Error-Resilient Data Compression With Tunstall Codes

Shanshan Liu, *Member, IEEE*, Pedro Reviriego, *Senior Member, IEEE*, Anees Ullah, *Member, IEEE*, Ahmed Louri, *Fellow, IEEE*, and Fabrizio Lombardi, *Life Fellow, IEEE*

***Abstract*— Data compression has been commonly employed to reduce the required memory size for emerging applications with large storage needs like Big Data and Machine Learning (ML). When considering the flexibility of decompression and its hardware implementation, variable-to-fixed length codes (e.g., Tunstall codes) are usually selected. However, memories are prone to suffer different types of errors, causing the stored data to be corrupted; if an error affects the compressed data, it can propagate and cause corruption in a sequence of bits of the decompressed data. Therefore, error resilience should be built-in as part of the memory design to provide reliable data, especially for safety-critical applications. However, Error Correction Codes (ECCs) that are widely used for memory protection, are not very efficient to protect compressed data, because ECCs further increase the memory size and the additional decoding process can impact the latency to decompress the stored data. In this paper, an efficient error-resilient data compression technique with Tunstall codes is proposed; it requires almost no memory overhead and can correct most errors during the decompression process by introducing a conversion table. An enhanced design is also presented to reduce the impact of errors when they cannot be corrected. The proposed scheme has been implemented and evaluated on three ML datasets; results show that it can deal with up to 99.98% errors with almost no memory overhead when Tunstall codes with smaller than 16-bit symbols are employed. The scheme has also been evaluated for two ML applications; results show that even though a small number of errors cannot be corrected in the proposed scheme, they have an extremely low impact on the classification results and the protection overhead is significantly lower than existing ECC techniques.**

***Index Terms*— Data compression, error resilience, Tunstall codes, machine learning, error correction codes.**

Manuscript received 31 October 2022; revised 28 January 2023; accepted 12 February 2023. Date of publication 17 February 2023; date of current version 28 April 2023. This work was supported in part by the U.S. Department of Defense under Contract W52P1J2093009; in part by the Spanish Agencia Estatal de Investigación under Grant PID2019-104207RB-I00 and Grant TSI-063000-2021-127; and in part by the NSF under Grant CCF-1953961, Grant 1812467, Grant 1812495, and Grant 1953980. This article was recommended by Associate Editor M. Martina. *(Corresponding author: Shanshan Liu.)*

Shanshan Liu is with the Klipsch School of Electrical and Computer Engineering, New Mexico State University, Las Cruces, NM 88001 USA (e-mail: ssliu@nmsu.edu).

Pedro Reviriego is with the Departamento de Ingeniería de Sistemas Telemáticos, Universidad Politécnica de Madrid, 28040 Madrid, Spain.

Anees Ullah is with the Department of Electronics Engineering, University of Engineering and Technology Peshawar, Abbottabad 220101, Pakistan.

Ahmed Louri is with the Department of Electrical and Computer Engineering, George Washington University, Washington, DC 20052 USA.

Fabrizio Lombardi is with the Department of Electrical and Computer Engineering, Northeastern University, Boston, MA 02115 USA.

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TCSI.2023.3245022.

Digital Object Identifier 10.1109/TCSI.2023.3245022

I. Introduction

As Big Data and Machine Learning (ML) have been widely advocated and employed in the last decade, the volume of data handled by computing systems has increased dramatically. Storage of data has become a major challenge due to technology (such as CMOS scaling) and often strict requirements in many resource-constrained platforms. A commonly used solution to address this issue is to employ data compression by coding, because it requires no change in the processor architecture and can be performed nearly transparently to the system hardware [1], [2]; so, instead of the original dataset, only the compressed data that has a significantly smaller size, needs to be stored in memory. When the dataset needs to be accessed for an application, the compressed data is read out of memory and then, decompression is performed to recover the original dataset.

An often-employed category of data compression codes is the fixed-to-variable length (FV) codes like Huffman codes [1]; they compress a dataset by converting a fixed length of elements (i.e., data patterns) to a variable length of binary bits (i.e., symbols). To achieve a lower compression ratio, frequent data patterns are usually assigned to symbols of smaller size. However, symbols have variable lengths, so errors (as introduced next) changing one symbol to another with different length, may likely propagate in all subsequent symbols, causing them to shift forward or backward. Another disadvantage of FV codes is that despite large hardware, they incur in a significant decompression latency, because the location of each symbol cannot be promptly identified (i.e., decompression must be performed serially symbol by symbol, and usually in software). Another category is variable-to-fixed length (VF) codes like Tunstall codes [2], [3], [4]; VF codes convert data patterns with variable length to symbols with fixed length. While their compression ratio is usually worse than FV codes, they are readily indexable due to the fixed length symbols, so more attractive when considering the decompression flexibility and a hardware implementation. In the decompression of VF codes, the compressed data can be divided into single blocks, so that all blocks are decompressed in a single clock cycle. Moreover, errors affecting a symbol of VF codes have a smaller impact than in FV codes, because they are bounded in a single symbol.

