CORRECTING FAULTS IN WRITE-ONCE MEMORY†

Danny Dolev *Hebrew University*
David Maier *Oregon Graduate Center*
Harry Mairson‡ *INRIA*
Jeffrey Ullman *Stanford University*

We consider codes for write-once memory in the presence of stuck-at-0 and stuck-at-1 faults. Such fault-tolerant codes generally require less redundancy than error-correcting codes, as faults detected during the writing process can affect subsequent behavior of that process. We present *pointer codes*, which use part of a codeword to point to faults in other parts of the codeword. A pointer code can encode n-bit messages in the presence of f faults with only $f(\log_2 n + o(1))$ redundancy. We derive a lower bound on the redundancy of such a fault-tolerant code of slightly less than $f \log n$. We also examine some models where all stuck-at information is known in advance, and analyze the expected redundancy of pointer codes.

I. INTRODUCTION

Write-once memory (wom) is a storage medium of two-state cells in which a 0 can be changed to a 1, but a 1 cannot be changed back to a 0. Standard examples of such memory devices include paper tape and punch cards, where holes can be punched, but not unpunched. The emergence of high-density and cheap write-once devices such as digital optical disks (where a 12-inch disk can store over 10^{11} bits of data) has provoked a more serious examination of how information should be encoded in such memories.

We investigate here the problem of encoding data in a write-once memory where cells in the memory may be defective, and consider what kind of redundancy must be included so that these faults can be corrected. A defective cell may be *stuck-at-1*, i.e., already changed to 1, or *stuck-at-0*, where attempts to change the cell from 0 to 1 fail. These "stuck-at" faults correspond to flaws in the fabrication of write-once memories such as optical disks, where the metal film used as the storage medium may be of uneven thickness.

Standard error-correcting schemes such as Hamming codes can indeed be used to correct faults in a write-once memory, but under certain circumstances we can do better. In particular, we examine the case where the number of stuck bits is fixed or slowly increasing with the size of the encoded message, rather than assuming that bits will be stuck with some fixed probability. Furthermore, standard error-correcting schemes assume an encoding and decoding model where the encoder does not ever find out what bits are in error: it is forced to compute the redundancy before the message is transmitted and subjected to noise. With a write-once memory, it is entirely appropriate to let the encoder know what bits are stuck either before or during encoding, so that the encoding algorithm may dynamically adapt to these faults.

In Section III we present *pointer codes*, which we use to encode n-bit messages in the presence of f faults with only $f(\log_2 n + o(1))$ redundancy. A lower bound proof in Section IV shows that this encoding scheme is essentially the best possible. In Section V we show how the upper bound can be improved if the encoder knows all the stuck-at information in advance, and in Section VI we examine the expected redundancy of the pointer code algorithm.

II. PRELIMINARIES AND MODEL OF COMPUTATION

We imagine an *encoder* that wants to send n bits of information to a *decoder*. The medium of the message (take that, Marshall McLuhan) is $b(n) \geq n$ bits of write-once memory. The decoder must be able to recover the original n-bit message by examining the $b(n)$-bit transmission, but is not allowed to change any of the bits, and has no side information about which bits are stuck. The encoder is allowed to examine the $b(n)$ bits *before* beginning the encoding algorithm to see which bits are stuck-at-1; since a "brand new" memory should have bits all set to zero, the stuck-at-1 bits are immediately visible.

An *encoding algorithm* consists of a set of decision trees, one for each message and pattern of stuck-at-1 bits, as suggested by Fig. 1. To encode message m, the encoding algorithm begins by attempting to write a 1 in bit position b_1, $1 \leq b_1 \leq b(n)$. The memory responds "stuck-at-0" or "OK," depending on whether or not the write was successful, and the encoding algorithm branches accordingly. Note that the encoding algorithm is characterized by where it writes 1 bits, as "write 0" means nothing in a write-once memory; also note that the encoding algorithm can be fully adaptive to faults. The encoder knows stuck-at-1 information in advance, so that it need not ever try to write a 1 in a

† Work partially supported by AFOSR grant 80-0212.
‡ Work performed while at Stanford Univ.

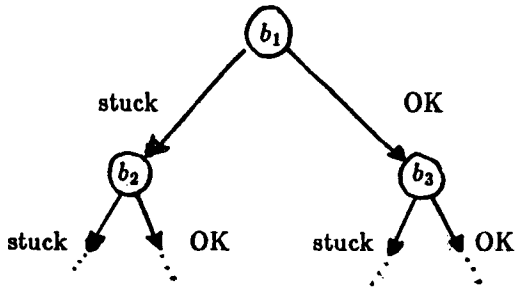

Fig. 1. An encoding algorithm.

bit already stuck-at-1, but only discovers that a bit is stuck-at-0 when it tries to change it from a 0 to a 1.

While the encoding algorithms we describe will not be stated explicitly in terms of decision trees, it will be clear that they can be so translated. The decision tree model is used in deriving the lower bounds of Section IV.

Codes to tolerate stuck-at faults have been studied for rewritable memory [HE, KT]. Lower bounds on those codes are lower bounds on fault-tolerant codes for write-once memory. Other work on codes for write-once memory has concentrated in its use as memory with a limited number of rewrites [M, RS].

III. POINTER CODES

We begin with the basic idea, and then make some refinements. To encode an n-bit message in $b(n)$ bits, divide the b bits of write-once memory into $f+1$ blocks $0, 1, \ldots, f$. Block 0, the *initial block*, consists of n bits and is used to hold the binary representation of messages. Blocks $1, 2, \ldots, f$ are *correction blocks* that hold pointers to indicate faults. Bit positions are numbered left to right starting at 1. A pointer of all 0's means no correction is indicated. While our model allows bit-at-a-time writes, the pointer code algorithm writes an entire block at once.

Let bin(i) be the binary representation of integer i. The writing procedure for a pointer code attempts to write a codeword for message m by first writing bin(m) in the initial block; if there are faults in the initial block, correction blocks are written to point to the faults.

Example 1: Let $n = 15$ and $f = 3$. Let the correction blocks have 4 bits each. Suppose bit 5 is stuck-at-1 and bit 12 is stuck-at-0, and we wish to write 0 and 1 there, respectively. Then we can use two correction blocks to point to the faults:

11110 11100 01000 0101 1100 xxxx

□

There are two problems with the basic idea. There may be faults in the correction blocks. We solve this problem by letting correction block i point to a fault in any of the blocks $0, \ldots, i-1$. The other problem is that unused correction blocks may contain stuck-at-1 faults that cause them to point to positions where no faults exist. There may not be enough correction blocks to indicate these faults. In the last example, if the last block contains 0100, it will incorrectly point to a fault, and there are no correction blocks to fix this error. We therefore let a correction block of all 1's also mean no correction, and indicate that all blocks to the right are to be ignored.

To decode an encoded word c, we let k denote the smallest nonnegative integer such that block k is all 1's; if there is no such block, let $k = f + 1$. Then for $i = k-1, k-2, \ldots, 1$, we complement the bit position in c to which block i points. (If block i is all 0's or all 1's, no correction is made.) Block 0 is then returned as the message.

Example 2: Suppose $b = 35$, $n = 20$, and $f = 3$. Correction blocks have 5 bits. To decode

01000 00000 00001 01100 00010 11001 11111

we first disregard correction block 3 (if there were blocks to the right of block 3, they would also be ignored.) Block 2 points to position 25, so we complement the bit there to get

00100 00000 00001 01100 00011 11001 11111

Block 1 now points to position 3, so we complement that bit, giving

00000 00000 00001 01100 00011 11001 11111

The decoded message can now be read from block 0 as 44. □

We omit here a more detailed description of the encoding algorithm and a formal proof of its correctness, and just give an informal correctness proof.

Theorem 1: Pointer codes with f correction blocks exist which can always correct up to f faults.

Proof: Use induction of f. The basis $f = 0$ is obvious. For $f > 0$, if there is one fault in the final block f, we can set it to all 0's or all 1's (depending on the parity of the stuck-at fault), and apply induction. If block f has j faults, $j \geq 2$, we ignore block f, and note by induction that $f-1$ correction blocks can correct $f-j$ faults, while also providing a block of all 1's to ensure block f is ignored. If block f is fault-free, by induction $f-1$ faults can be corrected in the first $f-1$ correction blocks, and block f can be used to fix the final remaining fault. □

Encoding

We now sketch the encoding algorithm. First the encoder writes bin(m) in block 0. Let P denote the bit positions of faults in block 0 which must be corrected. (Note not all faults must be corrected: a bit could be stuck in the "right" direction.) Next, the encoder removes the smallest p in P from P, writes p in the leftmost unwritten correction block, and adds to P the bit positions in that block where faults occurred. This removal, writing, and addition of faults continues until

P is empty. It is clear that when P is empty, no more than f correction blocks have been written.

Suppose j correction blocks were written. If $j = f$ the encoder is done, otherwise blocks $j+1, \ldots, f$ must be *nullified* so they do not point to nonexistent faults, and mess up the decoding. Here things get a little tricky, as the nullification must be done with some delicacy. The encoder searches blocks $j+1, j+2, \ldots, f$ until it finds a block i that has a stuck-at-1 fault, and tries to write all 1's in this block. If successful, the encoding is finished, since this block will indicate that blocks to the right are ignored. If unsuccessful, the encoder tries to correct block i with correction blocks $i+1, i+2, \ldots, f$ just as block 0 was corrected earlier.

However, correcting block i, unlike block 0, will not succeed if there are more than $f-i$ faults to be corrected in blocks $i, i+1, \ldots, f$ (one of them, the stuck-at-1 fault in block i, doesn't need to be corrected). But if this is the case, one of the blocks $j+1, \ldots, i-1$ must be fault-free: the encoder then tries to write all 1's in blocks $i-1, i-2, \ldots, j+1$ until it succeeds, and is guaranteed to do so. Of course, correcting block i may succeed, in which case unused blocks to the right are nullified again. A careful inductive argument shows this encoding scheme is correct.

The length $b(n)$ of the encoding clearly satisfies the inequality $b(n) \leq n + \log_2 b(n)$. If can be shown when $f = o(n/\log n)$ that $\log_2 b(n) \leq f + \log_2 n$, so that the redundancy is $f(\log_2 n + o(1))$.

A variant encoding scheme works when there are asymptotically more faults. Let each correction block have a *cancel bit*, which is set to 1 when the correction written in the block has a fault. A correction block with 1 fault, or one where the cancel bit is stuck-at-1, can then cancel itself, as the cancel bit says "I'm lying." Furthermore, correction blocks need only point to positions in block 0, or cancel bits of other blocks, so that a correction block is no more then $1 + \log_2(n+f)$ bits long; when $f = o(n)$ the total length of the encoding is than $n + f(\log_2 n + 1 + o(1))$. This scheme is smaller than the one above when $f \geq n/(\log_2 n - 2)$. The decoding algorithm for this scheme, which we omit here, is not hard.

IV. A LOWER BOUND

We now derive a lower bound on the redundancy necessary to correct up to f faults. Following our intuition that stuck-at-0 faults are harder to correct than stuck-at-1 faults, we prove a lower bound on the redundancy needed to correct *exactly* f stuck-at-0 faults.

Assume we are given an encoding algorithm as described in Section II, i.e., a set of decision trees, one for each message. Given enough time we could try encoding all possible n-bit messages under all possible error patterns, and we could fill in the table suggested by Fig. 2.

	Error Patterns			
Messages	E_1	E_2	$\cdots$	$E_{\binom{b}{f}}$
0				
1				
$\vdots$				
$2^n - 1$				

Fig. 2. Table of error patterns.

The entry in row m, column j gives the algorithm's encoding of message m in the presence of *fault pattern* E_j. A fault pattern E is a set $\{p_1, p_2, \ldots, p_f\}$, $1 \leq p_i \leq b(n)$, of bit positions that are stuck-at-0. There are clearly $\binom{b}{f}$ such patterns.

Any such table must satisfy the following criteria:

1. *(Physical consistency)* Any codeword in column $E = \{p_1, \ldots, p_f\}$ must have 0's in at least the positions $p_1, \ldots, p_f$.
2. *(Unique decoding)* Codewords in different rows must be distinct.
3. *(Lack of clairvoyancy)* Let m be a message, and let w_i and w_j be the codewords in columns E_i and E_j of row m. Then in producing w_i and w_j, the writing procedure must follow the same path down the decision tree for m, until a write is attempted at a position p in $E_i - E_j$ or $E_j - E_i$.

We count the number of distinct entries needed for the table to satisfy (1)-(3) above. The logarithm of this number is a lower bound on $b(n)$. The next lemma shows that for almost all messages, the writing procedure must attempt to write a large number of 1's for any fault pattern.

Lemma 2: Assume $f = o(n/\log n)$ and let ϵ be an arbitrarily small positive constant. Then there are at least $2^n(1 - o(1))$ messages where the writing procedure attempts to write at least $b/(2+\epsilon)$ 1 bits for *every* fault pattern, and the $o(1)$ depends on n.

Proof: Call a codeword requiring fewer than $\frac{b}{2+\epsilon}$ attempts to write a 1 a *bad codeword*. The number of bad codewords is given by

$$\sum_{0 \leq t < \frac{b}{2+\epsilon}} \binom{b}{t} < \sum_{0 \leq t < \frac{n+f\log_2 n}{2+\epsilon}} \binom{n + f\log_2 n}{t}$$

$$< 2^{(n+f\log_2 n)H(\frac{1}{2+\epsilon})}$$

where $H(\alpha) = -\alpha \log_2 \alpha - (1-\alpha)\log_2(1-\alpha)$ is the entropy function. The first inequality is true since we assume $b < n + f\log_2 n$ (otherwise a lower bound of $b(n) \geq n + f\log_2 n$ is immediate). The second inequality is a well-known bound from information theory (see, for example, [G]). If a bad codeword occurs on row m, we call m a *bad message*, otherwise m is called a

good message. The number of good messages is minimized when each bad encoding occurs in a different row, so that the number of good messages is at least

$$2^n - 2^{(n+f\log_2 n)H(\frac{1}{2+\epsilon})}$$

This quantity is $2^n(1-o(1))$ when $f = o(n/\log n)$. □

Lemma 3: If m is a good message, then at least $\binom{\frac{n}{2+\epsilon}}{f}$ different codewords appear in row m of the table.

Proof: Let m be a good message and consider its encoding in the presence of some error pattern. We know the encoding algorithm must attempt to write a 1 at least $\frac{n}{2+\epsilon}$ times. Any f of these attempts could conceivably fail, and the encoding algorithm must be able to recover from the failure.

Analogous to the fault pattern, we define a *glitch pattern* to be a sequence $1 \le g_1 < g_2 < \cdots < g_f \le \frac{n}{2+\epsilon}$ which denotes stuck-at-0 failures during attempts $g_1, g_2, \ldots, g_f$ to write a 1 bit, i.e., the first failure occurs at the attempt to write the g_1-th 1, etc. Let $\mathbf{g} = (g_1, \ldots, g_f)$ and $\mathbf{g}' = (g'_1, \ldots, g'_f)$ be two glitch patterns, $\mathbf{g} \sim \mathbf{g}'$. Then in encoding a good message m with glitch pattern $\mathbf{g}$ and then with glitch pattern $\mathbf{g}'$, the encoding algorithm must generate two different encodings for m, and each glitch pattern corresponds to a different error pattern.

Let g_i be the lowest integer appearing in $\mathbf{g}$ but not in $\mathbf{g}'$. By the principle of lack of clairvoyance, the encoding algorithm behaves identically for $\mathbf{g}$ and $\mathbf{g}'$ when encoding m until the g_i-th attempt to write a 1; at this point the respective encodings are guaranteed to be dissimilar, and the fault patterns are clearly different. Since there are at least $\binom{\frac{n}{2+\epsilon}}{f}$ glitch patterns, there are that many codewords in the row for a good message. □

Multiplying the number of good messages by the number of different codewords per good message gives a lower bound of

$$2^n(1-o(1))\binom{\frac{n}{2+\epsilon}}{f}$$

codewords in the table. The logarithm of this expression gives a bound on $b(n)$.

Theorem 4: If $f = o(n/\log n)$, then

$$b(n) \ge n + f\log_2 n - f\log_2(2+\epsilon) - \log_2 f! + o(1)$$

□

We note that this lower bound resembles the lower bound problem studied in [K*] of conducting a binary search in the presence of errors. However, a direct conversion from their problem to ours or vice versa does not appear easy.

V. WHEN FAULTS ARE KNOWN IN ADVANCE

The lower bound shows that with one stuck-at-0 error, about $n + \log_2 n$ bits of write-once memory are needed to encode an n-bit message. What if it were a stuck-at-1 error instead? In this case, $n+1$ bits suffice to encode the message. This encoding shows that the lower bound depends critically on what fault information is known to the encoder in advance.

To correct one stuck-at-1 fault in $n+1$ bits, we divide the write-once memory into a 1-bit *flag* and an n-bit *message region*. The encoder contemplates writing the message m as is in the message region. There are three cases:

Case 1. The stuck bit is in the message region, but does not conflict with the message, i.e., we want to write a 1 in the position that is stuck-at-1. The encoder simply writes the message and leaves the flag bit set to 0.

Case 2. The stuck bit is in the message region, but conflicts with the message. The encoder then writes the *complement* of the message (which does not conflict), and sets the flag bit to 1.

Case 3. The stuck bit is the flag bit. Then the encoder writes the complement of the message.

To decode is easy: if the flag bit is 0, the message region holds the message, and if the flag bit is 1, the message region holds the complement of the message. Note the decoder doesn't know where the stuck bit is.

When $f > 1$ and the location of all stuck bits is known in advance, the above scheme can be applied to pointer codes. By adding a flag bit for the initial block and each correction block, and allowing complementing of the blocks, we can reduce the redundancy of pointer codes from $f\log_2 n(1+o(1))$ to $\frac{1}{2}f\log_2 n(1+o(1))$.

