APPLICATIONS FOR ARITHMETIC ERROR CODES IN LARGE, HIGH-PERFORMANCE COMPUTERS*

Algirdas Avizienis†

†UCLA Computer Science Department
University of California
Los Angeles, CA 90024 USA

C. S. Raghavendra††

††Electrical Engineering-Systems Department
University of Southern California
Los Angeles, CA 90089 USA

ABSTRACT

Large, high-performance computers are too costly to allow full replication for fault detection and error correction in the communication and processing of numerical information. For this reason more cost-effective arithmetic error code applications offer an attractive alternative.

Part I of this paper presents a generalization of low-cost inverse residue codes into two-dimensional encodings. Error detecting and error correcting properties of two-dimensional inverse residue codes are discussed.

Part II discusses a multi-phase application of inverse residue codes in which the form of encoding is altered and additional time is allocated after faults occur. The goal is to defer repair and to continue operation at a slower speed until scheduled maintenance can take place.

Part I

Two-Dimensional Low-Cost Arithmetic Error-Detecting Codes

by Algirdas Avizienis

1. Introduction

A general approach to the cost and effectiveness study of low-cost arithmetic error codes has been presented in [AVIZ 71a]. This paper introduced the concepts of *inverse residue* codes and of *multiple* arithmetic error codes. The concept of *repeated use faults* was presented and the effectiveness of various arithmetic codes with respect to both *determinate* and *indeterminate* repeated-use faults was established. An important result was the proof that inverse residue codes can detect the "compensating" determinate repeated-use faults that are not detected by ordinary residue codes. The modulo 15 inverse residue code was applied in the JPL-STAR experimental computer [AVIZ 71b]. Further results on determinate faults were presented in [PARH 73] and [PARH 78]. An extension to signed-digit arithmetic is found in [AVIZ 81]. A.M. Usas has demonstrated the advantages of inverse residue codes for multiple unidirectional error detection, when compared to inverse checksum codes [USAS 78].

A new generalization presented here extends the application of low-cost inverse residue codes into two dimensions: row (byte) and column (line) residues. This extension improves the detection of errors, especially of those due to indeterminate faults, and provides certain error-correction capabilities.

The purpose of this paper is to demonstrate the advantages offered by two-dimensional inverse residue codes in the detection and correction of errors that affect byte-wide communication paths and processing elements. Such paths are widely used in high-performance array processors and for inter-processor communication in large multi-processor systems. Byte-wide processing elements are very suitable for the implementation of large processing arrays [AVIZ 70], [TUNG 70] and variable-precision signed-digit arithmetic [AVIZ 62].

2. Models of the Communication Path, Codes and Faults

We consider a parallel *communication path* consisting of b bit lines $(0, 1, \ldots, b-1)$. A *message* X consists of kb bits, transmitted as k bytes $(X_0, \ldots, X_i, \ldots, X_{k-1})$ of b bits length each (Figure 1).

Six types of low-cost encoding are applicable:

(a) *Checksum Code:* the k bytes are summed modulo 2^b, and the checksum byte representing $C = 2^b \left| \sum_{i=0}^{k-1} X_i \right.$ is attached to the message (now $k+1$ bytes long).

(b) *Inverse Checksum Code:* instead of C above, the checksum byte represents the value $C' = 2^b - C$.

(c) *Residue Code:* the k bytes carry an error-detecting encoding byte X_k that represents the modulo 2^b-1 residue X' of the message X: $X' = (2^b-1)|X$; and the message is now $k+1$ bytes long. Usually the residue value $X'=0$ is represented by a string of b ones. If the all-zero message can exist, its residue will be b zeros, unless explicitly disallowed.

(d) *Inverse Residue Code:* the inverse residue byte X_k represents the value X'' that is the (2^b-1)'s complement of X'. It is obtained as $X'' = (2^b-1) - X' = (2^b-1) - (2^b-1)|X$; and the message is again $k+1$ bytes long. The residue value $X'=0$ is represented by b ones, and the inverse residue X'' in this case is represented by b zeros. The all-zero message X has an inverse residue code X'' represented by b ones.

(e) *Byte-Parity:* One parity bit is attached to each byte. One more *check* line is added to the communication path (now $b+1$ lines wide).

* This research has been performed at the UCLA Computer Science Department and supported by ONR contract N00014-79-C-866, "Research in Distributed Processing," and a Research Grant from the Battelle Memorial Institute.

(f) *Two-dimensional Residue (or Inverse Residue) Code:* Instead of parity bits, the check bits on the check line now represent the modulo $2^{k+1}-1$ line residue Y', or the inverse line residue Y'' of the message. The message is now treated as b lines of $k+1$ bits length each. If the residue is taken modulo 2^{k+1}, we obtain a checksum (or inverse checksum) encoding of the lines instead.

The faults of interest are of three classes:

(1) faults affecting single bits of the message;

(2) faults affecting one byte (or adjacent bytes) of the message;

(3) faults affecting one line (or adjacent lines) of the path.

The first and second classes are likely to result because of transient external interference with the transmission of the byte(s), or because of failures in the source of the information being transmitted. The third class are "repeated-use" faults due to failures of the transmission path itself.

Determinate (permanently stuck-on-one or stuck-on-zero) faults and *indeterminate* faults that vary between one and zero during repeated uses of the faulty line(s) need to be considered for each type of faults enumerated below:

1. *One-Byte Faults* (b bits per byte except $b+1$ bits when the check line is used): (a) one bit; (b) two adjacent bits; (c) any two bits; (d) m adjacent bits ($b+1 \geq m > 2$); (e) any m bits ($b+1 > m > 2$).

2. *Adjacent-Byte Faults*
same as in (1) above, for $p \geq 2$ adjacent bytes.

3. *Bit-Line (Repeated Use) Faults* ($k+1$ bytes per message; k when parity alone is used): (a) one line; (b) two adjacent lines; (c) any two lines; (d) m adjacent lines ($b+1 \geq m > 2$)

The *measure of effectiveness* of a given code class x is given in *miss percentage* $M(x)$ which is calculated as:

$$M(x) = \frac{\text{no. of undetectable error patterns}}{\text{total no. of possible error patterns}} 100$$

The *error patterns* are determined by assuming that the correct message may contain either a "0" or a "1" in each position affected by the fault. A "1" in the error pattern represents the change 0→1, and "$\bar{1}$" (minus one) — a change 1→0. A zero indicates that a given position is not changed by the fault. Thus an "all-zero" error patterns indicates that the fault does not cause an error for the given original message. The *bit error value* E_i^j assumes one of three values for a given bit X_i^j of Figure 1:

$E_i^j = 1$ for X_i^j changing 0→1

$E_i^j = \bar{1}$ for X_i^j changing 1→0

$E_i^j = 0$ for no change in X_i^j.

For example, a "stuck-on-one" fault affecting two adjacent bit positions (containing 00, 01, 10, or 11) will produce four possible error patterns

$$11(00 \to 11), 10(01 \to 11), 01(10 \to 11), \text{ and } 00(11 \to 11)$$

The miss percentage for parity is: $M(\text{parity}) = 100(1/4) = 25\%$, since only the 00→11 transformation goes undetected. A "stuck-on-X" indeterminate fault in two adjacent bit positions

Check line b	line $b-1$		line j		line 1	line 0	byte
X_0^b	X_0^{b-1}	...	X_0^j	...	X_0^1	X_0^0	X_0
⋮	⋮		⋮		⋮	⋮	
X_i^b	X_i^{b-1}	...	X_i^j	...	X_i^1	X_i^0	X_i
⋮	⋮		⋮		⋮	⋮	
X_{k-1}^b	X_{k-1}^{b-1}	...	X_{k-1}^j	...	X_{k-1}^1	X_{k-1}^0	X_{k-1}
X_k^b	X_k^{b-1}	...	X_k^j	...	X_k^1	X_k^0	check byte
Check bits	message bytes (b bits each)						

Figure 1 — Model of the Path and Messages

will produce 16 error patterns of nine distinct types, of which those with two nonzero entries (total of 4) will be undetected. Thus the miss percentage again is $M(\text{parity}) = 100(4/16) = 25\%$.

In general, when a determinate fault affects m positions, 2^m (unidirectional) error patterns are possible. When the fault is indeterminate, there are 2^{2m} (bidirectional) error patterns, of which 3^m are distinct.

3. Detection and Correction of Unidirectional Errors Due to Determinate Faults

Given a modulo 2^b-1 inverse residue code, the undetectable unidirectional errors are those that have *message error values* E congruent to zero modulo 2^b-1, where

$$E = \sum_{j=0}^{b-1} \left(\sum_{i=0}^{k} E_i^j \right) 2^j$$

All other unidirectional errors will be detected; however, there are no error correction properties.

There is one undetectable one-byte unidirectional error; it results when an all-zero byte X_i is changed to an all-ones byte, or vice versa. The miss percentage for this "stuck byte" fault is $(100/2^b)\%$. Introduction of byte parity bits will detect only one of the two (stuck-on-one and stuck-on-zero) "stuck bytes."

The stuck byte detection problem is fully solved by the use of two-dimensional inverse residue encoding. There is one additional check bit X_i^b for each byte X_i ($i=0, \ldots, k$). The check bits ($X_k^b, \ldots, X_0^b$) represent the modulo $2^{k+1}-1$ inverse line residue Y'' of the message X that is now interpreted as b lines X^j ($j=0, \ldots, b-1$) of $k+1$ bits length each. It is evident that every "stuck byte" now will be detected by the use of Y'' as long as the condition $(b+1) < (2^{k+1}-1)$ is satisfied.

In general, the remaining undetectable errors in the message X are those that are missed by *both* checks: modulo 2^b-1 over the bytes (not including the check line bits X_i^b), and modulo $2^{k+1}-1$ over the lines, with the check byte bits X_k^j included in each line j. Most unidirectional errors are detectable; furthermore, the detection of bidirectional errors is significantly improved, as discussed in the next section.

The introduction of the inverse line residue Y'' also makes error correction possible. As shown in [AVIZ 71a], the low-cost inverse residue codes have the "partial error location" property. Therefore a single-bit error value $E_i^j = \pm 1$ $(0 \le j \le b-1;\ 0 \le i \le k)$ will produce a unique indication for line j in the modulo $2^b - 1$ check and for the byte i in the modulo $2^{k+1}-1$ check, making a correction of E_i^j possible in the message X.

The correction property can be extended to unidirectional single-line errors as follows. If we assume a determinate single-line fault on line j, the message error values $E(j)$ will fall into the range:

$$-2^j \sum_{i=0}^{k} E_i^j \le E(j) \le 2^j \sum_{i=0}^{k} E_i^j$$

The positive values will be due to a stuck-on-one (s-o-1) and negative values — due to a stuck-on-zero (s-o-0). The actual byte check results will assume the values $C(j) = (2^b - 1) \Big| E(j)$, and as long as $(k+1) < (2^b - 1)$ holds, all error values due to a s-o-1 fault will be detectable and have a unique byte check result $C(j)$ in the range

$$0 \le C_1(j) \le (2^b - 1) \Big| (k+1)2^j$$

Similarly, the error values due to a s-o-0 fault will have the byte check result in the range:

$$0 \le C_0(j) \le (2^b - 1) \Big| (-2^j)(k+1)$$

However, many other error patterns (on two or more lines) can produce the same values of check results, and error correction is not possible with the byte residue encoding alone.