Memories are prone to suffer from errors, causing data corruption. For example, radiation-induced soft errors [5], [6], [7], [8], pose a significant reliability issue for

memories, as they can erroneously flip the value stored in a memory cell. This may have no impact on some applications that are naturally tolerant to errors to a large extent (and in which error protection techniques are often not employed). However, such data corruption is not acceptable in some safety-critical applications allowing zero or very small error rates because an incorrect outcome may cause a potential system failure [9], [10], and thus, error protection techniques are required for memories. This is also applicable to memories that store compressed data, because even a single error (the most common type of errors [5], so considered in this paper) may corrupt a sequence of bits in the decompressed data.

Error Correction Codes (ECCs) are widely used to provide error protection for memory [11], [12], [13], [14]. By adding several ECC bits to each memory word, errors can be detected/corrected during the decoding process, such that the correct data is finally provided. The overhead due to ECCs originates from two sources: i) the additional memory cells in each word for storing the ECC bits; ii) the hardware for the encoder that generates the ECC bits and the decoder that performs error detection or correction. Therefore, even though the overhead is significantly lower than other redundancy-based protection techniques like duplication or triplication [15], the use of ECCs for memories storing compressed data in resource-constrained platforms or high-performance systems presents unique challenges: i) the size of compressed data can also be large, so an ECC further increases the required memory size; ii) there is already a large number of memory reads in Big Data/ML applications, so ECC decoding that is performed prior to decompression, further increases the burden on timing. Therefore, with the need of memory protection against errors and the limitations of using ECCs, an error protection technique with reduced latency and hardware area is always attractive. This has motivated this paper to propose an efficient error-resilient scheme with low impact on both memory size and decompression latency.

The main contributions of this paper are as follows:

- An error protection scheme for a memory that stores data compressed by using VF codes (in particular, Tunstall codes), is proposed; it detects/corrects up to 99.98% errors in the memory with a slight or almost no memory overhead when Tunstall codes with smaller than 16 bits symbols are used. Moreover, it also significantly reduces the latency added to the unprotected implementation compared to existing protection techniques.
- When an error cannot be detected/corrected, an additional enhanced design reduces its impact, so that it affects only up to a few elements and does not propagate, hence preventing the occurrence of a so-called global error.

The rest of the paper is organized as follows. Section II reviews data compression by utilizing Tunstall codes, the impact of errors on data compression and traditional error protection solutions. Section III presents the proposed error-resilient compression scheme that can correct nearly all errors; an improvement is further designed to reduce the impact of the remaining uncorrectable errors. The error protection capability and memory overhead of the proposed schemes have been evaluated in Section IV. The use of the proposed scheme in two applications including k Nearest Neighbors [16] and Stochastic Computing-based Neural Networks [17] is studied in Section V. Finally, the paper ends with the conclusion in Section VI.

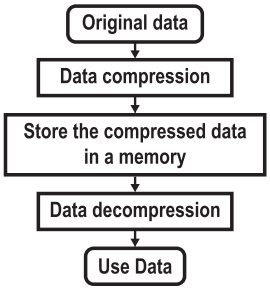

Fig. 1. Flowchart of data compression/decompression.

II. Preliminaries

Codes are often employed for compressing data; instead of the original dataset, compressed data is stored in a memory, so significantly reducing the required memory size. When the original data needs to be accessed/used, it is recovered by decompressing the stored data once it is read out from the memory (as illustrated in Fig. 1). In this section, a brief description of data compression codes focusing on Tunstall codes and implementations are first provided. Then, the impact of errors in the memory is discussed. Finally, error-resilient and protection techniques employing ECCs are reviewed, and the need for efficient alternative approaches is discussed.

A. Tunstall Codes

Prior to introducing Tunstall codes, a terminology is introduced for clarification:

- The ***dataset*** refers to the original uncompressed data stream (e.g., AABABCAAAB).
- An ***element*** is the basic member (i.e., numbers, letters, etc.) of the dataset (e.g., A, B, C).
- The ***data pattern*** is a set of few elements in a specific order; for Tunstall codes, the data patterns have a variable length (e.g., B, AB).
- A ***symbol*** is a set of a binary bits that are associated with each data pattern; for Tunstall codes, symbols have a fixed length (e.g., 000, 010).
- A ***codeword*** refers to the compressed dataset (e.g., 101010001100000).

