Transient Errors and Rollback Recovery in LZ Compression

Wei-Je Huang and Edward J. McCluskey
CENTER FOR RELIABLE COMPUTING
Computer Systems Laboratory, Department of Electrical Engineering
Stanford University, Stanford, California 94305
{weije, ejm}@CRC.stanford.edu

Abstract

This paper analyzes the data integrity of one of the most widely used lossless data compression techniques, Lempel-Ziv (LZ) compression. In this algorithm, because the data reconstruction from compressed codewords relies on previously decoded results, a transient error during compression may propagate to the decoder and cause a significant corruption in the reconstructed data. To recover the system from transient faults, we designed two rollback error recovery schemes for the LZ compression hardware, the "reload-retry" and "direct-retry" schemes. Statistical analyses show that the "reload-retry" scheme can recover the LZ compression process from transient faults in one dictionary reload cycle with a small amount of hardware redundancy. The "direct-retry" scheme can recover normal operations with a shorter latency but with a small degradation in the compression ratio.

Keywords: *Fault tolerant computing, LZ compression, Rollback recovery, Concurrent error detection.*

1. Introduction

The scarcity of communication and storage bandwidth highlights the importance of the efficient utilization of channel resources. To this objective, data compression techniques are used to remove the redundancy inherent in the transmitted information.

Lossless compression techniques, where original data can be perfectly reconstructed from the encoded data, are commonly used for applications that require identical reconstruction of the data after decompression. Data size is reduced by encoding long but frequently encountered strings into short codewords with the help of a dictionary. Typical examples include Huffman coding [1], arithmetic coding [2], and Lempel-Ziv (LZ) compression [3][4].

LZ data compression has been widely used in various standards since the mid-80's, including the Unix compress functions, GIF image compression format, and CCITT V.42bis modem. An important distinction of LZ data compression is the *universal* property, which means that the encoding dictionary is built dynamically without any prior knowledge about the statistics of source data. This property is achieved by updating a dictionary sequentially according to previously encountered source data.

The core computation of the LZ data compression includes an exhaustive matching between source data and dictionary elements. Different parallel architectures for speeding up the exhaustive matching process have been proposed, including the Content-Addressable Memory (CAM) approach [5][6] and the systolic array approach [7][8][9]. Although the throughput can be improved by parallel architectures, there is a data integrity issue caused by the sequential update of the dictionary. The data integrity issue occurs because the LZ decoder relies on the previously reconstructed data to update the decoding dictionary, and thus a transient error during compression or in the transmission channel can corrupt the decoding dictionary and successive reconstructed data significantly.

Figure 1 shows a general block diagram of a communication or storage system. Two types of coding schemes are used; *source coding* for data compression and *channel coding* (ECC) for correcting transmission errors. In this system, the effect of interference caused by channel noise and hardware failures in the shaded area can be minimized through ECC, modulation, and equalization techniques. However, these techniques do not provide protection if the input to the channel encoder is faulty due to errors in the source encoder.

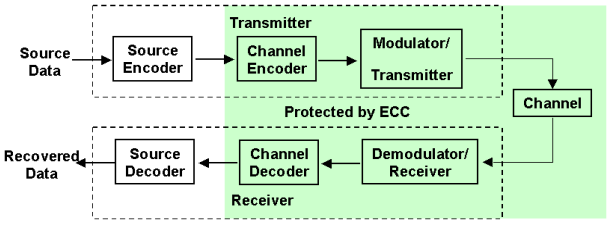

Figure 1: Block diagram of a communication or storage system.

In this paper, the data integrity of the LZ compression algorithm is analyzed quantitatively by computing the number of incorrectly reconstructed data symbols due to a transient error during compression. To guarantee data integrity, efficient Concurrent Error Detection (CED) techniques in the LZ compression hardware have been proposed in [10] and [11]. By checking whether the lossless property of the LZ algorithm is preserved, these

techniques ensure that the compressed codeword can be correctly reconstructed. Based on these CED schemes, we propose efficient error recovery techniques targeting transient errors in this paper.

The organization of this paper is as follows. In Sec. 2, we summarize the LZ compression algorithm. In Sec. 3, we analyze the effect of error propagation due to transient errors in LZ compression. In Sec. 4, we present the CED technique and propose two rollback error recovery schemes for the LZ compression. In Sec. 5, mathematical analyses and emulation results of our rollback recovery techniques are presented. Section 6 concludes the paper.

2. Overview of LZ Compression Algorithm

Figure 2 shows an encoder diagram for LZ data compression [3]. Initially, the dictionary array is empty. In each cycle of the encoding process, the latest source symbol and all dictionary elements shift one position leftwards, and the oldest element in the array is shifted out of the dictionary. In this way, the dictionary contains the most recent source data and is updated dynamically.

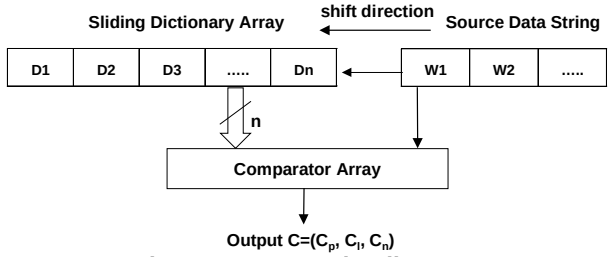

Figure 2: LZ encoder diagram.

Compression is achieved by replacing source data strings with pointers to dictionary elements. Since the dictionary contains the most recent data patterns, long strings that are frequently encountered can be replaced by shorter codewords. To do this, one must find the maximum-length-matching sequence between source data strings and the dictionary elements. Once such a matching sequence is found, the encoder generates a *compressed codeword* $C=(C_p, C_l, C_n)$. Each codeword contains three elements: the *pointer to the starting position* (C_p) of the matching sequence in the dictionary, the *length of the matching sequence* (C_l), and the *next source data symbol* (C_n) immediately following the matching sequence.

Figure 3 shows a simplified example of the LZ77 encoding process. We assume that the 16-element sliding dictionary is initially filled with "betbedbeebearbe ", and the next source data in the input window is "beta bets". We assign a 3-bit length component C_l in each codeword, which encodes a *maximum matching length* (L_{max}) of seven. In this example, the 9-symbol input string can be encoded as two codewords. If the source data is encoded in 8-bit ASCII code, the resulting output size will be 2*(4-bit C_p + 3-bit C_l + 8-bit C_n)=30 bits. Compared to the original data of 9*8=72 bits, the compression ratio is 2.4:1.

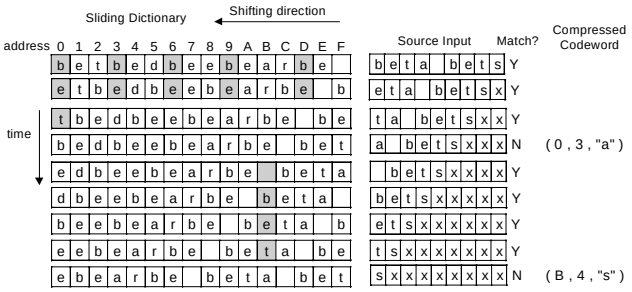

Figure 3: Example of LZ encoding process.

To decode LZ-compressed codewords, one needs to construct a decoding dictionary in the same way as the encoder. In other words, the reconstructed data is shifted into the decoding dictionary in each cycle. Once a compressed codeword is received, original data strings can be reconstructed by extracting C_l symbols starting from position C_p in the decoding dictionary and appending the symbol C_n afterwards. Figure 4 illustrates the decoding process for the example in Fig. 3.

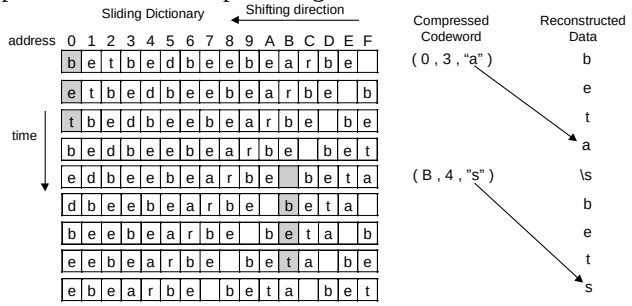

Figure 4: Example of LZ decoding process.

3. Transient Error Analysis

The dynamic construction of the encoding dictionary makes LZ compression useful for all kinds of source data without prior knowledge of their statistics. However, it also makes the reconstructed data prone to error propagation when a transient error occurs during encoding.

Consider a scenario where a transient fault in the encoder causes a bit-flip in the compressed codewords. Since the subsequent channel coding in Fig. 1 does not provide any protection against errors in its input data, the receiver cannot detect errors that occur in the LZ encoder. As a result, an incorrect string is generated in the decoder and is used both in decompression and in updating the decoding dictionary. Once the decoding dictionary holds erroneous sequences, successive reconstructed data may be damaged in an avalanche process even if all the subsequent encoding and decoding operations are error-free.

Figure 5 shows an example from [11] that illustrates this data integrity problem. The error-free encoding and decoding processes of this example are shown in Fig. 3 and Fig. 4. In Fig. 5, the incorrect data are shown in brackets, and the shaded part of the dictionary represents the corrupted entries in each instant of time. Suppose the encoder suffers a transient fault that complements one bit of the position pointer, C_p, in the output and results in an

erroneous compressed codeword (4, 3, “a”) instead of (0, 3, "a"). The decoder extracts an erroneous string of length three due to the faulty C_p pointer. This string then shifts into the dictionary as if it were correct. Since the following codeword refers to the erroneous data string shifted into the dictionary, the subsequent reconstructed data is damaged accordingly. In this example, the single-bit encoding error propagates to corrupt six out of nine reconstructed data symbols.

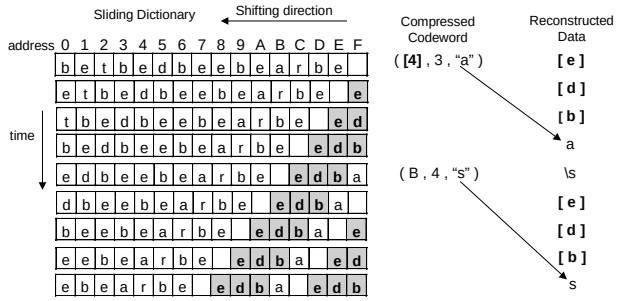

Figure 5: Example of error propagation.

Depending on the nature of source data, the tolerable number of incorrectly decoded symbols can be different. For example, if the LZ compression is used to encode pixel samples in an image in GIF format, a reconstructed image with some erroneous pixels may still be tolerable. Conversely, if the LZ compression is used to encode data that has a bit-by-bit precision requirement, no error is allowed in the reconstructed data. Therefore, to study the effect of this data integrity problem in a particular application, one needs to estimate the total number of erroneous data symbols caused by the error propagation.

To analyze the effect of error propagation in the reconstructed data due to a single transient error during compression, we assume that the position pointer C_p is uniformly distributed over all dictionary positions. Intuitively, because of the temporal locality characteristics in most data [12], C_p should be biased towards the entries that contain the most recent data symbols. Also, the real distribution of C_p depends on the source data statistics. However, because the correlation of any pair of data symbols in a short data sequence is similar, this assumption is close to reality for a small dictionary size (of the order of 1K).

Suppose a transient error during compression creates an incorrect compressed codeword C_{error}, which is the only erroneous codeword in the system. We define several parameters as follows:

- k = the number of erroneous symbols that are shifted into the decoding dictionary immediately after C_{error} is decoded.
- N = the number of entries in the dictionary,
- L_i = the matching length C_l of the i-th codeword following the first erroneous codeword C_{error},
- E_i = the number of incorrect symbols in the decoding dictionary after the i-th codeword following C_{error} is decoded,

and m = lifetime of the first incorrect symbol in the decoding dictionary (in units of the number of compressed codewords encountered in the decoder).

The parameter m means that the first incorrect symbol in the decoding dictionary shifts to the oldest dictionary cell and is shifted out of the dictionary when the m-th codeword following C_{error} is decoded. For example, in Fig. 5, C_{error} = (4, 3, "a"), N = 16, k = 3, L_1 = 4, E_1 = 6, and m = the number of compressed codewords encountered in the decoder when the first corrupted symbols, the first “edb” string constructed by decoding (4, 3, "a"), shift to the address zero.

Before the i-th codeword following C_{error} arrives, the probability that C_p points to an incorrect symbol in the decoding dictionary is E_{i-1}/N. When the decoder processes the i-th codeword, L_i symbols are extracted from the decoding dictionary, one correct symbol is generated because of the correct C_n component, and (L_i+1) old dictionary entries are shifted out. This generates (L_iE_{i-1}/N) extra incorrect symbols on the average. In addition, when $i \geq m$, $(L_i+1)E_{i-1}/N$ incorrect symbols are shifted out of the dictionary. Therefore, E_i can be computed iteratively:

$$\text{Initial } E_0 = \text{number of initil incorrect symbols after decoding } C_{error} = k$$

$$i < m : E_i = E_{i-1} + \text{extra corrupted symbols} = E_{i-1} + \frac{E_{i-1}}{N} L_i$$

$$= E_{i-1}\left(1+\frac{L_i}{N}\right) = k\prod_{j=1}^{i}\left(1+\frac{L_j}{N}\right)$$

$$i \geq m : E_i = E_{i-1} - \text{old corrupted symbols evicted from the dicionary} + \text{extra corrupted symbols}$$

$$= E_{i-1} - \frac{E_{i-1}}{N}(L_i+1) + \frac{E_{i-1}}{N} L_i$$

$$= E_{i-1}\left(1-\frac{1}{N}\right) = E_{m-1}\left(1-\frac{1}{N}\right)^{i-m+1}.$$

Note that E_i increases for $i<m$ towards a maximum value E_{max}:

$$E_{max} = E_{m-1} = k\prod_{i=1}^{m-1}\left(1+\frac{L_i}{N}\right),$$

and converges to zero at a rate of ($1-1/N$) after reaching its peak. The number of incorrect reconstructed symbols n_{error} after decoding r codewords is

$$\text{For } r < m : n_{error}(r) = E_r = k\prod_{i=1}^{r}\left(1+\frac{L_i}{N}\right)$$

$$\text{For } r \geq m : n_{error}(r) = n_{error}(m-1) + \sum_{i=m}^{r}\frac{E_{i-1}}{N} L_i$$

$$= E_{max}\left(1+\sum_{i=m}^{r}\frac{L_i}{N}\left(1-\frac{1}{N}\right)^{i-m}\right).$$

To simplify the analysis, we replace L_i by *the average matching length* L_{avg} in the LZ compression algorithm. If p percent of data size is reduced by LZ compression with *codeword size* = L_{code} bits and *source data size* = L_{source} bits, the results become

$$L_i = L_{avg} = average\ matching\ length = \frac{L_{code}}{(1-p)\times L_{source}} - 1,$$

$$m \approx \frac{Number\ of\ entries\ in\ the\ dictionary}{Number\ of\ symbols\ shifted\ in\ after\ decoding\ a\ codeword} = \frac{N}{L_{avg}+1},$$

$$E_{max} = k\prod_{i=1}^{m-1}\left(1+\frac{L_i}{N}\right) \approx k\left(1+\frac{L_{avg}}{N}\right)^{m-1},$$

$$Total\ number\ of\ incorrect\ reconstructed\ symbols = n_{error}(\infty) \approx E_{max}\left(1+\sum_{i=m}^{\infty}\frac{L_{avg}}{N}\left(1-\frac{1}{N}\right)^{i-m}\right) \approx k\left(1+\frac{L_{avg}}{N}\right)^{m-1}\left(1+L_{avg}\right).$$

Table 1 lists the analytical results for k = 1 and L_{avg} with N ranging from 512 to 4096. k = 1 corresponds to the corruption of the C_n component in the codeword, and k = L_{avg} corresponds to the corruption of the C_p component in the average case. To calculate L_{avg}, we chose 18 Calgary text compression corpus files [13] as a benchmark and measured the average compression percentage p. From this table, it is obvious that a single-bit error propagates to corrupt multiple decoded symbols after a certain period. The total amount of incorrectly reconstructed data is proportional to the number of initial corrupted data symbols k in the decoding dictionary.

For transient errors that corrupt the C_l component in a codeword, we can analyze the effect as follows. Once a codeword with an incorrect C_l is encountered in the decoder, the decoding dictionary becomes a shifted version of its encoding counterpart. All future references to the dictionary will lead to incorrect elements even though the position pointers are valid. This situation can be modeled by a large k value, which represents the difference between the decoding dictionary and its encoding counterpart in an entry-by-entry basis after the first erroneous codeword is encountered. Since a larger k value results in more incorrect reconstructed data, the resulting error propagation problem is more serious when C_l is corrupted.

The previous analysis assumes uniformly distributed matching positions. If we consider the worst case where a single-bit error occurs in encoding a frequently encountered long string, such as the example in Fig. 5, more data will be corrupted as a consequence.

Note that adding error control codes either *before* or *after* compression does not avoid this situation. Applying error control codes such as cyclic redundant checks (CRC) *after* the compressor can improve the system immunity to the noise and interference in communication or storage channels. However, they can neither correct nor detect any hardware failures that occur during compression. Moreover, note that this corruption starts with an incorrect dictionary reference to previously decoded data. Error control codes applied *before* compression may still fail to avoid errors because the incorrectly extracted string from the dictionary may contain a valid (data, checksum) pair reconstructed in previous decoding cycles.

Table 1: Analytical results for transient error effects for N = 512 to 4096.

Number of entries in the dictionary, N		512	1,024	2,048	4,096
Average compression percentage, p		38.67%	43.21%	49.5%	52.96%
Codeword size, L_{code} (bits)		23	24	25	26
Source data size, L_{source} (bits)		8	8	8	8
Average matching length, L_{avg}		3.69	4.28	6.19	6.91
Lifetime of corrupted symbols, m (codewords encountered)		109	194	285	518
Max. number of erroneous symbols in the dictionary, E_{max}	k = 1	2.17	2.23	2.36	2.39
	$k = L_{avg}$	8.01	9.57	14.59	16.52
Total number of incorrect reconstructed symbols, $n_{error}(\infty)$	k = 1	10.18	11.77	16.97	18.90
	$k = L_{avg}$	37.57	50.53	104.90	130.67

4. Error Recovery from Transient Faults

4.1 LZ Compression Hardware and CED Scheme

Figure 6 shows a high-performance systolic array structure of the LZ compression hardware proposed in [9]. There are 512 elements in the dictionary. The sliding dictionary is implemented in shift registers, and each processing element (PE) compares the source data with the corresponding element in the dictionary. The output of a PE represents the matching result of the corresponding position, and it propagates in the next cycle to perform the string matching process. These PE outputs are collectively encoded to generate a matching position pointer and a global *Matched* signal. If a maximum-length matching string is found, all PE outputs and the global *Matched* signal are disabled, and the output codeword is enabled.

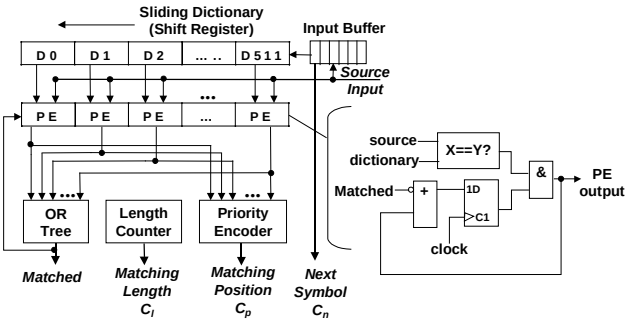

Figure 6: Systolic-array LZ compression structure.

With the advantage of massive parallelism in the systolic array architecture, the encoding throughput is up to one source data symbol per cycle. However, the large array structure makes the traditional *duplex* CED scheme [14] very expensive because every hardware element needs to be replicated. For LZ data compression, a more cost-effective CED technique, called *inverse comparison*, is proposed in [11] and shown in Fig. 7.