To obtain single-line unidirectional error correction, we use the additional information provided by the line check result obtained from the inverse line residue encoding. Given a byte check result $C_1(j)$ discussed above, we find its value to be N, represented by b bits $(N_{b-1}, \ldots, N_0)$.

First we form the hypothesis that N is due to a single-line stuck-on-one determinate fault on line j $(0 \le j \le b-1)$. If the fault is on line $j = 0$, then $N(0) = N$ error bits $E_i^0 = 1$ in line 0 will produce the byte check result N. We determine the numbers $N(j)$ of error bits $E_i^j = 1$ on lines $j = 1, \ldots, b-1$ respectively that would be needed to produce the byte check result N by end-around shifting N to the right $b-1$ times. The shifts will produce the numbers $N(1), \ldots, N(b-1)$ in succession.

The number of error bits $E_i^j = \bar{1}$ (due to a stuck-on-zero line) that would be needed to produce $N(j)$ for any $0 \le j \le b-1$ is given by $(2^b - 1) - N(j)$, that is, the "one's complement" of $N(j)$. All values of $N(j)$ and $(2^b - 1) - N(j)$ that are greater than $k+1$ are discarded as impossible solutions.

To test the hypothesis that a given byte check result N is due to a single-line determinate fault, we use the line check result

$$R = (2^{k+1} - 1) \Big| \sum_{j=0}^{b} \left[\sum_{i=0}^{k} X_i^j 2^i \right]$$

This result will contain $N(j)$ digits $R_i = 1$ $(0 \le i \le k+1)$ if there is a single-line determinate (stuck-on-one) fault in the line j. The presence of each $R_i = 1$ indicates that the digits X_i^j should be corrected by the $1 \to 0$ change.

The line check result R will contain $(2^b - 1) - N(j)$ digits $R_i = 0$ $(0 \le i \le k+1)$ if there is a single-line determinate (stuck-on-zero) fault in the line j. The presence of each $R_i = 0$ indicates that the digit X_i^j should be corrected by the $0 \to 1$ change.

Example 1: Single-line Error Correction

Consider a message X with seven bytes $(k = 7)$ of 4 bits each $(b = 4)$. Inverse reside coding is used for the bytes (modulo $2^b - 1 = 15$) and for the lines (modulo $2^{k+1} - 1 = 255$). The encoded message (following Figure 1) is shown below:

check line	line 3	line 2	line 1	line 0	
1	0	1	0	0	byte 0
0	0	0	1	1	byte 1
0	1	0	0	1	byte 2
0	0	0	0	0	byte 3
0	1	0	1	1	byte 4
1	0	0	1	0	byte 5
0	1	0	1	0	byte 6
0	0	1	1	0	check byte 7

The byte check result (modulo 15) is $N = 1111$, and the line check result (modulo 255) is $R = 11111111$. No errors are indicated.

Now assume a stuck-on-one line 2 and set all digits in line 2 to one. The new byte check result is $N = 1001$. The single-line determinate fault possibilities are:

Stuck-on-One	*Stuck-on-Zero*
$N(0) = 1001 = 9$	$15 - N(0) = 6$
$N(1) = 1100 = 12$	$15 - N(1) = 3$
$N(2) = 0110 = 6$	$15 - N(2) = 9$
$N(3) = 0011 = 3$	$15 - N(3) = 12$

The values greater than $k+1 = 8$ are discarded, and the remaining possibilities are: line 2 (6 errors) or line 3 (3 errors) stuck-on-one, and line 0 (6 errors) or line 1 (3 errors) stuck-on-zero.

The new line check result is

$$R = (R_7, \ldots, R_0) = 01111110$$

The six ones in R indicate that the "line 2 stuck-on-one" hypothesis is valid, and the corresponding six positions in line 2 are corrected by setting them to zero.