Next, for simplicity, a dataset that consists only of elements A, B, C is used as an example to introduce the compression mechanism of Tunstall codes. To compress such a dataset with N different elements and maximize the average number of data patterns, the standard n-bit Tunstall code [1] is designed as:

$$N + k\,(N-1) \leq 2^n, \tag{1}$$

where k is the number of iterations to determine data patterns.

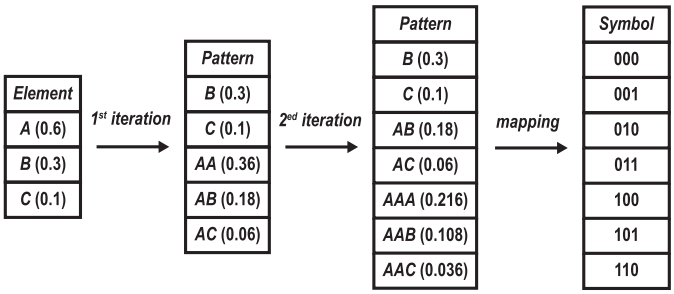

Fig. 2. An example of determining data patterns to compress a dataset with elements {A, B, C} by using a 3-bit Tunstall code.

To determine the data patterns, all elements are sorted by their frequency[1] p first; then the most frequent element is removed from the list and utilized to build new patterns with each of the original elements in the first iteration. For example, consider a dataset with $p(\mathrm{A}) = 0.6$, $p(\mathrm{B}) = 0.3$ and $p(\mathrm{C}) = 0.1$; element A is removed first and then the new patterns AA, AB, AC are obtained with p (AA) = 0.36, p (AB) = 0.18, p (AC) = 0.06. Therefore, the list of patterns is updated (i.e., including B, C, AA, AB, and AC). In the second iteration, the most frequent pattern of the updated list (i.e., AA) is removed to build new patterns by combining it with the original elements. Therefore, the pattern list is updated to B, C, AB, AC, AAA, AAB, AAC (with p (AAA) = 0.216, p (AAB) = 0.108, p (AAC) = 0.036). The same process applies to the remaining k-2 iterations, until the largest value of k for which Eq. (1) holds, is reached. Then different Tunstall symbols are assigned to each pattern. Assume that a 3-bit Tunstall code is employed in the example considered previously; the procedure to determine the data patterns using two iterations and the mapping with symbols in a sequential binary order are illustrated in Fig. 2.

Once the data patterns have been prepared and the mapping with symbols is determined, the compression is performed by serially checking all elements of the dataset with each pattern from the most frequent to the least frequent ones. Once a pattern is matched, it is kept as a part of the codeword and the remaining elements are checked with each pattern again, until all elements have been encoded. Consider the example shown in Fig. 2; if the dataset is AABABCAAAB, the encoded codeword (i.e., the compressed data) is 101010001100000.

Since compression is performed on a pattern-by-pattern basis, the last few elements may not match any pattern. A solution is to pad one or more additional elements to make them match a predetermined pattern; then these extra elements are removed during decompression. However, in this case the size of the original dataset must be stored a priori, that can have disastrous consequences if it is affected by an error [3]. Another solution is to take them as an additional data pattern and assigning a remaining symbol; this is feasible because there are always some symbols unused for any pattern (as per observation 2 in Section III-A) and this will be employed in this paper.

[1]The assumption that each element has an independent frequency is taken in this paper. However, the proposed scheme also applies to the case where elements are dependent; this is valid because the proposed scheme uses the same data patterns as in the original unprotected scheme (that are determined based on either the dependency or independency of elements).

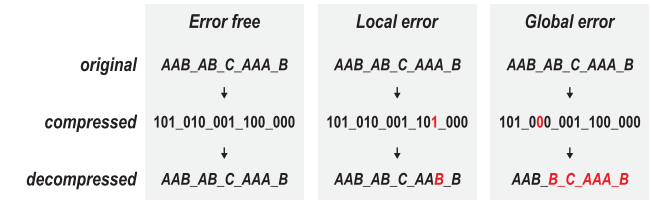

Fig. 3. Impact of different errors (marked in red) on the decompressed data (compression is based on Fig. 2).

When the dataset needs to be accessed, the compressed data is read out from the memory and the decompression process is executed. This is implemented by checking every n bits (i.e., a symbol) to identify the associated data pattern as per a look-up table (which also needs to be stored in memory). The decoding procedure is completed once all data patterns have been recovered from the codeword; the original uncompressed dataset is then output.

B. