elaborate error handling schemes would in all probability not be provided by hardware.

More elaborate parity schemes could detect errors and, in addition, could identify the positions of erroneous bits. For example, consider a two-dimensional parity scheme used on a finite square matrix of bits shown in Figure 4 (pictured as a 4 x 4 matrix for convenience). The horizontal parity would be stored as a transpose (stored horizontally like a row).

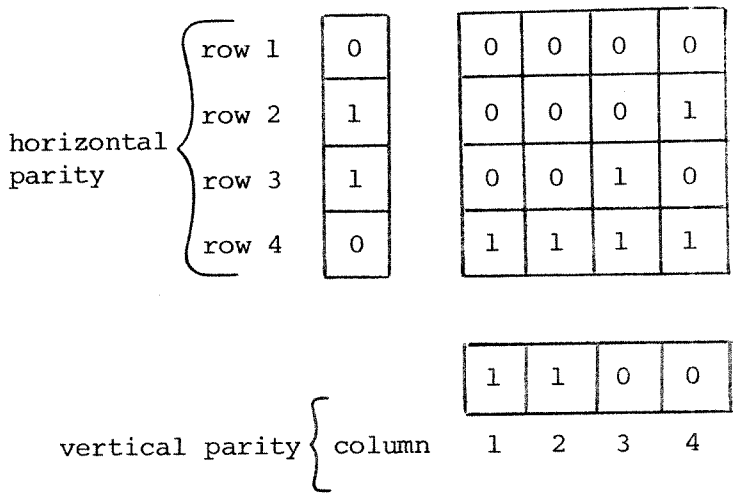


Figure 4: Horizontal and Vertical Parities

In a more powerful parity scheme, groups of the above matrices with horizontal and vertical parity could be combined into a three-dimensional scheme using Z-axis parity. This is illustrated in Figure 5.

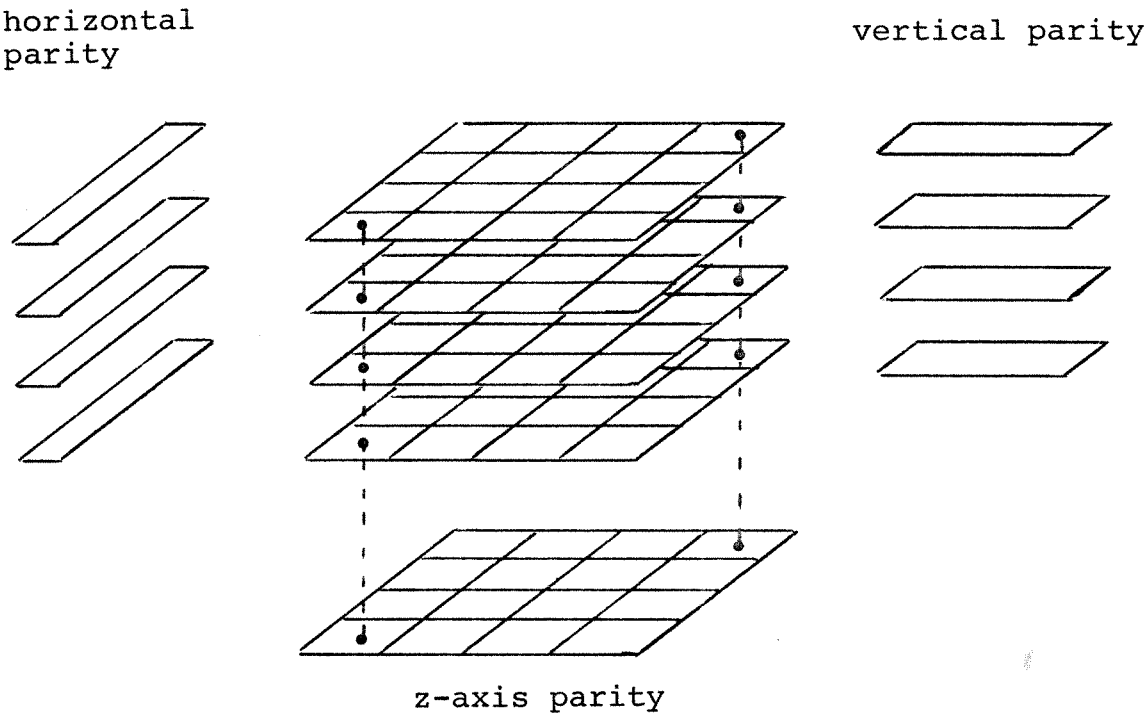


Figure 5: Horizontal, Vertical, and Z-axis Parities

In a more unusual scheme, consider parity taken on the diagonals of a matrix of bits as shown in Figure 6.

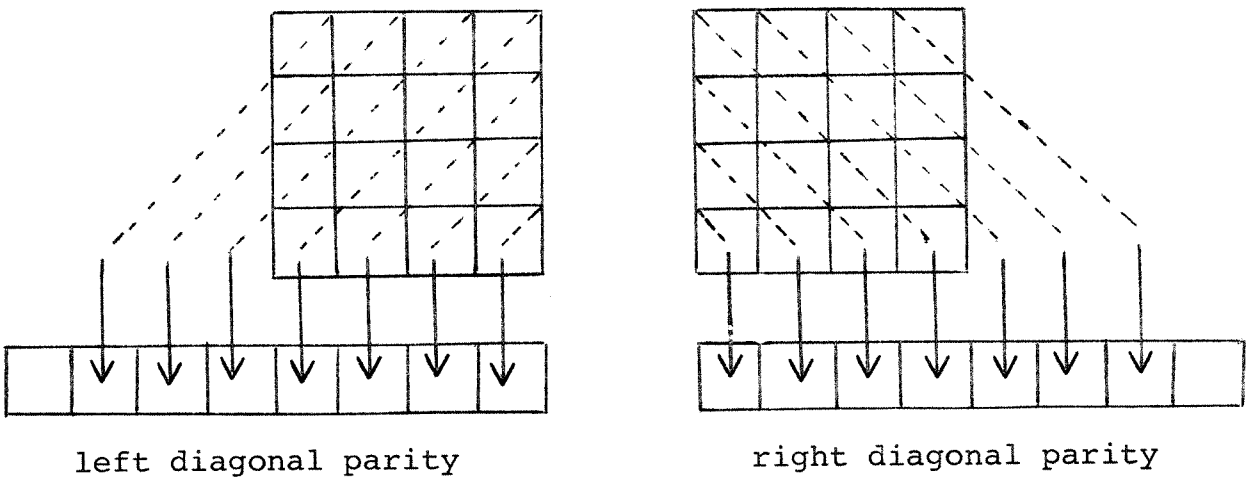
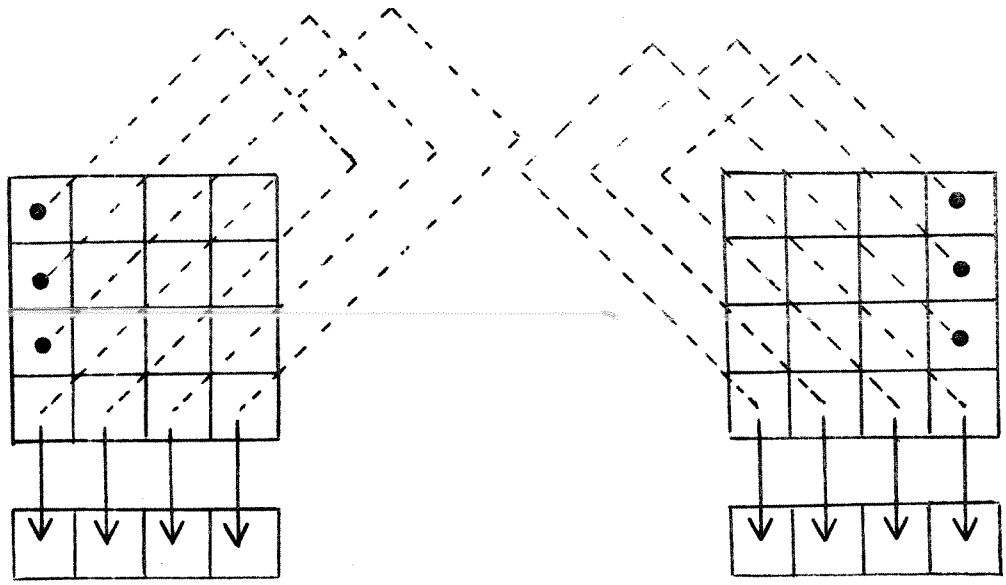


Figure 6: Left and Right Diagonal Parities

Note this scheme requires two rows to store each diagonal parity. Storage can be halved by allowing wraparound of the diagonals as shown in Figure 7. Note that wraparound does not cause any diagonal to occupy more than one position on any one column (or any one row).



left diagonal parity with wraparound

right diagonal parity with wraparound

Figure 7: Left and Right Diagonal Parities With Wraparound

LIMITATIONS OF PARITY AND CHECKSUM

Parity schemes involve comparing the current parity just computed with the original parity stored with the data. Any change detects that an error has occurred. However,

there are cases where no change is apparent between the original and the current parity--yet errors have occurred. Specifically the problem is one of camouflage due to self-compensating errors. Any even number of errors cancel each other out in the parity so that detection is impossible. Actually parity is designed to detect only single errors. It exceeds its function by always detecting an odd number of errors. Checksum also has design limitations. Table 3 shows two bits in their original state and demonstrates how two erroneous bits could never be detected by parity. In addition, the table shows how a checksum would fail to detect 50 percent of the time.

original			current with 2 errors			results
data	parity	checksum	data	parity	checksum	
00	0	0	11	0	2	checksum would detect
01	1	1	10	1	1	undetected even by checksum!
10	1	1	01	1	1	undetected even by checksum!
11	0	2	00	0	0	checksum would detect

Table 3: Detection Limitations Using a Checksum

This same problem appears whenever any even number of bits have changed; parity will fail to detect 100 percent of the

time. However, checksum detection improves as the number of erroneous bits increases. For example, it goes from a 50 percent success rate for two bits to a 63-1/3 percent success rate for four erroneous bits.