Example 2

Now assume that line 1 is stuck-on-zero. The byte check result is $N = 0101$, and the possibilities are

Stuck-on-One	*Stuck-on-Zero*
$N(0) = 0101 = 5$	$15 - N(0) = 10$
$N(1) = 1010 = 10$	$15 - N(1) = 5$
$N(2) = 0101 = 5$	$15 - N(2) = 10$
$N(3) = 1010 = 10$	$15 - N(3) = 5$

The remaining possibilities all point to five errors. The modulo 255 line check result is

$$R = 00001101$$

The five zeros in R (positions 7,6,5,4,1) indicate a stuck-on-zero on line 1 or line 3. To resolve the ambiguity, we find that 3 already has "1" digits in positions 6 and 4, and cannot be corrected there; therefore it must be line 1.

4. Detection of Bidirectional Errors

It has been noted that low-cost inverse residue codes are considerably less effective in detecting bidirectional errors due to indeterminate repeated-use faults [AVIZ 71a]. The addition of the line residue (i.e., the second dimension of encoding) allows the detection of *all* bidirectional errors that affect a single line, as well as *all* bidirectional double errors affecting any two bits of the message X. The double bidirectional errors on one line that were undetected by the byte check are now detected by the line check, while those in one byte are detected by the byte check.

The remaining undetectable bidirectional errors are those that are simultaneously undetectable by the byte check and the line check. An illustration is the quadruple error that changes Z to Z^* as shown below:

$$Z = \begin{matrix} 0 & 1 \\ 1 & 0 \end{matrix} \Rightarrow Z^* = \begin{matrix} 1 & 0 \\ 0 & 1 \end{matrix}$$

Here an even number of opposite-direction changes occurs simultaneously in the bytes and lines of the message X. In general, all quadruple errors of this type (at four corners of a rectangle of bits within the message X) are undetectable.

Part II

Multi-Mode Applications of Arithmetic Codes in Interprocessor Communication

by Algirdas Avizienis and C.S. Raghavendra

1. Introduction

We consider computer systems that use a large number of processors, with high-speed communication among processors in the execution of programs. In such a system the interprocessor communication must be highly reliable. Examples of such systems are the Data Flow Machine [DENN 80, LEUN 80] and the cellular architecture for executing reduction language programs [MAGO 79].

In the MIT data flow machine, a large number of processing elements is organized as a network of self-timed modules which communicate by sending packets to each other. In this architecture, packet communication is completely asynchronous, and packet bytes are transmitted and received in byte-serial manner. It is extremely important that the packet communication should be protected from hardware failures and transient faults, as otherwise the performance of this machine will be severely degraded. The coding scheme described in this paper is one approach to provide fault-tolerance in data flow and such other machines.

2. A Three-Mode Checking Scheme

The approach consists of three modes of checking that offer increased error correction capability at the cost of additional time required for message transmission.

In Mode A, the two-dimensional inverse residue checking is applied. The messages are as shown in Figure 1, with the typical value of byte length $b=8$ and modulo 255 inverse residue encoding. The number k of bytes per message can be varied; $k=8$ or $k=16$ appear to be typical; such that line checking is module $2^8-1=511$, or $2^{17}-1$. The cost of encoding is one added bit line (the check line), and one extra byte transmission time per message (for the check byte).

When an incorrectable error is detected in Mode A operation, or a permanent one-line fault occurs, Mode B is invoked. In Mode B every byte X_i of the message is provided with its own check byte of the value $\overline{X}_i=(2^b-1)-X_i$, which is the one's complement of X_i. The check line now carries byte parity both for X_i and $\overline{X}_i$ respectively. The time to transmit the message is now $2k$ byte times.

Byte check results are formed for every pair of bytes. A check sum (modulo 255) of all ones means that either there is no error or there may be some compensating errors. A *compensating double error* occurs when a particular bit (say the jth) of both X_i and $\overline{X}_i$ are incorrect. That is, the received values are interchanged. If any one position of the check result has a single 0 or a single 1, then we have detected a single error. The parity bits are used to find out whether the error is in information byte X_i or code byte $\overline{X}_i$. This also allows us to correct the single bit error because we know which bit is in error. Any other check result indicates multiple errors, and such errors can only be detected. In fact, this code scheme allows us to detect all possible unidirectional errors (due to stuck-at type faults). It should be noted that the check adder can be transformed to Exclusive-OR circuits by disconnecting the carry connection and still serve as a byte comparator for Mode B.

The only type of error that cannot be detected when operating in Mode B of the coding scheme is that with an even number of compensating errors. Odd numbers of compensating errors can be detected because in such situations both the parity bits will indicate an error, but the checker will not. The occurrences of compensating failures are quite rare in digital systems; the chance of two or more compensating failures will be extremely small.

When error correction cannot be performed, i.e., when multiple errors are detected, we switch to Mode C. In this mode, for each byte X two more bytes X' and X'' are generated by rotating the bits of X by $n1$ and $n2$ positions respectively. Now, the redundancy is increased to 200% and hence there is a further degradation in performance. This is basically redundancy in time and thus no complex hardware is required. The bits of X, X', and X'' travel on different physical wires because of the rotation of bits. At the receiver the bits of the three bytes are voted by majority voters. We note that the Carry output of a binary full adder (FA) serves as a Majority output and the Sum output serves as a Disagreement Detector when all three FA inputs accept the inputs to be voted; then the same checking hardware is still usable.

Mode C operation allows the masking of all single errors and of many double and triple errors. If the double and triple errors are such that they don't occur on same bits of X, X', and X'', they can be masked by the voters. In fact, for any value of $n1$ and $n2$ ($n1 \neq n2$) there are N combinations of double and N combinations of triple errors that can be masked. Since transient faults may introduce multiple errors, and usually on adjacent lines, we select $n1$ and $n2$ such that there is maximum protection from transient faults. The optimum values of $n1$ and $n2$ for byte serial communication are 3 and 6 respectively. This selection allows maximum separation between ith bits of X, X', and X''

3. Reliability Analysis

In this section we perform reliability analysis of the coding scheme and derive a simple expression for reliability of communication. The parameters used in the analysis are: R, the reliability of a line, and N, the number of lines. We have the following cases for which the communication is correct:

(1) All lines are working;
(2) Any one out of N lines failed;
(3) Any two failures such that they are masked by voters;

(4) Any three failures such that they are masked by voters.

The circuits required for generating error code, checking circuits, circuits for rotating information bits, and voting circuits are assumed to be protected by other forms of fault-tolerance, such as self-checking. Their reliability is not included here. We also assume that the coverage for switching from one mode to another is 1.

Now we have the following simple expression for reliability of communication,

$$R_c = R^N + NR^{N-1}(1-R) + NR^{N-2}(1-R)^2 + NR^{N-3}(1-R)^3$$

The amount of rotation of information bits in mode B can be any number of positions. For maximum protection against transient faults the amount of rotation should be $N/3$ and $2N/3$ for the second and third bytes respectively.

We have assumed previously that the probability of undetected errors is extremely small. Now we give a quantitative estimate of this probability. The probability of undetected errors corresponds to the occurrence of an even number of compensating errors. A simple expression for this probability, under the pessimistic assumption that the probability of a normal error is the same as the compensating error, is given by:

$$R_f = {}^N C_2(1-R)^4 + {}^N C_4(1-R)^8 + \ldots.$$

Only the first term will be significant. With $N=9$ and $R=0.95$ this probability is,

$$R_f = 2.25 \times 10^{-4}$$

A more realistic assumption about the probability of compensating errors is that it is ten times smaller than the normal errors, in which case we have:

$$R_f = 2.25 \times 10^{-8}$$

Therefore, we can conclude that the probability of detecting all types of errors with the proposed coding scheme is extremely high.

4. Concluding Remarks

The goal of Part II has been to point out the flexibility and convenience of arithmetic checking of data communication in large-scale computers. We also note that arithmetic codes can be used to check the numeric operations that follow the interprocessor transmission of operands. These factors strongly suggest the superiority of arithmetic codes to other methods of checking.