VI. AN AVERAGE CASE ANALYSIS

Another variation of the problem is to consider the expected behavior of the pointer code algorithm. Assume stuck-at-0 and stuck-at-1 errors are equally likely. Then for any constant $\epsilon > 0$, the pointer code algorithm can correct $(2-\epsilon)f$ errors in $n + f\log_2 n$ bits with probability $1 - o(1)$, where the $o(1)$ depends of f. This result can be derived using the strong law of large numbers or the entropy formula used in our lower bound proof.

Our pointer code algorithm is oblivious about stuck-at faults until they are detected, and does not use the fact that a bit is stuck-at-1 to any advantage. When the algorithm tries to write in a stuck bit, the write-once memory either *agrees* or *disagrees* with the bit intended to be written. If we want to write 1 and the bit is stuck-at-1, the fault doesn't have to be corrected. Every disagreement, though, requires a pointer block to correct it.

If b is a stuck bit, and

$$\mathbf{Pr}\{b \text{ is stuck-at-0}\} = \mathbf{Pr}\{b \text{ is stuck-at-1}\} = \frac{1}{2}$$

then

$$\mathbf{Pr}\{\text{write ends in agreement}\} = \mathbf{Pr}\{\text{write ends in disagreement}\} = \tfrac{1}{2}$$

If there are $(2-\epsilon)f$ stuck-at bits, the expected number of disagreements is $(1-\epsilon/2)f$.

Now consider the behavior of the algorithm, trying to encode n bits of information in $n+f\log_2 n$ bits, with $(2-\epsilon)f$ stuck-at faults. Three things can happen:

1. There are $\leq f$ disagreements.
2. There are $> f$ disagreements, but a lot of them are in pointer blocks that are unused, or
3. There are $> f$ disagreements, all requiring correction.

The probability of cases (2) or (3) happening is very small (note we err on the side of safety here, as case (2) is successful). That is:

$$\mathbf{Pr}\{\text{case (2) or (3) happens}\} = \frac{1}{2^{(2-\epsilon)f}} \sum_{d>f} \binom{(2-\epsilon)f}{d}$$

where d is the number of disagreements. But

$$\sum_{d>f} \binom{(2-\epsilon)f}{d} < 2^{(2-\epsilon)fH(\frac{1-\epsilon}{2-\epsilon})}$$

Since $\epsilon \sim 0$ we know $H(\frac{1-\epsilon}{2-\epsilon}) = \epsilon' < 1$, and

$$\mathbf{Pr}\{\text{case (2) or (3) happens}\} < 2^{(2-\epsilon)f(\epsilon'-1)} = O(\alpha^{-f}) = o(1)$$

for some constant $\alpha > 1$. Since the algorithm succeeds in case (1) and fails in cases (2) and (3), the probability that the pointer code algorithm encodes n bits in $n+f\log_2 n$ bits with $(2-\epsilon)f$ stuck-at bits is $1-o(1)$. Note f must increase slowly with n for this result to be true.

This average-case analysis is interesting when we compare it to the variant algorithm sketched in Section V that uses bit complementing when it knows all stuck-at information in advance. The variant algorithm guarantees an encoding of n bits of information in about $n+f\log_2 n$ bits when there are $2f$ stuck-at bits. However, the original algorithm can encode just as well with $(2-\epsilon)f$ stuck-at bits, with probability $1-o(1)$. The comparison of these two algorithms shows that the answer to "how much better can we do if all stuck-at errors are known in advance?" is answered (with respect to our two algorithms): knowing all stuck-at errors gives you about half as much redundancy in the worst case, but roughly the same redundancy on the average.

REFERENCES

[G] Gallager, R. G., *Information Theory and Reliable Communication*, Wiley, 1968.

[HE] Heegard, C. and A. El Gamal, *On the capacity of computer memory with defects*, submitted to *IEEE Trans. Inf. Theory*, 1982.

[K*] Kleitman, D. J., A. R. Meyer, R. L. Rivest, J. Spencer, and K. Winkelmann, *Coping with errors in binary search procedures, 10th ACM Symposium on Theory of Computing*, 1978, pp. 227-232.

[KT] Kusnetsov, A. V. and B. S. Tsybakov, *Coding in a memory with defective cells. Problemy Peredachi Informatsii* 10:2, 1974, pp. 52-60.

[M] Maier, D. *Using write-once memory for database storage, 1982 ACM Symposium on Principles of Database Systems*, March 1982, pp. 239-246.

[RS] Rivest, R. L. and A. Shamir, *How to reuse a "write-once" memory, 14th ACM Symposium on Theory of Computing*, 1982, pp. 105-113.