Let us put this perfectionist's nightmare into perspective. Such undetectable erroneous situations have a limited probability of occurring, especially when limiting the number of bits included in one parity scheme. See the discussion in the preceding section PROBABILITY OF ERRORS IN A DATA BASE. The error most often expected would be a change in one bit. This is always detectable with a parity check or with a checksum.

To fully evaluate error correction algorithms, attention must be devoted to all cases that the algorithms would conceivably have to analyze. The algorithm must be able to recognize situations that are beyond its designed capabilities and attempt no action. Indeterminant changes to the data must be avoided.

The following discussion of probability lays the foundation for evaluating various algorithms which are partially capable of reconstructing two or three errors in one nxn matrix. Most often, problems arise when multiple errors lie on the same direction such as on the same row or the same column. For an even number of errors in the same direction, the corresponding parity inaccurately detects no change.

Given that two errors occur in one nxn matrix,

the probability that they both occur in the same row is given by the formula:

$$P = \frac{n - 1}{n^2 - 1}$$

The formula is the same for the probability of two errors occurring in the same column or the probability of both occurring in the same diagonal. Combining these formulas, the probability that the two errors both occur in either the same row, same column, or same diagonal is given by the formula:

$$P = \frac{3(n-1)}{n^2-1}$$

Given that three errors occur in one matrix, the probability that two or three of these bits share a common direction is expressed in the formula:

$$P = \frac{(n-2)(n^2-6n+10)}{(n+1)(n^2-2)}$$

Corresponding formulas for an nxm matrix are not given. A bound on the probabilities may be obtained by selecting the larger of n and m. That number may be used as n in the above formulas.

ERROR CORRECTION ALGORITHMS

Various error correction algorithms require differ-

ent combinations of abbreviated additional information to reconstruct a finite matrix of bits called a packet. Selections are from the following which are shown with abbreviations.

- C - checksum for one packet
- H - horizontal parity on x-axis
- V - vertical parity on y-axis
- Z - Z-axis parity making a three-dimensional scheme
- D - diagonal parity with wraparound
  D_L means left diagonal
  D_R means right diagonal
- D1 and D2 - diagonal parity without wraparound
  $D1_L$, $D2_L$ mean left diagonal parity
  $D1_R$, $D2_R$ mean right diagonal parity

The error correction algorithms are defined by the additional information used as shown in Table 4. Here D may refer to either D_L or D_R although diagrams in this paper arbitrarily show D_L. Algorithms using D1 and D2 are not listed because there is no net advantage.

Algorithm HV

Algorithm HV requires horizontal parity and vertical parity to be computed and stored each time a matrix is changed. Each time the matrix is to be used, both parities are recomputed and used in an "exculsive OR" with the corresponding original parities. Any bits left on in the

REDUNDANCY	ERROR CORRECTION ALGORITHM						
	HV	HVC	HVZ	VD	VDC	VH	HVD
C		X			X		
H	X	X	X			X	X
V	X	X	X	X	X	X	X
Z			X				
D				X	X		X

Table 4: Error Correction Algorithms

result identify a row or column which may contain an error. The intersection of these two dimensions identifies a questionable bit. For the example diagrammed in Figure 8, there is only one bit in error for the entire matrix. Thus the identified intersection does contain the error, marked by * in the diagram.

Now that the exact location of the error has been detected, error correction is simple. Complement or flip the bit at the intersection (change "on" to "off" and "off" to "on").

However, confusion exists when two or more bits in a matrix become erroneous before a test. For a two bit error, either four or zero intersections are identified. The case of zero intersections would occur if the two errors appeared in either the same row or the same column. This would produce the camouflage effect discussed in the section LIMITATIONS OF PARITY AND CHECKSUM. In the case of four intersections, there is no extra information to

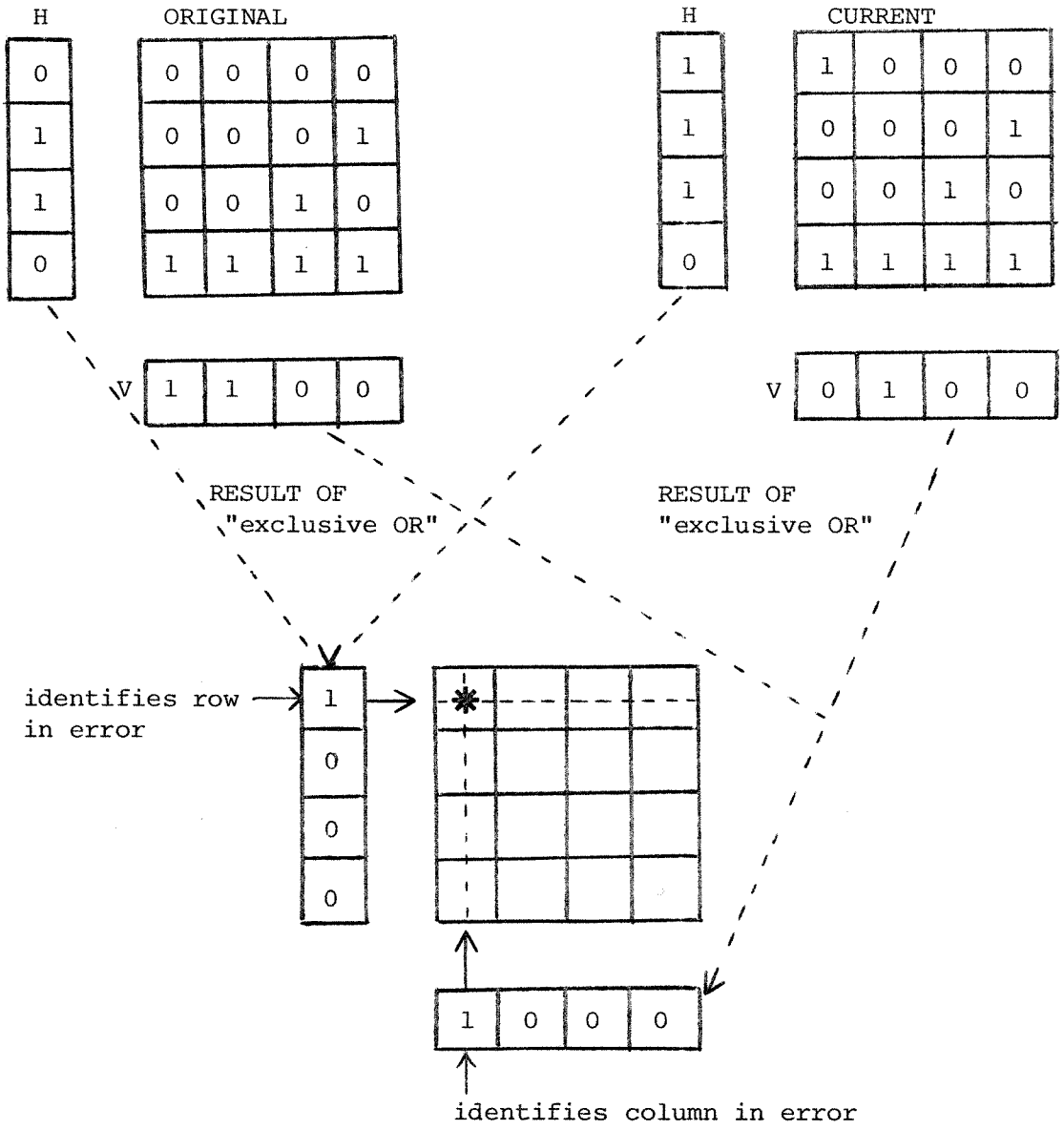


Figure 8: Error Correction Using Horizontal and Vertical Parities

distinguish which intersections are erroneous and which intersections are correct but are identified due to symmetry. Let us examine using Figure 9 the case of two bits in error but not in the same row or column. Label the intersections as X1, X2 in row X and Y1, Y2 in row Y. Either the pair X1, Y2 contains the erroneous bits and the pair X2, Y1 is correct but identified due to symmetry or vice versa. Algorithm HV is indeed capable of recognizing a double error in one matrix and would not attempt to reconstruct.