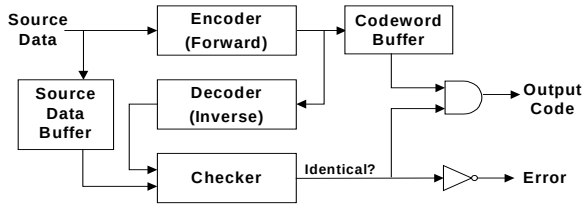

Figure 7: Inverse comparison CED scheme.

The inverse comparison CED technique is based on the lossless property of the LZ compression algorithm. Instead of performing the same encoding computations in multiple copies of hardware, this approach ensures the encoder integrity by checking whether the source data can be reconstructed from the encoded output. The inverse comparison CED scheme has a smaller area overhead than the traditional duplex CED scheme because LZ decoding is simply a memory access, which does not involve a large parallel array for fast comparisons. In addition, the throughput remains unchanged by pipelining the encoding and decoding operations and inserting a source data buffer and a codeword buffer.

4.2 Rollback Error Recovery Schemes

Once an error in a computing system is detected, the next step to improve dependability is to recover from the error and to continue normal operations. There are two major approaches to recover a computing system from transient errors [14]. One approach, *forward recovery*, is to continue to execute the program right from where the error is detected and use alternative mechanisms to ensure correctness during the recovery period. To achieve this objective using hardware approaches, there must be multiple modules in the system to execute the process simultaneously and mask the faulty module if an error occurs. However, for LZ compression hardware with a large array structure, this approach is very expensive.

Another major recovery approach, *rollback recovery*, is to roll the system back to a reliable state to retry the computations again. This introduces a sacrifice in the computation throughput, and one has to estimate the recovery latency of this approach to determine if the throughput penalty is tolerable for the target application. The latency of the rollback recovery consists of two parts: one is the time required for setting up the machine state before the retry, and the other is the re-computing period in which faulty computations are re-executed.

The re-computing period depends on the *rollback distance*, which is measured by the number of cycles between the detection of the error and the restarting point of the retry in the original computation. In a system with clear partitions among the tasks, the restarting point can simply be chosen at the beginning of the faulty task. For example, if the LZ compression hardware is used in a storage system to compress the source files, the restarting point of the rollback recovery can be assigned to the beginning of the file. This assignment method has the advantage of a simple construction of the initial state for the retry. Generally, this also shortens the second part of the recovery latency, the set-up time for the initial state of retry, if resetting the system for a new task is simple. However, this approach causes a large re-computing latency in situations when the transient fault occurs near the end of the task. Also, in *stream-based*, *real-time* applications, where the input stream does not come from a file in the memory system, defining clear partitions of tasks is difficult.

Another approach is to assign the restarting point based on checkpoints. In LZ compression with inverse comparison CED, a *checkpoint* can be defined as the instant when an encoded codeword is generated because the CED only verifies the final encoder output. A checkpoint is passed if the corresponding codeword passes the verification of inverse comparison. Since previous correct outputs need not be regenerated all over again, it is efficient to rollback only to the last passed checkpoint. This approach generally has a short re-computing latency, but one must guarantee the correctness of machine states at checkpoints in order to restart the retry process properly.

The machine state in the LZ compression hardware includes the length counter and the elements in the sliding dictionary. Since the checkpoints in the inverse comparison CED technique are based on the completion of the codeword generation, the length counter simply resets to zero immediately following each checkpoint. For state reconstruction in the dictionary, a simple method is to enable 2-way shifting in the sliding dictionary so that the dictionary entries can be shifted backwards, and the dictionary can return to previous states. However, this method not only doubles the routing and control overhead but may also fail to recover the system if the error occurs in the length counter or a dictionary element, or if there are previously undetected faults in the dictionary elements.

To construct a safe state in the dictionary before retry, we propose the following two techniques: "*reload-retry*" and "*direct-retry*".

4.2.1 The "*reload-retry*" scheme. Since the LZ encoding dictionary is constructed from the previous source data, a reliable approach to rebuilding the dictionary is to shift the source data again from the input end. For file compression in a storage system, we only need an extra register R_A to record the offset pointer that locates the head element of the dictionary in the input file. For stream-based, real-time applications, the system needs an extra storage space for previous source data besides the register R_A. In this approach, the data for the state reconstruction is stored as a separate backup, and thus the correct state can be rebuilt successfully with a greater probability. This is because the state recovery scheme only fails when both the source data backup and the encoder are faulty simultaneously.

To estimate the storage space required for the state rollback during the retry process, let us consider the *error*

detection latency L_{CED} in the LZ compressor. L_{CED} is defined as the duration from the beginning of corruption in matching results to the detection of the error in the same checkpoint period. If we only consider the faults that compromise the encoder integrity, the maximum value of L_{CED} is equal to the duration to process an input sequence with maximum matching length L_{max} in the encoder. This is equal to L_{max} cycles for the encoder with a one symbol per cycle throughput in Fig. 6. For such an encoder, in the worst case, the duration between the beginning of corruption in matching results and the last passed checkpoint is also equal to L_{max} cycles. Therefore, the amount of source data required to be stored for the retrial purpose is $(N + L_{max} + L_{CED,\ max}) = (N + 2L_{max})$ in the LZ compression hardware with an N-entry dictionary.

If the input end of the encoder already has a large FIFO (of size greater than $N + 2L_{max}$) to coordinate the incoming source data flow between the LZ compressor and the external world, we can use it to store source data for retrial without extra hardware cost. To control the input FIFO and the reload of the dictionary, the register R_A is needed to record the FIFO address of the head of dictionary at the beginning of a checkpoint period. After this codeword passes the CED, R_A is incremented by (the matching length of the codeword + 1) to keep track of the next head-of-dictionary address in the input FIFO. When the system needs to fetch data from the external world because of an empty FIFO, the data fetch process stops before it overwrites the cell indexed by R_A.

The "*reload-retry*" scheme in the LZ compression hardware can be summarized as follows:

(1) A buffer of size $(N + 2L_{max})$ is needed to store the stream-based input source data for rollback recovery. The buffer can be assumed to be fault-free at the time of the retry either by the addition of ECC, or by the assumption that there is at most one fault at a time in the system.

(2) An extra register R_A in the LZ encoder stores the buffer address of the head of the dictionary in the beginning of each checkpoint period. R_A is updated when error detection of a codeword is passed successfully. Correct codewords can be transmitted as the encoder output.

(3) A Finite State Machine (FSM) in the LZ encoder is used for retry control, as shown in Fig. 8. In fault-free situations, the FSM is in the *normal* state that indicates a normal operation. Once an error is detected, a reset signal is asserted, and the FSM switches from the *normal* state to a *reload* state.

(4) During the *reload* state, the source data is reloaded from the buffer starting at the position indicated by R_A. The FSM stays at this state for N cycles to completely reload the dictionary. Then the FSM enters a *retry* state, and the encoder tries to process the string S that encountered a fault in the initial trial.

(5) If the codeword representing S passes error detection during the *retry* state, the FSM switches back to the *normal* state. In this case, we claim that the previous error is caused by a transient fault, and the system has recovered from the fault. Otherwise, the FSM goes back to the *reload* state and starts another retry process. After a certain threshold number of failed retrials, we can infer that the system encountered a permanent fault.

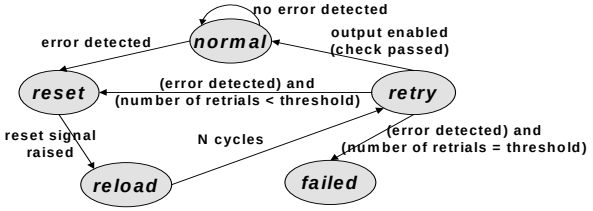

Figure 8: State diagram of the *reload-retry* scheme.

In this *"reload-retry"* scheme, the latency of the rollback recovery is dictated by the time required to reload the dictionary. The latency is *O(N)*, where *N* is the number of entries in the dictionary. In practice, *N* is in the order of 512 to several thousands for a good compression ratio. However, a large *N* value may cause long latency for the rollback recovery and increases the probability to be hit by another upset during recovery. We will analyze the effect of this in Sec. 5.

4.2.2 The "*direct-retry*" scheme. To shorten the rollback recovery latency, another method is to reset the dictionary without reloading it. When an error is detected by the inverse comparison CED, a reset signal is inserted in the encoder output to synchronize the reset operation in the dictionaries of both the encoding and decoding ends. The retry process in the encoder begins immediately after the encoding dictionary is flushed.

Equivalently, the encoder dictionary becomes empty before the source data string under retry is encoded. The encoder is completely initialized once the compression restarts from the source data string that does not pass the CED for the first trial.

In order to insert the reset signal in the encoder output, we need to redefine the codeword components. Since the position pointer C_p is not used for decoding when the matching length C_l is zero, we can use a particular value in C_p with a zero C_l in the codeword to represent a reset signal. This results in a zero penalty in compression ratio during normal operations.

Because the *N*-cycle dictionary reload is not required in the *"direct-retry"* scheme, the rollback recovery latency here is *N* cycles smaller than that in the *"reload-retry"* scheme. Therefore, the worst-case rollback recovery latency in this *"direct-retry"* scheme is $2L_{max}$ cycles (L_{max} is generally 32 to 128). This is at least two times shorter than an *N*-cycle reload in the *"reload-retry"* scheme because *N* is generally greater than 512 for a good compression ratio. Also, the extra backup storage for retry is only $2L_{max}$.

However, the drawback of this scheme is that there is a period of compression ratio degradation after the recovery.

This is because the dictionary is not fully loaded within N cycles after the retry begins, and fewer historical data symbols in the dictionary results in a worse compression ratio. We will analyze this effect in the next section.

5. Analyses of the Rollback Recovery Schemes

In this section, we evaluate the two rollback recovery schemes from different perspectives, including the recovery latency, area overhead, and compression ratio degradation.

5.1 Recovery Latency

As discussed in Sec. 4, the *"reload-retry"* scheme has longer recovery latency than the *"direct-retry"* scheme, and the recovery latency in the *"reload-retry"* scheme is dominated by the dictionary reload period. For a large dictionary with a long reload period, the probability of another upset during recovery increases. In the worst case, the system is always stuck at retrial cycles and fails to recover from transient errors. Therefore, we need to evaluate the effect of this *O(N)* reload period in the error recovery capability of the *"reload-retry"* scheme.

