P. Weatherill, P. R. Eiseman, J. Hauser, and J. F. Thompson, Eds., p. 151. Proc. 4th Int. Grid Conf. Pineridge Press Limited. Swansea, Wales, UK.

Marcum, D. L. 1995. Generation of unstructured grids for viscous flow applications. *33rd Aerospace Sci. Meeting and Exhibit,* AIAA Paper 95-0212, Reno, NV, Jan. 9–12.

Meakin, R. L. 1991. A new method for establishing intergrid communication among systems of overset grids. In *10th AIAA Comput. Fluid Dyn. Conf.*, AIAA Paper 91-1586, Honolulu, HI, June.

Meakin, R. L. 1995. Grid related issues for static and dynamic geometry problems using systems of overset structured grids, p. 181. In *Proc. Surface Modeling, Grid Generation Related Issues Comput. Fluid Dyn. Workshop*. NASA Conf. Pub. 3291, NASA Lewis Research Center, Cleveland, OH, May.

Parikh, P. and Pirzadeh, S. 1992. Recent advanced in unstructured grid generation, p. 435. In *Proc. Software Syst. Surface Modeling Grid Generation Workshop*. R. E. Smith, Ed. NASA Conf. Pub. 3143, NASA Langley Research Center, Hampton, VA.

Peraire, J., Morgan, K., and Peiro, J. 1990. Unstructured finite element mesh generation and adaptive procedures for CFD, pp. 18.1–18.12. In *Appl. Mesh Generation Complex 3-D Configurations*, AGARD Conf. Proc. No. 464.

Peraire, J., Peiro, J., Formaggia, L., Morgan, K., and Zeinkiewica, O. C. 1988. Finite element euler computations in three dimensions. *Int. J. Num. Methods Eng.* 26.

Peraire, J., Vahdati, M., Morgan, K., and Zienkiewicz, O. C. 1987. Adaptive remeshing for compressible flow computations. *J. Complex Physics* 72:449–466.

Soni, B. K. 1992a. Grid generation: algebraic and partial differential equations techniques revisited. In *Proc. Comput. Fluid Dyn. '92*, C. Hirsch, J. Periaux, and W. Kordulla, Eds. Vol. 2, pp. 929–936, Sept.

Soni, B. K. 1992b. Grid generation for internal flow configurations. *Comput. Math. Appl.* 24(5/6):191–201.

Soni, B. K., Thompson, J. F., Stokes, M. L., and Shih, M.-H. 1992. GENIE^{++}, EAGLEView and TIGER: general and special purpose graphically interactive grid systems. *30th AIAA Aerospace Sci. Meeting*, AIAA Paper 92-0071, Reno, NV, Jan.

Soni, B. K., Weatherill, N. P., and Thompson, J. F. 1993. Grid adaptive strategies in CFD. In *Advances in Hydro–Science & Engineering*. S. S. Y. Wang, Ed., pp. 1.A:201–208. University of Mississippi Press, Jackson, MS.

Sorenson, R. L. and Alta, S. J. 1995. 3-D GRAPE/AL: the Ames-Langley technology upgrade, p. 447. In *Proc. Surface Modeling, Grid Generation Related Issues Comput. Fluid Dyn. Workshop*. NASA Conf. Pub. 3291, NASA Lewis Research Center, Cleveland, Ohio, May.

Steger, J. L. and Chaussee, D. S. 1980. Generation of body-fitted coordinates using hyperbolic pratial differential equations. *SIAM J. Sci. and Stat. Comput.* 1:431–443.

Steinbrenner, J. P. and Chawner, J. R. 1993. Incorporation of a hierarchical grid component structure into GRIDGEN. *31st AIAA Aerospace Sci. Meeting*. AIAA Paper 93-0429. Reno, NV. Jan.

Thompson, J. F. 1985. A survey of dynamically adaptive grids in numerical solution of partial differential equations. *Appl. Numerical Math.* 1:3–27.

Thompson, J. F. 1987a. A composite grid generation code for general 3-D regions. *25th AIAA Aerospace Sci. Meeting*. AIAA Paper 87-0275. Reno, NV, Jan. 1987.

Thompson, J. F. 1987b. A general three-dimensional elliptic grid generation system on a composite block structure. *Comput. Methods Appl. Mech. Eng.* 64:377–411.

Thompson, J. F. 1993. The national grid project. *Comput. Syst. Eng.* 3(1–4):393–399.

Thompson, J. F. 1996. A reflection on grid generation in the 90's: trends, needs, and influences. In *Int. Num. Grid Generation Comput. Field Simulations*. B. K. Soni, J. F. Thompson, J. Hauser, and P. R. Eiseman, Eds., 1029. Proc. 5th Int. Grid Generation Conf. ERC Press.

Thompson, J. F. and Mastin, C. W. 1983. Order of difference expressions curvilinear coordinate systems. *J. Fluids Eng.* 50:215.

Thompson, J. F., Warsi, Z. U. A., and Mastin, C. W. 1985. *Numerical Grid Generation: Foundations and Applications*. North Holland, Amsterdam.

Thornburg, H., Soni, B., and Boyalakuntla, K. 1998. A structured based solution–adaptive technique for complex separated flows. *Appl. Math. Comput.* 89:199–211.

Voronoi, G. 1908. Nouvelles applications des parametres continus a la theorie des formes quadratiques, Rescherches sur les parallelloedres primitifs. *J. Reine Angew. Math.* 134.

Weatherill, N. P. 1990. Mixed structured-unstructured meshes for aerodynamic flow simulation. *Aeronautical J.* 94(934):111–123.

Yu, T.-U. 1995. *CAGD Techniques in Grid Generation*. Ph.D. dissertation, Computational Engineering Program, Mississippi State University.

Further Information

For complete in-depth literature on grid generation, the reader is referred to Thompson et al. [1985] and five proceedings associated with the 1985, 1988, 1992, 1994, and 1996 International Conferences on Numerical Grid Generation in Computational Fluid Dynamics and Related Areas. (The first four proceedings were published by Pineridge Press and the fifth conference proceedings was published by the Engineering Research Center at Mississippi State University.) The literature on surface grid generation and practical applications can also be found in the NASA conference *Proceedings on Surface Modeling, Grid Generation, and Related Issues in Computational Fluid Dynamics Solutions* of 1992 and 1995.

In view of the importance of and worldwide interest in grid generation, the organizing committee of the 5th International Conference has proposed an establishment of the International Society of Grid Generation (ISGG). The ISGG will be a focal point for assimilating the progress and advances realized in grid generation by publishing a quarterly electronic journal of grid generation and a newsletter and by maintaining a grid generation Internet index to the grid generation literature, researchers, test cases, and information on public domain and commercial geometry-grid systems. The ISGG can also be the focal point for organizing future grid generation related workshops and conferences. The organization committee feels that the time is right for the emergence of a formal society and journal for grid generation. Additional information on the ISGG can be obtained from Bharat Soni, NSF Engineering Research Center, Mississippi State University, by e-mailing: bsoni@erc.msstate.edu.

27 Scientific Visualization

William R. Sherman
National Center for Supercomputing Applications

Alan B. Craig
National Center for Supercomputing Applications

M. Pauline Baker
National Center for Supercomputing Applications

Colleen Bushell
National Center for Supercomputing Applications

27.1 Introduction

The field of scientific visualization is broad and requires technical knowledge and an understanding of many communication issues. This chapter provides information about its evolution, its uses in computational science, and the creative process involved. Also included are descriptions of various software tools currently available and examples of work which illustrate various visualization techniques. Relevant concerns, such as visual perception, representation, audience communication, and information design, are discussed throughout the chapter and are referenced for further investigation. An overview of current research efforts provides insight into the future directions of this field.

27.2 Historic Overview

Visualization did not begin after the advent of computers. There has always been a need for people to visualize information. At the dawn of human history, humans began spreading pigment on surfaces to convey events that took place and later to indicate quantities of goods. From that time on, the medium of choice for representing such information has continued to evolve (Figure 27.1).

In general, visualization efforts required that the creators of the image represent their data by hand. Often this was a painstaking process that involved an artistic ability to mentally envision a pictorial representation of a phenomenon and the manual skills required to transpose the mental image into a suitable medium.

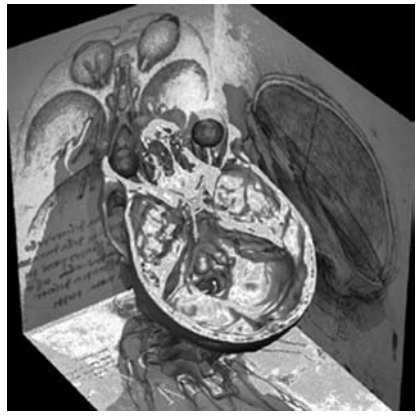


FIGURE 27.1 A combination of scientific representations from the fifteenth century and today demonstrates that the craft of visualization has been practiced for many years. (Courtesy of U. Tiede, T. Schiemann, and K. H. Hohne, IMDM, University of Hamburg, Germany, 1996.)