The logic that Algorithm HV would use is as follows: the number of bits turned on in the result of the "exclusive OR" involving the original and current horizontal parity should be equal to one to proceed with correcting a one bit error. In other words, one and only one erroneous column should be identified. This identifies a single intersection containing the error. Under any other conditions, the reconstruction would not be attempted.

There is a slight possibility that a triple error could be mistaken as a single error. If a triple error occurs such that each bit is at the corner of an imaginary rectangular (one bit in same row as another error and also in same column as a third error) then the fourth corner of the imaginary rectangle is inaccurately identified as the only erroneous bit in that packet.

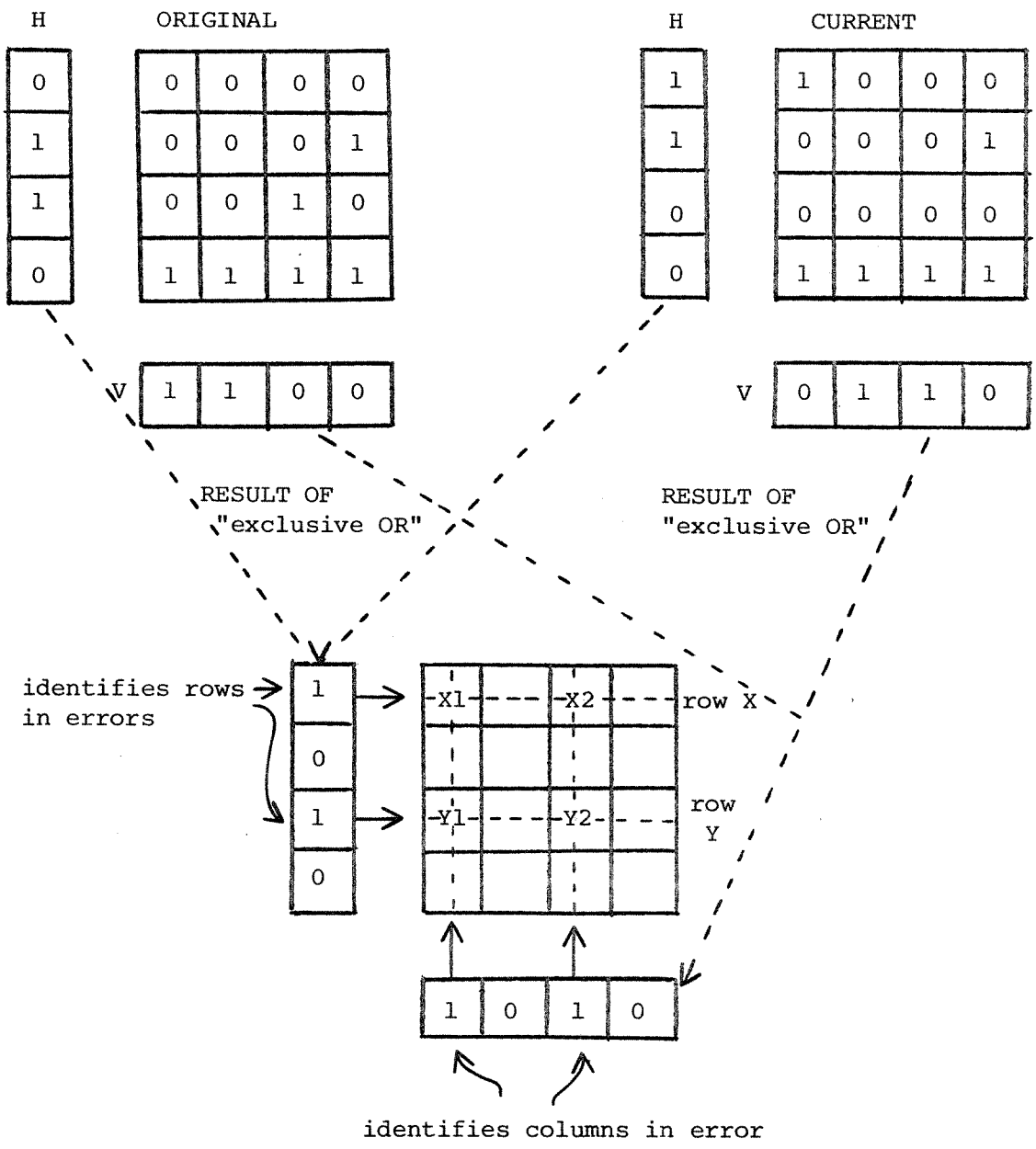


Figure 9: Ambiguity of Two Erroneous Bits Using Algorithm HV

Algorithm HVC

Algorithm HVC is designed to handle a majority of the cases where two bits are in error. A checksum is also kept for each nxm matrix. The way the old checksum compares with the new checksum indicates the nature of the changes which have occurred in the matrix. Consider what a checksum comparison would indicate when two bits have become erroneous in a matrix. This is outlined in Table 5.

RELATIONSHIP	CHECKSUM FORMULA	MEANING	PROBABILITY OF OCCURRENCE
checksum has decreased by two	$C_{NEW} = C_{OLD} - 2$	$\begin{Bmatrix} 1\to0 \\ 1\to0 \end{Bmatrix}$	1/4
checksum has increased by two	$C_{NEW} = C_{OLD} + 2$	$\begin{Bmatrix} 0\to1 \\ 0\to1 \end{Bmatrix}$	1/4
checksum remains constant	$C_{NEW} = C_{OLD}$	$\begin{Bmatrix} 0\to1 \\ 1\to0 \end{Bmatrix}$ or $\begin{Bmatrix} 1\to0 \\ 0\to1 \end{Bmatrix}$	1/2

Table 5: Checksum Comparisons

This knowledge of the nature of the change in conjunction with the present settings of the bits at the four questionable intersections allows accurate reconstruction in many cases. These possibilities are detailed in the tables included in Appendix A. Some combinations

cannot occur while other conditions are indeterminate and cannot be reconstructed properly. Determinate cases are shown with the suggested reconstruction.

Combining the results found in Appendix A yields a success rate of over 73 percent in correcting two errors occurring in different rows and columns of a matrix. Including the cases where the two bits do occur in the same row or column reduces the success rate only about 1 percent if there are 60 rows and 60 columns. Reconstruction of all two bit errors in one matrix would be successful over 72 percent of the time for Algorithm HVC using this tabular approach.

The possibility of three errors occurring in one matrix before a test is indeed rare. However, the theoretical possibility should be considered. For Algorithm HVC, an approach similar to the analysis of two errors suffers from a combinatorial explosion. Two errors involved four intersections and 48 cases. Three errors involves nine intersections and 3584 cases. It becomes readily apparent that the tabular probability scheme used by Algorithm HVC is unmanageable for considering more than two errors at one time.

Algorithm HVZ

A theoretically advanced solution to the whole problem of reconstruction would be to save a third dimension of parity along the Z-axis as well as saving vertical

and horizontal parity. Algorithm HVZ has the advantage of being intuitively obvious. Its technique can easily handle two or three errors in one level, with a very high rate of success. It fails only on extremely rare cases such as two bits being in the same row or column as well as in the same Z-axis parity with two other bits in other levels. This is sketched in Figure 10 where the four erroneous bits are marked with heavy dots.

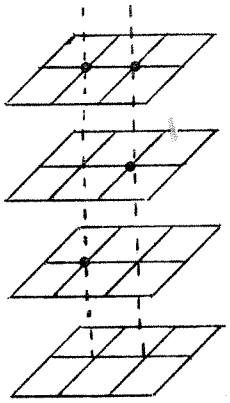


Figure 10: Ambiguity of Multiple Erroneous Bits Using Algorithm HVZ

Algorithm HVZ has a drawback which seems to make its use prohibitive if the Z-axis parity is taken on very many layers. Reconstruction of one matrix would involve a consideration of many matrices making up the Z-axis parity. The problem of initially developing and re-developing the Z-axis parity is quite complex on an actual computer. Assuming that each matrix would probably require

a unique input command, generating a Z-axis parity would be most expensive in terms of resource allocation and overall timing.

Discussion of Diagonal Parities in Algorithms

Let us consider diagonal parities to evaluate possible algorithms. Diagonal parities remain in a two dimensional plane where one matrix is a reconstruction entity. There are many algorithms that could be defined combining various diagonal parities with other types of data. Here this paper must be selective rather than complete.

At first glance, a diagonal without wraparound seems much less confusing than one with wraparound. However, this criticism fades with the realization that any diagonal occupies only one position on any one column (or row). Thus adding a vertical parity to an algorithm resolves any ambiguity. The net storage requirements are equal for (D1, D2) or (D, V) and much more information is available in (D, V). As an additional disadvantage, it is slower to compute (D1, D2) than D.

In defining an algorithm using D, either the right or left diagonal parity may be used. Arbitrarily, the diagrams in this paper show the left diagonal D_L. On certain computers, a savings in time to develop the parities would provide a criteria for selection. On the Control Data computer, equal time is required to compute

either parity. (Note a right shift of n would distribute the sign bit so instead code a left shift of 60-n.)

Algorithms VD and VH

Algorithm VD uses vertical parity and a diagonal parity with wraparound. Algorithm VD is motivated by a slight advantage over Algorithm VH specifically concerned with the instruction set on the Control Data Computer [17]. Each horizontal parity (H) may be computed with a "Count", "Mask", and "OR". Each diagonal parity (D) may be computed with a shift and an "exclusive OR". Thus the computation of the horizontal parity is slower by the time it takes to do a "Count". Either algorithm is completely deterministic on one bit errors. Also either always recognizes a multiple error and does not attempt any reconstruction since there is no additional information to resolve ambiguities.

Algorithm VDC

Algorithm VDC adds a checksum to the saved information of Algorithm VD. It is an improvement comparable to the expansion of Algorithm HVC from Algorithm HV. Algorithm VDC is slightly superior to Algorithm HVC because of the savings in computing diagonal parity D instead of horizontal parity H. While a one bit error is 100 percent correctable using either algorithm, a different set of failures occurs with two bit errors for an overall compar-

able rate of success. Both fail when the two errors are on the same column. Balancing out, Algorithm HVC fails when both errors lie on the same row while Algorithm VDC fails when both errors lie on the same diagonal.

For solvable two bit errors, Algorithm VDC must use a tabular approach as shown in Appendix A to yield an overall success rate of more than 72 percent for reconstruction of two bit errors. Diagrammed in Figure 11 is a case of two bits in error--not in the same diagonal or column. Label the four intersections as X1, X2 in diagonal X and Y1, Y2 in row Y. Either pair X1, Y2 contains the erroneous bits and the pair X2, Y1 is correct but identified due to symmetry or vice versa.

For three or more errors in one matrix, this tabular probability scheme suffers from a combinatorial explosion. Thus Algorithm VDC is just as incapable as Algorithm HVC in reconstructing more than two errors at a time.

Algorithm HVD

Algorithm HVD requires three parities to be saved: horizontal, vertical, and a diagonal with wrap-around. Detailed logic for this algorithm is presented in Appendix B. Actual COMPASS code to compute redundancy is presented in Appendix C with a timing analysis.

Algorithm HVD differs from the algorithms previously presented--it involves more complicated logic when

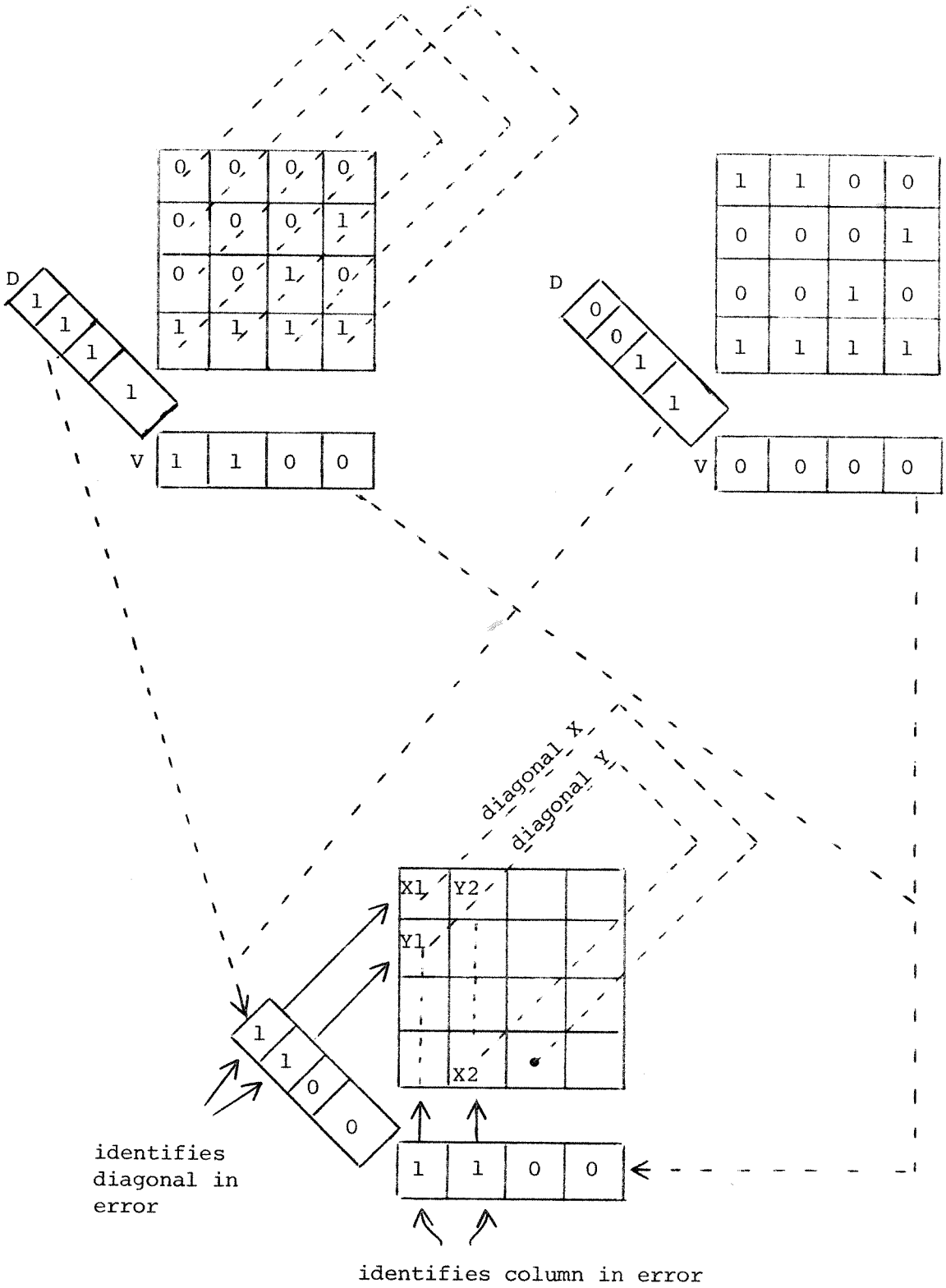


Figure 11: Ambiguity of Two Erroneous Bits Using Algorithm VDC

errors are encountered and as a result is more successful. Of course one bit errors are 100 percent correctable but, in addition, two bit errors are 100 percent correctable and three bit errors are over 85 percent correctable (assuming a 60x60 matrix). Even more of the remaining 15 percent of the three bit cases could be handled, but not easily. Therefore, a smaller matrix size is recommended if many 3 bit errors are expected in an nxm matrix.

Consider the most straightforward case involving two errors in a matrix where there is no common row, column, or diagonal. Figure 12 shows the reconstruction problem using the result of the "exclusive OR" between each original and corresponding current parity. Since there are three way intersections, ignore the intersection of two directions as seen in b, c, d, e, f, and g. There are only two positions that are intersections of three directions--a and h. The parities indicate there are exactly two errors, no two of which are in the same row, column, or diagonal. This means a and h are required solutions.

For a more complicated situation, consider a case where the two errors lie on a common direction, say in the same column. Therefore no column is identified as erroneous. No three way intersections exist so the two way intersections must be considered--a, b, c, and d. Because no erroneous column is identified, the two solutions can be deduced to lie in the same column. Of the four inter-

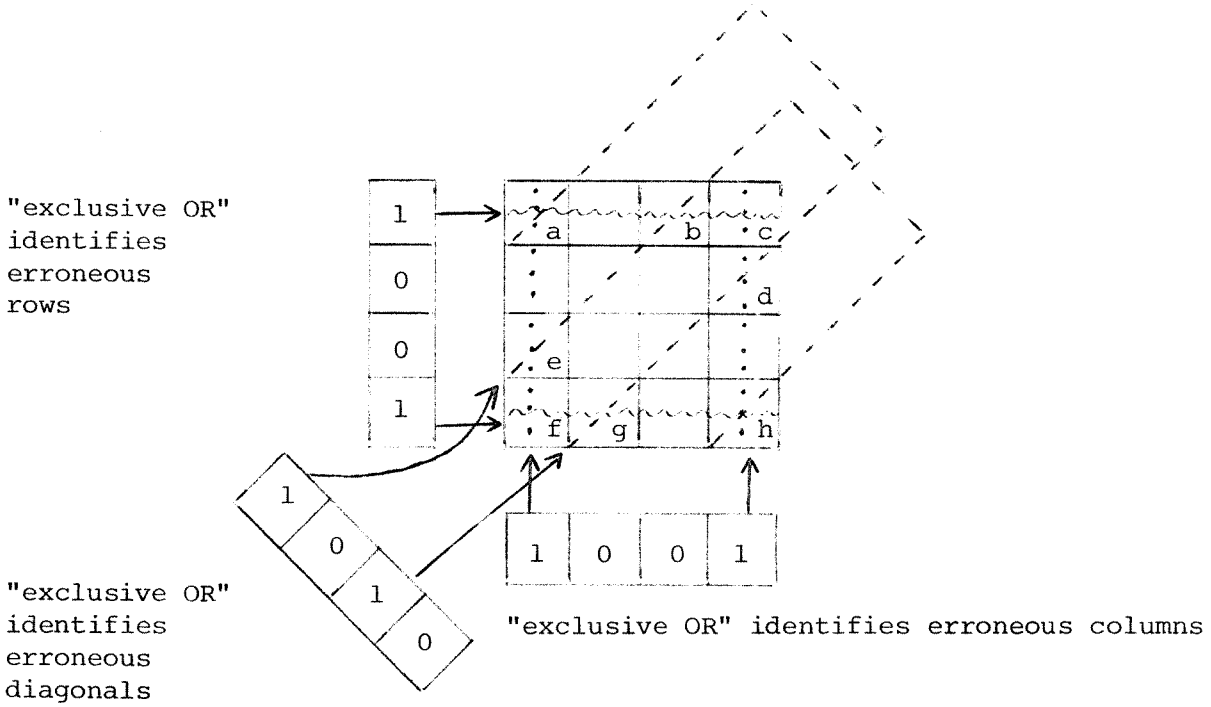


Figure 12: Error Correction Using Horizontal, Vertical, Diagonal Parities (Left-Diagonal with Wrap-around)

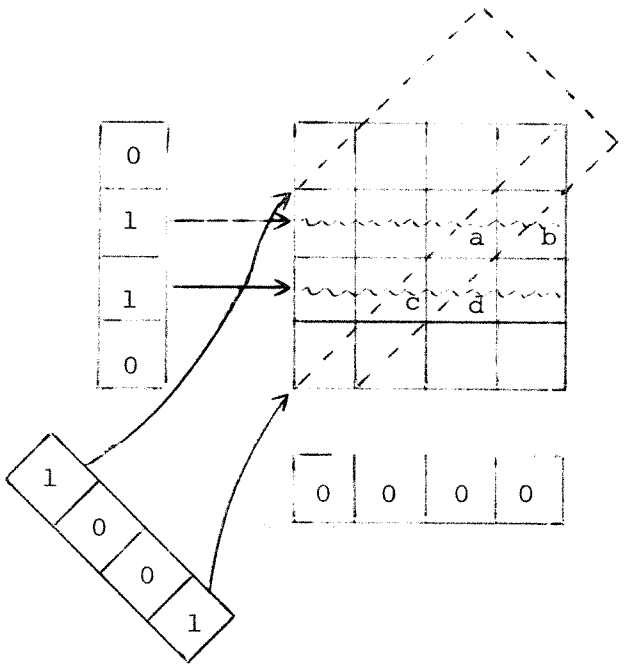


Figure 13: Ambiguity of Two Erroneous Bits Using Algorithm HVD

sections under consideration, only a and d satisfy the criteria of being in the same column. Thus by logic, an implicit condition was identified which allowed the problem to be solved.

The basic algorithm easily expands to handle three errors in one matrix when no errors have a common row, column, or diagonal. The basic approach corresponds to the case of two erroneous bits where there is no common direction. In either case, all three-way intersections mark an unquestionably erroneous bit without ambiguities. This logic is presented in algorithmic form in Appendix B.

It is not recommended that Algorithm HVD attempt to reconstruct any case where three errors have occurred with a shared direction, either the same row, column, or diagonal. But note that the ambiguity could be resolved by the use of a theorem prover given logical statements of implicit conditions.

SUMMARY OF ERROR CORRECTION ALGORITHMS

Table 6 presents a summary of the algorithms. Percentages are based on a 60x60 matrix (or on a 60x60x60 three dimensional array for Algorithm HVZ).

ALGORITHM	Saved Information		Success Percentage			Reasons for Failure (see notes below chart)
	#units	%increase	1 error	2 errors	3 errors	
HV	2	3-1/3	100	0	0	
HVC	3	5	100	72	0	A
VD or VH	2	3-1/3	100	0	0	
VDC	3	5	100	72	0	B
HVZ	3	5	100	99.9+	85	C
HVD	3	5	100	100	85	D

Table 6: Summary of Error Correction Algorithms

A - two errors in the same row or column

B - two errors in the same column or left diagonal (if D_L used in algorithm, otherwise D_R)

C - two errors in same row or column and also each error in same depth parity with another error on a different level

D - two or three errors in same row, column, or left diagonal (if D_L used in algorithm, otherwise D_R)

Algorithm	Comparison with other algorithms
HV	Slightly slower than VD
HVC	Slightly slower than VDC
VD	Best algorithm if only 1 bit errors are to be reconstructed. Will not handle two or more errors in one matrix.
VDC	A good algorithm for correcting all 1 bit and many two bit errors. Its tabular approach produces speed at the cost of storage. In all it is deemed inferior to Algorithm HVD.
HVZ	Computing Z-axis parity is prohibitive primarily because of input requirements.
HVD	Best algorithm found for reconstructing all one bit and two bit errors as well as most three bit errors. It is reasonably fast and rather low in storage requirements.

Table 7: Comparison of Error Correction Algorithms

CHAPTER V

A SAMPLE COST ANALYSIS

PARAMETERS INVOLVED IN A COST ANALYSIS

Cost analysis of error handling is concerned with the following parameters which describe a system:

1. volume of data in long term storage (bits)
2. storage media fault rate (errors/sec/bit)
3. transmission rate--volume of data per unit of time which is sent through the Input/Output device between central memory and long term storage (bits/sec)
4. I/O media fault rate (errors/bit)
5. transaction rate (transactions/sec)
6. software failure rate such that incorrect data is placed in long term storage (errors/transaction)
7. transaction processing cost (in resources)
8. error propagation rate for an undetected error
9. intangible costs of errors (such as a lost bomb)
10. time to perform a rollback and recovery
11. manual correction cost
12. reprocessing cost for a detected error
13. detection cost (in resources)

14. cost of implementation for error detection

15. error correction cost (in resources)

16. cost of implementation for error correction

Items 1-6 give the total rate of errors which can be predicted to occur in a system. These are the types of errors that change bits in the data base. The product of items 1 and 2 gives the number of errors per second which may occur in long term storage. Thus storage failures should increase with larger banks of storage. The product of items 3 and 4 gives the predicted number of errors per second which may occur while the data is being transmitted between central memory and long term storage (read or write). Thus transmission failures should increase with greater I/O activity. The product of items 5 and 6 gives the number of errors per second due to software failures. This is the rate which could occur during processing of information which alters the bits in the data base.

Items 8-10 cover the cost of errors which are not detected and subsequently corrected or insulated from the rest of the system. Items 11-14 describe the costs involved in detecting an error whether or not error correction will be incorporated into the system. Items 15-16 cover the cost of adding error correction to a system which already contains detection.

Chandy et al. [14] present some indices which may be useful in comparing different techniques for achieving

reliability in a specific system. These indices are called the Hardware Reliability Efficiency index (HRE), the Software Reliability Efficiency index (SRE), the Real-Time Criticality index (RTC) of a system, and the inclusion factor.

The cost of errors must be weighed against the cost of error detection/correction procedures to decide what degree of error processing to include in a system. This is inherently a management decision since several of the comparative costs are intangible. This was discussed in the previous section INCREASING RELIABILITY IN COMPUTERS. The central question is unreliability in a data base without DIM versus increased overhead to incorporate DIM.

An analysis of a specific system can provide information as to the degree and importance of the various cost factors to be considered. A broad overview of a procedure to follow in such an analysis is presented in the following sections. The types of costs and their relative sizes are discussed in a qualitative manner. A more complete quantitative analysis would require extensive data gathering in the system to be studied (i.e. the mean error rate should be determined) and cost estimates of intangible values (i.e. a lost nuclear device) need to be made. Such a complete cost analysis of a complex computer system is beyond the scope of this thesis.

THE COST OF ERRORS WITHOUT A DETECTION/ CORRECTION PROCEDURE

The cost of errors is determined by the frequency of errors and the average cost of each error. Hardware specifications should provide an upper bound on the frequency of errors but measurements of a system in operation are necessary for accurate results. Few statistics are available on the error rate for the NOLS set of hardware. This is a common problem for complex computer installations. The section on PROBABILITY OF ERRORS IN A DATA BASE provides some numbers for the purposes of estimation, but these numbers are only approximate.