For an N-entry dictionary, we can analyze the problem by calculating two parameters: *the probability of fault-free dictionary reload process* (p_{FFR}), and *the expected reload latency* (T_{reload}). We consider two methods of decoding dictionary management in the inverse comparison CED scheme proposed in [11]. One approach is to use a separate dictionary in the LZ decoder for CED. The other approach is to share the dictionary between the encoder and decoder since they contain the same elements. For the separate-dictionary approach, parity check is not used because there are two separate copies of dictionaries. For the shared-dictionary approach, parity checks are used for error detection in the dictionary elements.

Let p be *the per-bit error rate per cycle in the dictionaries*, and c be *the number of check bits per 8-bit dictionary cell*. For the separate dictionary approach, c = 0. Since there are two N-entry sliding dictionary arrays for reloading, the *probability of a fault-free reload in one cycle* is $p_h = (1\text{-}p)^{16N}$. Similarly, for the shared-dictionary approach with c check bits per 8-bit dictionary cell, $p_h = (1\text{-}p)^{(8+c)N}$. Therefore,

$$p_{FFR} = p_h^N = (1\text{-}p)^{16N^2}$$

for the separate dictionary approach, and

$$p_{FFR} = p_h^N = (1\text{-}p)^{(8+c)N^2}$$

for the shared dictionary approach.

The effect on T_{reload} can be analyzed using the state diagrams shown in Fig. 9. When the separate dictionary is used (Fig. 9(a)), an error is not detected until the dictionary reload is completed because the dictionaries do not contain parity checks. After one dictionary reload, the system encounters an error again with probability ($1\text{-}p_{FFR}$). Assuming the system retries once an error is detected, the reload process can be modeled by a sequence of independent Bernoulli trials with the probability of success = p_{FFR}. Therefore, the number of reload processes needed until the first success is a geometrically distributed random variable with parameter p_{FFR}, which has an expected value of $1/p_{FFR}$. The expected reload latency is $T_{reload} = N / p_{FFR}$.

When the shared dictionary is used with check bits and a checking circuitry in each dictionary cell (Fig. 9(b)), errors in dictionary cells during the reload period can be detected immediately, and the system returns to the beginning of the reload once an error is encountered. Let T_k = *the number of cycles remaining to finish reload when loading the k-th entry*. The recursive equation representing the state diagram becomes

$$T_k = (1\text{-}p_h)(T_1+1) + p_h(T_{k+1}+1),$$

with a final value of $T_{N+1} = 0$.

After recursive computation, we get

$$T_1 = (1\text{-}p_h^N)\, T_1 + (1\text{-}p_h^N) / (1\text{-}p_h),$$

and $T_{reload} = T_1 = (1\text{-}p_h^N) / [\, p_h^N(1\text{-}p_h)\,]$ for the shared-dictionary with parity check approach.

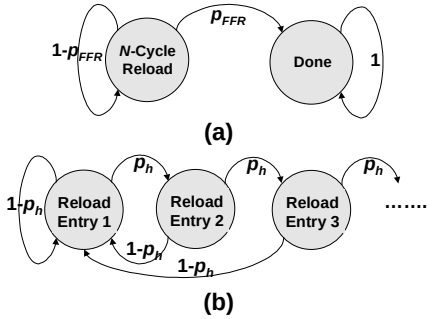

Figure 9: State diagrams for the analysis of reload latency: (a) separate dictionary without parity checks; (b) shared dictionary with parity checks.

For a 16K-entry dictionary with a very high bit-flip probability $p = 10^{-5}$ per second per bit, the estimated probability of faulty reload ($1\text{-}p_{FFR}$) = 2.68×10^{-3} for the separate dictionary approach and 1.51×10^{-3} for the shared dictionary approach with 1 check bit per 8-bit cell. The expected reload latency T_{reload} = 1.003 dictionary reloads (16428 cycles) for the separate dictionary approach and 1.0008 dictionary reloads (16396 cycles) for the shared dictionary approach. Here, the clock rate is assumed to be 16MHz, which is the frequency of our LZ compressor emulation in [11] using the WILDFORCE system [15] with 4 Xilinx 4036XLA FPGAs [16]. For an optimized system with a higher clock rate, smaller per-bit error rate per cycle and shorter recovery latency are expected.

It is clear that even when p and N is very large, the system is still safe from deadlock in retrial cycles. For normal values of p or N, T_{reload} is closer to one dictionary reload cycle. This corresponds to 32 μsec for a 512-entry dictionary reload, or 1.024 msec for a 16K-entry dictionary reload at 16MHz clock rate.

In addition, even if a 16K-entry compressor encounters an extreme environment with a very large $p = 10^{-5}$ per second per bit, the throughput penalty due to the "*reload-retry*" scheme is still very small. In this extreme case, the

occurrence rate of transient errors is about 1 error per second, and the throughput penalty is in the order of 1ms/1sec = 0.1%.

5.2 Area Overhead

As discussed in Sec. 4, the extra storage required in stream-based applications is ($N + 2L_{max}$) entries for the *"reload-retry"* scheme and $2L_{max}$ entries for the *"direct-retry"* scheme. Since this source data FIFO can be stored in a RAM, this extra storage is more area-efficient than the encoding dictionary that is stored either in a CAM array or shift registers for parallel matching. For example, in our FPGA emulation in [11] with N = 512 and L_{max} = 63, the area overhead in terms of Configuation Logic Blocks (CLBs) is only 16% for the *"reload-retry"* scheme and 3% for the *"direct-retry"* scheme.

5.3 Compression Ratio Degradation

The *"reload-retry"* scheme does not cause any degradation in compression ratio because the dictionary is fully reloaded. However, for the *"direct-retry"* scheme, the compression ratio is sacrificed during the first N cycles after the retry process begins.

The compression ratio degradation in the *"direct-retry"* scheme can be measured by simulating the average compression percentage during the initialization phase of the dictionary. The *initialization phase* is defined as the period when the number of data symbols in the dictionary grows from zero to N. We used 18 Calgary compression corpus files [13] as our benchmark and flushed the dictionary every N cycles during compression. The resulting average compression ratio degradation during the initialization phase of the dictionary is 22% to 25% for N=512 to 4K. If we calculate the average compression ratio by considering an extreme case of transient error occurrence rate described in Sec. 5.1, the average degradation is negligible since the duration of the initialization phase is very small compared to normal operations in which the dictionary is fully loaded.

6. Conclusion

Transient errors in an LZ encoder can propagate to cause significant corruption to the reconstructed data. Two rollback recovery schemes based on the inverse comparison CED can be used to recover the LZ encoder from such transient errors.

In the *"reload-retry"* scheme, the error recovery latency is proportional to the number of dictionary elements. Statistical analyses show that this scheme can recover the LZ encoder within one dictionary reload cycle. The emulation results also show a small area overhead because of the extra storage. Since the compression ratio is unchanged, this scheme is suitable for applications with a very high compression rate requirement.

In the *"direct-retry"* scheme, both the recovery latency and extra area overhead are smaller than the *"reload-retry"* scheme, and this scheme can recover the LZ encoder with a small degradation in the compression ratio. This scheme is suitable for applications with a short recovery latency requirement.

Acknowledgements

The authors would like to thank Dr. Nirmal Saxena, Dr. Santiago Fernandez-Gomez, Dr. Subhasish Mitra, Philip Shirvani, and Shu-Yi Yu for their valuable feedback and suggestions. This work was supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. DABT63-97-C-0024.

References

[1] Huffman, D. A., "A Method for the Construction of Minimum Redundancy Codes," *Proc. IRE*, Vol. 40, pp. 1098-1102, 1952.

[2] Rissanen, J. J., "Generalized Kraft Inequality and Arithmetic Coding," *IBM J. Res. Develop*., Vol. 20, No. 3, pp. 198-203, 1976.

[3] Ziv, J., and A. Lempel, "A Universal Algorithm for Sequential data Compression," *IEEE Trans. Information Theory*, Vol. IT-23, No. 3, pp. 337-343, 1977.

[4] Ziv, J., and A. Lempel, "Compression of Individual Sequence via Variable-Rate Coding," *IEEE Trans. Information Theory*, Vol. IT-24, No. 5, pp. 530-536, 1978.

[5] Jones, S., "100-Mbps Adaptive Data Compressor Design Using Selectively Shiftable CAM," *IEE Proc. G*, Vol. 139, No. 8, pp. 498-502, 1992.

[6] Lee, C. Y., and R. Y. Yang, "High-Throughput Data Compressor Design Using Content Addressable Memory," *IEE Proc. G*, Vol. 142, No. 1, pp. 69-73, 1995.

[7] J. Storer, J. H. Reif, and T. Markas, "A Massively Parallel VLSI Design for Data Compression Using a Compacy Dynamic Dictionary, " Proc. IEEE Workshop VLSI Signal Processing, pp. 329-338, 1990.

[8] Ranganathan, R., and S. Henriques, "High-Speed VLSI Design for Lempel-Ziv-based Data Compression," *IEEE Trans. Circuits and Systems*, Vol. 40, pp. 96-106, Feb., 1993.

[9] Jung, B., and W. P. Burleson, "Efficient VLSI for Lempel-Ziv Compression in Wireless Data Communication Networks," *IEEE Trans. on VLSI Systems*, Vol. 6, No. 3, pp.475-483, 1998.

[10] Cheng, J.-M., L. M. Duyanovich, and D. J. Craft, "A Fast, Highly Reliable Data Compression Chip and Algorithm for Storage Systems," *IBM J. Research and Development*, Vol. 40, No. 6, pp. 603-613, 1996.

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

[12] Patterson, D. A. and J. L. Hennessy, *Computer Architecture - A Quantitative Approach*, Morgan Kaufmann Publishers, Inc., 1990.

[13] Calgary Text Compression Corpus, available from Univ. of Calgary, Canada. *ftp://ftp.cpsc.ucalgary.ca*.

[14] Pradhan, D. K., *Fault-Tolerant Computer System Design*, Prentice Hall, 1996.

[15] Annapolis Micro Systems Inc., *http://www.annapmicro.com*, 2000.

[16] Xilinx Inc., *http://www.xilinx.com*, 2000.