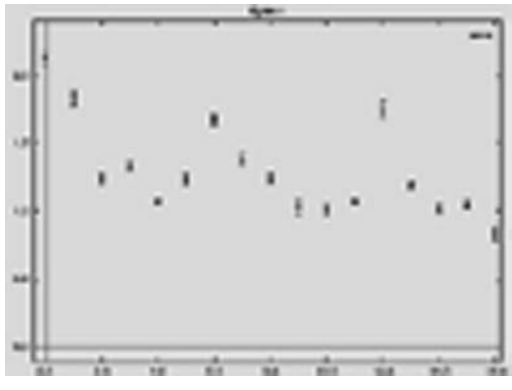


FIGURE 27.2 An XY plot with error bars.

The researcher had to be a capable artist and craftsman as well as a scientist. Usually, the visualizer would render the representation onto paper. However, other media for visualization were used as well.

As the scientific method developed, certain forms of visualization became accepted practices. As a scientist observed a phenomenon, it could be recorded onto an XY plot, representing the relationship between two quantities. A line was often drawn through the data to show the probable continuous pattern. Error bars were added to represent uncertainty in the data (Figure 27.2). We can now render detailed, data-based visual images by machine to show both quantitative relationships and qualitative overviews. How this process is accomplished, and its value to the scientific method, demand investigation.

27.2.1 The Motivation for Computer-Generated Visualization

The advent of the digital computer brought about the ability to collect, create, and store more information than previously. It also brought about a new method of science: *computational* [Kaufmann and Smarr 1993].

Computational science is the process of simulating a relevant subset of the laws of nature using a supercomputer. The laws of nature are described by a set of equations, which yield numeric solutions regarding the science being studied. Because these computational simulations of nature produce vasts

amounts of numeric information, a scientist may not be able to see, much less interpret, all the results. Fortunately, as the computational power of computers has increased, allowing these complex simulations to be calculated, so has the graphical power of computers increased. Thus, we also have access to a medium capable of creating and presenting all this information in a way useful to the researcher.

There are tradeoffs in how numeric information can be visualized. Interactive visualization gives the researcher the ability to control specific portions of the dataset to examine and to control the type and parameters of the visual output. Interactivity, though, may limit the percentage of the data that can be examined at a given moment and limit the types of representation available. Alternatively, the data display may be created as a batch process, allowing complex representations which are not possible in real time. The researcher may take advantage of both methods by beginning with interactive exploration and, when an interesting region of the data is located, producing a detailed animation.

Another consideration when creating visualization is whether to render a view of the entire dataset (a qualitative overview) or to precisely represent a subset of the data that the scientist can analyze (a quantitative study). Both are important in computational science. The qualitative overview can give the scientist a sense of the entire simulation, which can help in comparisons with observed nature. Because it gives an overall understanding of the dataset, it provides a sense of context when looking at the details.

The details are in the quantitative representation provided by a precise mathematical description. Qualitative information is helpful to this process, but high-resolution quantitative displays are essential. Representations such as contour plots and two-dimensional (2-D) vector diagrams are precise and aid in data analysis. Quantitative representations, such as these, provide the ability to pore over a particular subset of the data, even to the point of measuring phenomenon from the display. Because there is a limit to the amount of useful information that can be rendered on a screen, focusing on a subset allows the data to be more completely displayed.

27.2.2 The Process of Computational Science in Relation to Visualization

To understand the role of scientific visualization in the computational science process, we must first review the scientific process itself. Figure 27.3 depicts the steps involved in the computational science process [Arrott and Latta 1992]. Computational science begins with observations of some natural phenomenon,

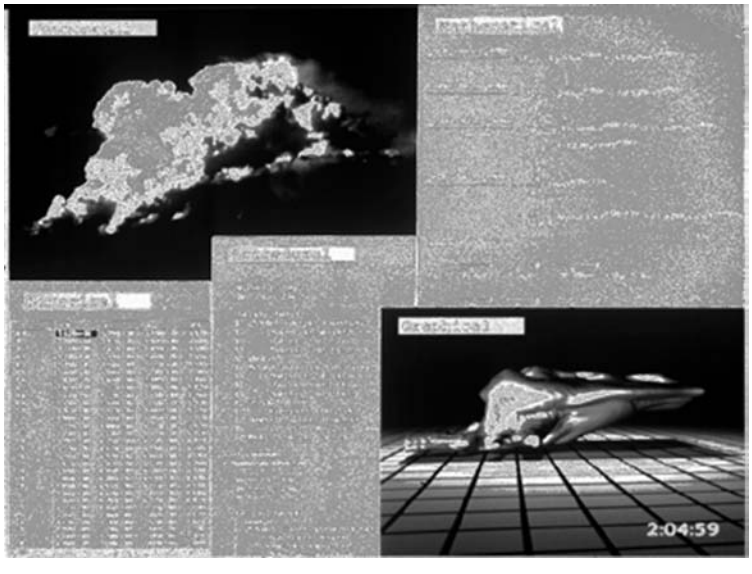


FIGURE 27.3 Computational science consists of observations, equations, algorithms, numerical solutions, and graphical representations. (Courtesy of Matthew Arrott; NCSA.)

in this particular case the formation of a severe thunderstorm. The scientist then expresses the observations in mathematics — the language of science. These equations can be manipulated by the researcher, though the problems are generally of a complexity sufficient to require solution on a supercomputer. In today's computing environment, the mathematical representation of the phenomenon is not a suitable means of input to typical computing systems. The computer requires the phenomenon be simulated in discrete steps of space and time, whereas mathematics allows a continuous representation. The mathematical representation is therefore translated into a programming language, implementing appropriate algorithms for a discrete numerical solution.

The resultant solution is typically a numeric value or set of values — a **dataset**. While the scientist may be able to gain insight from these numeric representations, a more intuitive visual form often aids in understanding. Also, others are often much more likely to understand the scientist's work when it is presented visually than as numbers only.

27.2.3 What Exactly Is Scientific Visualization?

Although we have previously discussed methods by which scientific data were represented before the advent of computers, for our purposes, **scientific visualization** is the use of data-driven computer graphics to aid in the understanding of scientific information. We will refer to an artist's rendering of a concept as illustration, or more specifically **scientific illustration**.

Is scientific visualization just computer graphics, then? Computer graphics is the medium in which modern visualization is practiced; however, visualization is more than simply computer graphics. Visualization uses graphics as the tool to represent data and concepts from computer simulations and collected data. Visualization is actually the process of selecting and combining representations and packaging this information in understandable presentations.

27.2.4 Other Modes of Presenting Information

There are ways to present information other than visually, of course. The primary of these is aurally, but one also might imagine the use of haptic (force, texture, temperature, etc.) display [Brooks et al. 1990], or even display to our other senses. Currently, sound (sonification) is increasingly recognized as an important method of information display for special types of data [Scaletti and Craig 1991].

Perhaps the field of representing information would be more appropriately termed **perceptualization**. The goal is, after all, to increase the information observer's perception of what is taking place in the data.

27.2.5 Application Areas

The use of scientific visualization to represent data is as broad as science itself. It spans the range of scales from the atomic and subatomic worlds to the vastness of the universe. It encompasses the study of complicated molecules and the building of complicated machinery. It looks at dynamic systems of living creatures and at the dynamics of whole ecosystems. Each of the areas touched by scientific visualization has representations that are particular to itself. Yet, there is much overlap in the techniques used by visualization developers due to commonality in the underlying mathematical expressions of natural systems.

Often a variety of what would seem unrelated sciences share similar or identical computational techniques. For example, computational fluid dynamics is used to study atmospheric effects, ocean currents, cosmology, mixing, injection molding, blood flow, and aeronautics. Finite-element analysis is used for solid and structural mechanics, fracture mechanics, crash-worthiness, heat transfer, electromagnetic fields, soil mechanics, metal forming, etc. Beyond these, there are many other fields that benefit from similar computational algorithms, including molecular modeling, population dynamics, diffusion, wave theory, and n-body problems.

27.2.6 Evolution of Scientific Visualization

The representation of the numeric output of simulations has developed from the simple printing of characters on paper, to vector display and plotter graphics, to three-dimensional (3-D) static images, to animated 2-D and then 3-D renderings of a simulation over time. The level of interaction has increased from the creation of visualization animations in batch mode, to real-time viewpoint control of fixed geometries, to interactive rendering allowing modification of the simulation in real time, and now to interacting with the simulation and representations in an immersive virtual environment.

As the underlying tools have improved, so have the idioms for representing information. New idioms are developed, and old ones are used in new ways. Many advances in representation are able to occur because of the advance of computing power and of computer input and output enhancements. Faster computing means more graphical computations can be done to create images, and higher-resolution displays allow for more detail to be presented.

These advances bring higher expectations. When 3-D pictures, animated 2-D pictures, and then finely rendered 3-D animations were first used by researchers to present their work, the visualization might have been considered the highlight of the presentation rather than the underlying science. The broader the audience, the more likely for this to be the case, giving rise to a situation where the scientists' credibility to the audience may be more correlated with the beauty of their images than with the underlying scientific theory. If scientists do not use the latest methods of computer graphics and animation to present their work, then it may not receive the attention it deserves. We are not arguing that this is how it *should* work, but it is important for scientists to be aware of the impact that visualization has on the communication of their research.

27.3 Underlying Principles

In this section, we will look at the various reasons for using scientific visualization and the effect they have on how a visualization is produced. We will also examine the basic concepts of visualization production and some of the considerations a producer should think about. Why are these important? Because scientific visualization is a means of communication. Sometimes the communication is between the raw numeric data and the researcher, and sometimes it is between the researcher and a group of people. Either way, for effective communication, it is important that both the producer of the visuals (or the tools used to create the visuals) and the audience have a grasp on what happens to the information as it passes from numbers to pictures.

27.3.1 The Goal of Scientific Visualization

Recall that the reason scientific visualization is employed as a tool is to more readily gain insight into a natural process. There are other similar goals that may be accomplished with scientific visualization. For instance, the goal might be to demonstrate a scientific concept to others, in which case the medium, representation, and the degree of detail chosen vary with the audience. A presentation shown to other scientists in the field will differ widely from one shown to the public or to government bodies.

The amount and the level of explanation required in a visualization is based on the experience of the intended audience. Also, design choices should be made which determine the amount of interactivity possible for wider audiences. For example, presentations designed for a mass audience are typically designed as noninteractive video animations. Alternatively, by utilizing computer delivered media, such as CD-ROMs, or multimedia presentations over the Internet, a limited dataset can be presented with a limited selection of visualization options, allowing for some experimentation by the audience.

When the audience is only the individual scientist, the primary goal is to uncover patterns in the data. Still, the goals of the study can vary. The goal might be to compare the patterns in the simulated data with patterns observed in nature. The closer these patterns match, the more confidence the scientist has in the theory expressed in the computational algorithm. Or the goal might be to discover new patterns that

give clues to a better mathematical expression of the process. This is more frequently the case when the process is less well understood and the data are collected rather than simulated. For example, in analysis of the stock market, researchers might look for patterns that give rise to the ability to determine profitable opportunities.

27.3.2 The Basic Steps of the Scientific Visualization Process

Though it is possible to jump right into using visualization as a tool, there are several important steps that occur in the process of creating an effective visualization. At one level, scientific visualization can be thought of analytically as simply a transfer function between numbers and images, bearing in mind that this transfer may be irreversible and cause distortion of meaning.

At another level, visualization involves a barrage of procedures, each of which influences the final outcome and its ability to convey meaningful information. That is, the process of visualization includes consideration for data filtering, representation, potential inaccuracy, and human perception.

27.3.2.1 Data Filtering

Seldom is it possible to make pictures straight from the data source (a computational simulation or data acquisition system). Typically, work needs to be done on the data before they can be appropriately visualized. This can include cleaning up the data and performing operations on them that yield more useful data. Examples of cleaning up data are removing noise, replacing missing values, clamping values to be within the range of interest, etc. New numeric forms of the data often are derived in order to produce a dataset which will lead to greater insight. For example, the vertical vorticity of a fluid flow can be calculated from the horizontal wind velocity.

The medium of presentation may also be the cause of data filtering. From a practical point of view, it is often necessary to adjust the data so that they can be conveniently produced within the constraints of a particular medium. For example, a standard NTSC video device displays at a rate of 30 frames per second. Thus, the dataset must be adjusted to produce 30 images for each second of the animation. This is done by interpolating the data over time. To fill spatial gaps in the visual imagery, interpolation is also frequently done over space.

No matter what the medium, whenever any form of interpolation or other data-filtering operation is used, there are problems that can arise which may cause the imagery to be misleading to viewers who don't know or understand what has happened to the data.

27.3.2.2 Representation Issues

As noted at the beginning of this chapter, computational science involves choosing appropriate representations of the phenomenon being studied (e.g., numeric, mathematical, etc.). The representations of the data appropriate for computer manipulation are not necessarily the most conducive to human understanding. Representing numeric data so that they can be more readily perceived by the audience involves mapping those numbers to a geometric form, sonic waves, etc.

Visual representation of information requires a certain literacy on the part of the developer and the viewer [Keates 1996]. Minimal information will be communicated if the viewer is not able to understand the visual language the developer has chosen. The language elements (symbols) of visualization come primarily from the adoption and evolution of symbols used in other visual domains, particularly scientific illustration. When new symbols are created, care must be taken to ensure that adequate explanation is given.

Beyond representing the numerical output of the simulation, it is desirable to indicate information about the simulation itself. This includes items such as a representation of the grid of the computational domain, the coordinate system, scale information, and the resolution of the computation.

When choosing which representational idiom to use, it is important to consider several issues before coming to a decision. What type of information is being investigated? Is the primary goal to convey quantitative or qualitative information? How detailed should the representations be? Making these determinations

depends to a great extent on the characteristics of the audience for which the visualization is intended. The resolution of the display affects the ability to present quantitative information and thus is a factor in which idioms can be chosen.

The goal of the visualization limits the medium of delivery. The medium, in turn, puts constraints on the possible choice of representation and interaction. So, for example, if motion is important to show some aspect of the data, then a medium that can support time-varying imagery needs to be used (e.g., film, videotape, interactive computer graphics). If the delivery medium is constrained to be a single image, then one must find a means to represent motion statically.

The ability to communicate with the audience relies on a well-designed presentation of information. A common problem is to give equal visual importance to all the elements in a scene, making it difficult to comprehend. We can learn techniques for making effective imagery from experts in information presentation such as graphic designers and instructional designers.

27.3.2.3 Accuracy

It is good practice for scientists to question the accuracy and validity of any information they are presented. All too often, compelling visualizations are used without the audience really questioning what they are seeing. Today, visualizations sometimes accompany peer-reviewed publications without being subjected to the same critical examination as the paper.

Where does inaccuracy come from, and what forms does it take? Whenever data change representation, the possibility for the introduction of error exists. Illusions are a danger in any medium. This is especially true when representing three-dimensional imagery on a two-dimensional display. For example, parallax can lead to misreading sizes of objects. The bias of the visualization producer often can affect the accuracy of the presentation. This does not necessarily imply that they might deliberately misrepresent the information, but many of the choices made during the production can add up to a presentation that gives an inaccurate view of the results. Some of these might include the choice of which representation to focus on, and which to leave in the background, or the selection of viewpoint or color and lighting that can make objects look ominous or insignificant.

High production values often lead to a sense of quality. The quality of the imagery does not necessarily reflect the quality of the underlying science or representations. The computer graphics techniques used should not get more emphasis than the science (i.e., should not have "glitz" merely for its own sake). Adding glitz can make a visualization appealing but can also occlude the important elements of the presentation. High production values and accuracy are two separate factors and should not be confused. The overuse of glitz in visualization is satirically treated in the animation *The Dangers of Glitziness and Other Visualization Faux Pas* (Figure 27.4) [Lytle 1993].

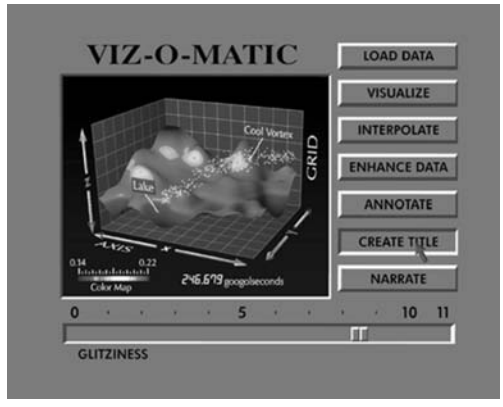


FIGURE 27.4 The Viz-o-matic animation pokes fun at the tendency to overuse graphic elements at the expense of accurate portrayal of the data. (Courtesy of Wayne Lytle; CTC.)