The cost of a given error depends on where it occurs. If it is in an index or address, it could cause a chain of errors in processing. There is a good chance that the results will be obviously incorrect and thus a problem will be detected. This type of error has a high cost for recovery (rollback and restart) and an intangible cost due to increased maintenance time for the system. More insidious are those errors in the data base which are undetected. The data provides incorrect results but the errors may not be detected until a user finds a discrepancy between the data base and the real world (i.e. lost ordnance). The cost of this type of error in a sensitive data base such as that of NOLS could be high in intangible and tangible terms. On the other hand, the error could be in some unimportant area where it has no opportunity to

propagate. The error might be written over before it has a significant effect on the system. Then the cost is negligible. The relative percentage of each type of error should be estimated in order to find the average cost of a single error.

In addition to the cost of each error, there is an intangible cost due to undermined user confidence even when the data is error-free.

To summarize, a measure of the frequency of errors along with a breakdown of the relative frequency of the different types of errors should provide some guidelines for the cost of errors without a detection/correction procedure.

THE COST OF AN ERROR DETECTION/CORRECTION PROCEDURE

There are two types of cost in the proposed error handling procedures: storage and processing overhead. The algorithms described earlier in this paper require up to 5 percent additional storage to maintain the redundancy. This storage could be allocated within the current system if it has the capacity or an additional storage system could be added. The NOLS system does have enough available storage to hold the redundancy. Additional storage would be very expensive if this were not the case.

The processing overhead for the error detection/correction procedure can be split into two parts: parity computation time and storage retrieval time. The parity computation time is required every time a logical record

is written to or read from the disk system.

The storage retrieval time is the additional overhead on the DMS caused by the storage and retrieval of the parity information. Implementation of error handling has a choice of storing the redundancy with the data or in a separate area. Storage with the data would cause an increase in storage retrieval time of approximately 5 percent (algorithms require up to 5 percent additional storage for redundancy). Storage in a separate area is more expensive since additional I/O requests are necessary.

A NEW MODEL TO ANALYZE THE NOLS SYSTEM

The system chosen for analysis is NOLS (Nuclear Ordnance Logistic System), a subsystem of the ALS (Advanced Logistic System) supported by and for the Air Force [3]. This system appears to be a good candidate for DIM for two reasons. First, the cost of undetected errors in the data base can be very high. Second, the system is lightly loaded, making the cost of error correction low in terms of mean system response or turnaround time.

Model studies can provide both insight and quantitative analysis if pursued in depth. A model of the NOLS system should be studied to analyze the cost of an error detection/correction procedure. Logic flow and statistics were found in [3]. A diagram of the model is shown in Figure 14.

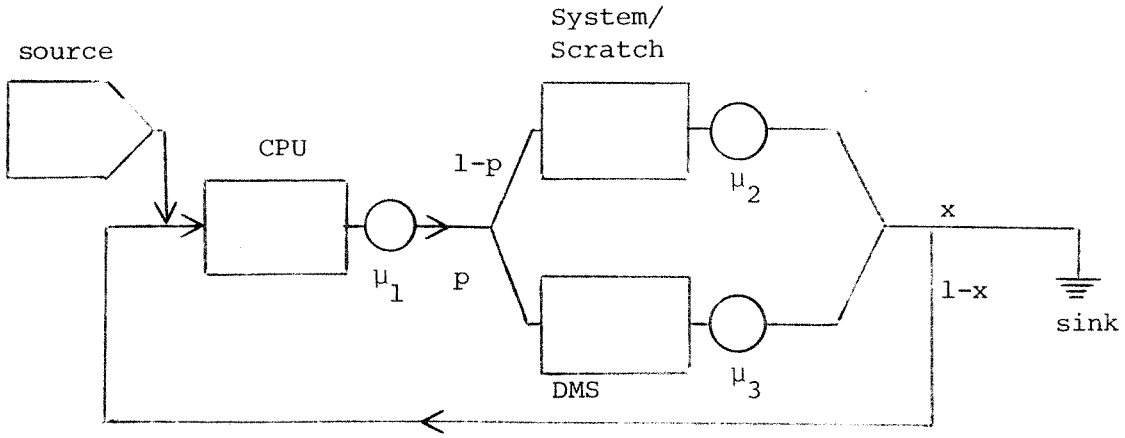


Figure 14: A Simplified Model of NOLS

This model describes the approximate sequence of events that occurs when a user of NOLS initiates a transaction. First, jobs arrive from the source (i.e. a remote job entry port) at a rate of λ and enter the CPU sub-system. This subsystem includes the central memory (CM) and extended core storage (ECS) areas as well as the central processing unit (CPU). It will be referred to as the CPU hereafter. Jobs are assumed to leave this subsystem at an exponential rate, μ_1. Thus, $1/\mu_1$ is the mean CPU burst time.

When a job leaves the CPU, it may either require service from the DMS subsystem (i.e. a request for information from the data base) or from the System/Scratch sub-

system (i.e. a library call or a read/write to a scratch area). The DMS subsystem will be referred to as the DMS and the System/Scratch subsystem will be referred to as the SYS. Jobs are assumed to require DMS service with a probability of p and SYS service with a probability of 1-p. Jobs are assumed to leave the SYS at a rate of μ_2 and the DMS at a rate of μ_3. Therefore, their respective mean burst times are $1/\mu_2$ and $1/\mu_3$.

When a job leaves either of the above subsystems, it may leave the system (if it has completed its transaction) or it may return to the CPU subsystem. In this model, a job will leave the system with a probability of x and return to the CPU with a probability of 1-x. For instance, if x = 1/100, then a job will cycle through the CPU and I/O subsystems an average of 100 times before leaving the system.

This model has omitted many details of the system to maintain clarity and to keep the analysis within manageable proportions. For example, the limitation which ECS places on the number of active chains in the system at a given time is ignored. While this type of model does not give fine resolution, it does yield measurements of device utilization, system throughput and response time with validity bounds of less than 10 percent for a large number of computer systems, including NOLS. See Baskett [15] for a discussion of modeling techniques. Note also that

when the model is used in a comparison with itself, the errors inherent in the model will tend to cancel out.

For the purposes of the cost analysis, "errors" refers only to those errors which would be detected by the proposed detection procedure. This restriction is reasonable because the cost of the other types of errors would not be changed whether or not the proposed error handling procedure is implemented.

Estimates for p, x, μ_1, μ_2, μ_3, and λ have been made by comparing NOLS and ALS configurations and by assuming that the characteristics of programs on NOLS correspond to those programs used as benchmarks in [3]. Modifications and allowances have been made for the differences between the two systems. Since an in-depth analysis of NOLS is beyond the scope of this thesis, these estimates will be used to provide a basis for a discussion of cost analysis.

The following statistics are used in the model. A transaction consists on the average of 100 cycles through the system (or 100 CPU bursts). Thus x=0.01. On the average 1 out of 5 CPU bursts cuases a DMS request. This sets p to 0.2.

In the system without DIM, the mean CPU burst ($1/\mu_1$) is 20 ms. and the mean burst times for the SYS and DMS subsystems ($1/\mu_2$, $1/\mu_3$) are each 40 ms. (seek and transfer time). In the system with DIM, these values will increase due to the overhead of the detection procedure as

outlined below.

The cost of error correction relative to its freqeuncy of occurrence can be neglected. It occurs less than once in a million times and takes much less than 1 second to process. Therefore the overhead is very small. This model is primarily concerned with detection costs. The cost of implementation is the major cost of an error correction package even using the slowest algorithm.

This model will use an estimate of a 5 percent increase in the mean CPU burst time ($1/\mu_1$). The actual overhead is expected to be somewhat less than that amount. This is based on the following reasoning. One milliseconds is an upper bound on the time required to create or check parity information on a physical record of 512 words (see Appendix C). With four physical records to one logical record, it takes 4 ms. extra to process a DMS transaction. On the average, 1 out of 5 CPU bursts causes a DMS request. Therefore, the mean CPU burst will take an additional 0.8 ms. The figure 1 ms. is used to allow for hidden costs such as increased swapping caused by less usable core. Thus, the average CPU burst is 21 ms.

Note that no change occurs in $1/\mu_2$ since the SYS subsystem is unaffected by the error detection procedure.

This model assumes that NOLS will implement separate storage to minimize impact on the operational system. In NOLS, the average logical record is contained in four

physical records. Redundancy for all of these could be stored in a single additional physical record. This would mean an increase in storage retrieval of approximately 25 percent. Thus, $1/\mu_3$ will increase to 50 ms.

Two cases are examined: a lightly loaded system and a heavily loaded system. The lightly loaded system has a transaction start on the average of once every 10 seconds. The heavily loaded system has a transaction start once every 4 seconds. This case is based on the possibility of increased usage in the future.

The queue lengths, utilizations, and response times of the various subsystems and the mean system response time were computed by standard analytical techniques found in Drake [16]. The results of these computations are shown in Table 8.