An early successful application of error correction by alternate data retry is found in the "Output Addressing" circuitry of the fault-tolerant Saturn V Launch Vehicle Digital Computer [ANDE 67]. The same computer also extensively used "TMR" majority voting to protect its serial CPU. Arithmetic codes were extensively used in the JPL-STAR computer [AVIZ 71b]. Thus the approach discussed here is not an innovation of method, but a synthesis of time-tested methods to fill an important need in a flexible manner. Recent work, such as [SHED 78] and others also address the alternate-data retry; however, in our case it is the coding rather than alternation that defines the mode of operation.

REFERENCES

[ANDE 67] Anderson, J.E., Macri, F.J., "Multiple Redundancy Applications in a Computer," *Proc. 1967 Ann. Symp. Reliability*, Washington, D.C., 1967, pp. 553-562.

[AVIZ 62] Avizienis, A. "On a Flexible Implementation of Digital Computer Arithmetic," *Information Processing 1962*, C.M. Popplewell, ed., North Holland Publishing Co., Amsterdam, 1963, pp. 664-670.

[AVIZ 70] Avizienis, A., Tung, C., "A Universal Arithmetic Building Element (ABE) and Design Methods for Arithmetic Processors," *IEEE Trans. on Computers*, C-19: 733-745, August 1970.

[AVIZ71a] Avizienis, A. "Arithmetic Error Codes: Cost and Effectiveness Studies for Application in Digital System Design," *IEEE Trans. on Computers*, C-20: 1322-1331, November 1971.

[AVIZ71b] Avizienis, A., et al., "The STAR (Self-Testing and Repairing) Computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design," *IEEE Trans. on Computers*, C-20: 1312-1321, November 1971. Reprinted in *Best Computer Papers of 1971*, L. Petrocelli, ed., Auerbach Publishers, 1972, pp. 165-185.

[AVIZ 81] Avizienis, A., "Low-Cost Residue and Inverse Residue Error-detecting Codes for Signed-Digit Arithmetic," *Proc. 5th Symposium on Computer Arithmetic*, 1981, pp. 165-168.

[DENN 80] Dennis, J.B., "Data Flow Supercomputers," *Computer*, November 1980, pp. 48-56.

[LEUN 80] Leung, C.K.C, Dennis, J.B., "Design of a Fault Tolerant Packet Communication Computer Architecture," *Proc. of the 1980 Fault Tolerant Computing Conference*, Kyoto 1980, pp. 328-335.

[MAGO 79] Mago, G.A., "A Network of Microcomputers to Execute Reduction Languages," *Int. Jour. of Computer and Information Sciences*, Vol. 8, No.5,6, 1979, pp. 349-385, 435-465.

[PARH 73] Parhami, B., Avizienis, A., "Application of Arithmetic Error Codes for Checking of Mass Memories," *Digest of the 1973 Int. Symposium on Fault-Tolerant Computing*, pp. 47-51, June 1973.

[PARH 78] Parhami, B., Avizienis, A., "Detection of Storage Errors in Mass Memories Using Low-Cost Arithmetic Codes," *IEEE Trans. on Computers*, C-27-4: 302-308, April 1978.

[SHED 78] Shedletsky, J.J., "Error Correction by Alternate-Data Retry," *IEEE Trans. on Computers*, C-27: 106-112, February 1978.

[TUNG 70] Tung, C., Avizienis, A., "Combinational Arithmetic Systems for the Approximation of Functions," *AFIPS Conf. Proc.* (1970 Spring Joint Computer Conf., Atlantic City, NJ), 36: 95-107, 1970.

[USAS 78] Usas, A.M., "Checksum Versus Residue Codes for Multiple Error Detection," *Digest of the 8th Annual International Conf. on Fault-Tolerant Computing*, p. 224, 1978.