The cost of the error detection/correction procedure in terms of increased system response time varies from 3 percent to 5 percent depending on the load that the system is carrying. The relative undesirability of this cost depends on the system. Degradation of response time should not be significant on the NOLS system. This is due to the manageable load on NOLS and the primary concern with long running events.

Another measure of cost is potential throughput of the system. The System/Scratch Disk subsystem is the bottleneck in this model whether or not error handling is

System Parameters	Lightly Loaded		Heavily Loaded	
	Without DIM	With DIM	Without DIM	With DIM
Mean CPU burst ($1/\mu_1$)	20ms	21ms	20ms	21ms
Mean SYS burst ($1/\mu_2$)	40ms	40ms	40ms	40ms
Mean DMS burst ($1/\mu_3$)	40ms	50ms	40ms	50ms
Mean time between arrivals	10sec	10sec	4sec	4sec
CPU utilization	.200	.210	.500	.525
SYS utilization	.320	.320	.800	.800
DMS utilization	.080	.100	.200	.250
Mean CPU queue length	.250	.266	1.00	1.105
Mean SYS queue length	.471	.471	4.00	4.00
Mean DMS queue length	.087	.111	.250	.333
Mean CPU response time	25.0ms	26.6ms	40.0ms	43.2ms
Mean SYS response time	58.8ms	58.8ms	200ms	200ms
Mean DMS response time	43.5ms	55.5ms	50ms	66.6ms
System response time	8.1sec	8.5sec	21sec	22.2sec
Percent increase in response time	5%		3%	

Table 8: Results of the NOLS Model Analysis

included. Error processing does not degrade the performance of that subsystem, based on the model of NOLS. The error detection/correction procedure does not decrease the potential throughput of the system. Therefore, this cost is zero for this model.

It must be recognized that the assumptions made in preparing these estimates are subject to error. For example, the system/scratch subsystem is assumed to be the major bottleneck in NOLS as it was in ALS [3]. If this assumption is not valid, the results given here are subject to some error. The estimates of changes in system response time would increase, and the estimates of maximum possible throughput would decrease. These changes are not critical so long as the actual system throughput remains significantly below system capabilities.

SUMMARY OF THE COST ANALYSIS

It is desirable to implement an error detection/correction procedure for NOLS. The cost of errors in the data base can be high. An error detection/correction procedure can virtually eliminate these costs at only a slight cost in terms of system performance and resources. Fewer errors mean better overall system performance with less time spent with manual corrections and Rollback/Restart/Recovery.

The recommendation to implement error handling in NOLS is partially based on the model in Figure 14.

Changes to this model could influence a final decision.

CHAPTER VI

SUMMARY

This paper shows that error detection and correction are feasible software schemes that offer many advantages for a data base system. Errors are detected and removed from the normal system flow before they may jeopardize further processing. The errors are corrected immediately by software routines if the error is within the design capabilities of the algorithm selected.

Many error detection and correction algorithms are presented which use redundancy composed of combinations of horizontal, vertical, and/or diagonal parities. Algorithm HVD is recommended as the most comprehensive of all those presented. Detection would catch almost all errors using such an amount of redundancy. The error correction algorithms are designed to handle three or less erroneous bits in one rectangular array of data. This covers a significant portion of the errors expected in a data base. Backup measures including other Data Integrity Management capabilities are discussed.

The costs of including error detection and correction in a data base system are discussed. Storage requirements would increase an estimated 5 percent to retain the redundancy. If the redundancy is stored with the data

it describes, the Input/Output time would increase by less than 5 percent with no additional I/O commands. If the redundancy is stored separate from the data, then an increase in storage retrieval of 25 percent is estimated. System response time would increase about 5 percent. These figures are estimated to cover any data base system although they were partially derived from a cost analysis of the Advanced Logistics System (ALS) of the Air Force Logistics Command.

APPENDIX A

APPENDIX A

RESOLVING AMBIGUITIES FOR ALGORITHMS HVC OR VDC

CHECKSUM INDICATES TWO BITS ERRONEOUSLY TURNED OFF in an nxn matrix								
CURRENT SETTING				MEANING*	RECONSTRUCTION NEEDED			
X1	Y2	Y1	X2		X1	Y2	Y1	X2
0	0	0	0	?				
0	0	0	1	X	1	1		
0	0	1	0	X	1	1		
0	0	1	1	X	1	1		
0	1	0	0	X			1	1
0	1	0	1	-				
0	1	1	0	-				
0	1	1	1	-				
1	0	0	0	X			1	1
1	0	0	1	-				
1	0	1	0	-				
1	0	1	1	-				
1	1	0	0	X			1	1
1	1	0	1	-				
1	1	1	0	-				
1	1	1	1					

*determinate (X), indeterminate (?), cannot occur (-)

Table 9: Algorithm HVC or VDC Logic When Two Bits Turned Off

Summary: of the 16 combinations, only 7 are possible for this case. Of these 7, 6 can be reconstructed deterministically yielding a success rate of 85-5/7 percent. In the remaining case, the situation is ambiguous and correction is not possible.

CHECKSUM INDICATES TWO BITS ERRONEOUSLY TURNED ON in an nxn matrix									
CURRENT SETTING				MEANING*	RECONSTRUCTION NEEDED				
X1	Y2	Y1	X2		X1	Y2	Y1	X2	
0	0	0	0	-					
0	0	0	1	-					
0	0	1	0	-					
0	0	1	1	X			0	0	
0	1	0	0	-					
0	1	0	1	-					
0	1	1	0	-					
0	1	1	1	X			0	0	
1	0	0	0	-					
1	0	0	1	-					
1	0	1	0	-					
1	0	1	1	X			0	0	
1	1	0	0	X	0	0			
1	1	0	1	X	0	0			
1	1	1	0	X	0	0			
1	1	1	1	?					
*determinate(X), indeterminate(?), cannot occur(-)									

Table 10: Algorithm HVC or VDC Logic When Two Bits Turned On

Summary: The results of this case exactly correspond to the previous results for two bits erroneously turned off. Of the 16 combinations, only 7 are possible for this case. Of these 7, 6 can be reconstructed deterministically, yielding a success rate of 85-5/7 percent. In the remaining case, the situation is ambiguous and correction is not possible.

CHECKSUM INDICATES ONE BIT TURNED OFF AND ANOTHER TURNED ON ERRONEOUSLY in an nxn matrix									
CURRENT SETTING				MEANING*	RECONSTRUCTION NEEDED				
X1	Y2	Y1	X2		X1	Y2	Y1	X2	
0	0	0	0	-					
0	0	0	1	X			1	0	
0	0	1	0	X			0	1	
0	0	1	1	-					
0	1	0	0	X	1	0			
0	1	0	1	?					
0	1	1	0	?					
0	1	1	1	X	1	0			
1	0	0	0	X	0	1			
1	0	0	1	?					
1	0	1	0	?					
1	0	1	1	X	0	1			
1	1	0	0	-					
1	1	0	1	X			1	0	
1	1	1	0	X			0	1	
1	1	1	1	-					
*determinate(X), indeterminate(?), cannot occur(-)									

Table 11: Algorithm HVC or VDC Logic When Two Bits Reversed

Summary: of the 16 combinations, only 12 are possible for this case. Of these 12, 8 can be reconstructed deterministically yielding a success rate of 66-2/3 percent. In the other 4 cases, the situation is ambiguous and correction is not possible.

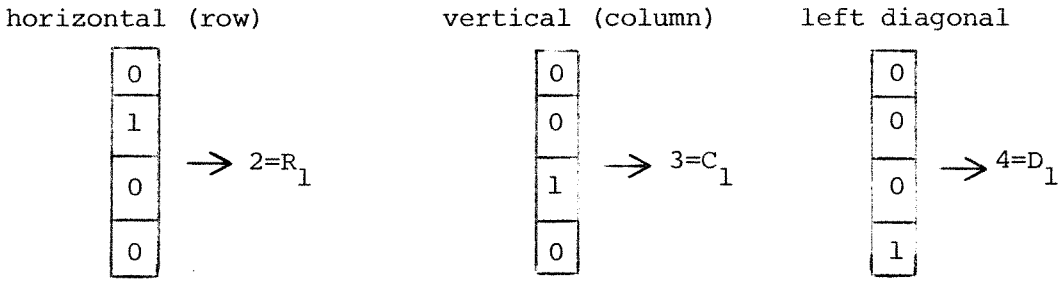
APPENDIX B

APPENDIX B

ALGORITHM HVD IN DETAILED LOGIC (using an nxn matrix)

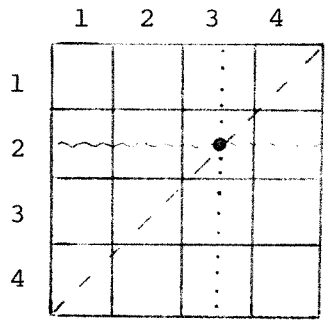
1. Convert each "XOR" information to numbers corresponding to coordinates of changed bits - R_i, C_j, D_k for i = 1,n; j = 1,n; k=1,n.

   Example:



2. Count the number of 1's in each "XOR" information.
3. If each "XOR" has a count equal to 1 then the ordered 3-tuple (R, C, D) is the location of the erroneous bit, go to step 8.

   Example: (2, 3, 4) is the location in the example of step 1.



4. Else if each "XOR" has a count equal to 2 then combine the two values of R and C to form 4 unique 2-tuples. (R_1, C_1) and (R_1, C_2), (R_2, C_1) and (R_2, C_2) and save D_1 and D_2 separately. Compute D for each 2-tuple using the formula

$$D = (R + C-1) \text{ if } R + C-1 \leq n$$

$$D = (R + C-1-n) \text{ if } R + C-1 > n$$

Compare each D with the two accepted values D_1 and D_2. If D equals D_1 or D_2 then the ordered 3-tuple (R_i, C_j, D_k) $i = 1,2$; $j = 1,2$; $k = 1,2$; should be saved as the position of an erroneous bit. There should be exactly two 3-tuples satisfying this test. Go to step 8.

5. Else if no "XOR" has a count > 2 then take the numbers corresponding to the two "XOR"'s with a count = 2 and use them in the following formulas to compute numbers corresponding to the missing coordinates. Arrange each (R_i, C_j, D_k) in an ordered 3-tuple (a total of 4 sets).

Given R and C:

$$D = (R + C - 1) \quad \text{if } R + C - 1 \leq n$$

$$D = (R + C - 1 - n) \quad \text{if } R + C - 1 > n$$

Given R and D:

$$C = (D + n + 1 - R) \quad \text{if } D < R$$

$$C = (D + 1 - R) \quad \text{if } D \geq R$$

Given C and D:

$$R = (D + n + 1 - C) \quad \text{if } D < C$$

$$R = (D + 1 - C) \quad \text{if } D \geq C$$

Identify the two 3-tuples with the same value just computed. These are the positions of the erroneous bits. Go to step 8.

6. If each "XOR" has a count = 3 then (paralleling step 4) combine the three values of R and C to form 9 unique 2-tuples. Save D_1, D_2, and D_3 separately. Compute a D for each 2-tuple using the formula in step 4. Compare each D with the three accepted values D_1, D_2, D_3. If D equals one of these, then the ordered 3-tuple (R_i, C_j, D_k) i = 1,2,3; j = 1,2,3; k = 1,2,3; should be saved as a position of an erroneous bit. There should be exactly three 3-tuples satisfying this test. Go to step 8.
7. Else exit with a failure indication.
8. Use each 3-tuple to flip the bit at the identified position:
   a. generate a mask of 00...010...000 where the 1 appears in the identified column C_j
   b. "XOR" this mask into the row identified by R_i.
9. Exit with a success indicator.

APPENDIX C

APPENDIX C

COMPASS SUBROUTINE TO COMPUTE REDUNDANCY FOR ALGORITHM HVD[1]

```
                **      CRHVD - COMPUTE REDUNDANCY FOR HVD
                *               ALGORITHM
                *
                *       ENTRY:  (B1) = FIRST WORD ADDRESS OF
                *                      PACKET
                *               (B2) = LAST WORD ADDRESS OF
                *                      PACKET
                *       EXIT:   (X5) = H - PARITY
                *               (X6) = V - PARITY
                *               (X7) = D - PARITY
                *
                *       NOTE:   PACKET SIZE MUST BE LESS THAN OR
                *               EQUAL TO 60 WORDS
                *
                CRHVD   RJ      *+400000B  ENTRY/EXIT WORD
                        SA1     B1
                        SB3     1
                        SB4     2          PREPARE TO
                        SX5     B0
                        SX6     B0           ENTER THE
                        SX7     B0
                        CX3     X1             LOOP
                        SB2     B2-B3
                        SA2     B1+B3
                        MX0     1
                        LX0     1
TIMING          *
ESTIMATE        *
FOR LOOP        *       LOOP BEGINS HERE
START END       *
                *
  0     2       CRHVD1  LX7     1          COMPUTE PARITIES
  1     3               BX6     X6-X1
  3     5               LX5     1
  4     6               BX7     X7-X1
  7     9               BX3     X3*X0
  8    15               CX4     X2
  9    11,16            SA1     A1+B4      FETCH NEXT WORD
 10    12               BX5     X5-X3
 11    13               LX7     1          COMPUTE PARITIES
 12    14               BX6     X6-X2
 14    16               LX5     1
```

---

[1]COMPASS is a assembly programming language [17].

```
START END

 15   17            BX7     X7-X2
 18   20            BX4     X4*X0
 19   26            CX3     X1
 20   22,27         SA2     A2+B4       FETCH NEXT WORD
 21   23            BX5     X5-X4
 22   29            LT      A1,B2,CRHVD1
              *
              *      END OF LOOP
              *
                    EQ      A1,B2,CRHVD2
                    SA1     B2
                    CX4     X2          COMPUTE PARITIES FOR
                    LX7     1           THE FINAL WORD IN THE
                    BX7     X7-X1       BLOCK IF NECESSARY
                    BX6     X6-X2
                    LX5     1
                    BX3     X3*X0
                    BX5     X5-X3
              *
              CRHVD2 SB4    B1+58       ADJUST POSITIONS OF
                    SB4     B4-B2       PARITIES IF NECESSARY
                    EQ      B4,B0,CRHVD
                    LX5     B4,X5
                    LX7     B4,X7
                    EQ      B0,B0,CRHVD
```

Every execution of the loop takes approximately 30 minor cycles. Therefore a packet of 60 words would take approximately 900 minor cycles to complete the loop. Allowing 100 minor cycles for the rest of the subroutine yields a total of 1000 minor cycles or 100 microseconds to compute the HVD parities for a packet. Therefore, a 512 word physical block consisting of 9 packets would require under 900 microseconds.

The above estimates assume that memory bank interference will be negligible. However, even if every other memory access encountered a bank conflict, the total computation time would not increase by more than 100 microseconds. Therefore, the redundancy for a 512 word block can be computed

in less than 1 millisecond.

BIBLIOGRAPHY

1. P. Elias, "Error-Free Coding," IRE Transactions on Information Theory, PGIT - 4, Sept., 1954, pp. 29-37.

2. K.M. Chandy and C.V. Ramamoorthy, "Rollback and Recovery Strategies for Computer Programs," IEEE Transactions on Computers, Volume C-21, Number 6, June, 1972, pp. 546-556.

3. Limiting Capability Analysis of the CYBER 70 ALS, Information Research Associates, 2200 San Antonio, Austin, TX 78705, Sept. 4, 1973.

4. R.W. Hamming, "Error Detecting and Error Correcting Codes," Bell System Tech. J., Vol. 29, April, 1950, pp. 147-160.

5. James Martin, Security, Accuracy, and Privacy in Computer Systems, Prentice-Hall, Inc., N.J., 1973.

6. C.V. Ramamoorthy, "Fault-Tolerant Computing: An Introduction and an Overview," IEEE Transactions on Computers, Vol. C-20, No. 11, November, 1971, pp. 1241-1244.

7. A.M. Patel and M.Y. Hsiao, "An Adaptive Error Correction Scheme for Computer Memory System," AFIPS Conference Proceedings, Vol. 41, Part I, 1972, pp. 83-87.

8. William Wesley Peterson, Error-Correcting Codes, M.I.T. Press and John Wiley and Sons, Inc., N.Y., London, 1961, p. 81.

9. C. William Gear, Computer Organization and Programming, McGraw-Hill Book Co., N.Y., pp. 64-65.

10. James Martin, Programming Real-Time Computer Systems, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1965, pp. 125-133.

11. A. Jelinski and P. Moranda, "Software Reliability Research," Statistical Computer Performance Evaluation edited by Walter Freiberger, Academic Press, N.Y. and London, 1972, pp. 465-484.

12. M.E. Senko, E.B. Altman, M.M. Astrahan, and P.L. Fehder, "Data Structures and Accessing in Data-Base Systems," IBM Systems Journal, Vol. 12, No. 1, 1973, p. 32.

13. ALS Application Programming and Development Guide, AFLCM 171-286, Change 3, 7 March 1974.

14. K.M. Chandy, C.V. Ramamoorthy and A. Cowan, "A Framework for Hardware - Software Tradeoffs in the Design of Fault-Tolerant Computers," Proc. FJCC 1972 AFIPS pp. 55-63.

15. Forest Baskett, III, Mathematical Models of Multiprogrammed Computer Systems, TSN-17, Univ. of Texas at Austin, Computation Center, Jan. 1971.

16. Alvin W. Drake, Fundamentals of Applied Probability Theory, McGraw-Hill Book Co., New York, 1967, pp. 188-190.

17. COMPASS Reference Manual, Control Data 6400/6500/6600 Computer Systems.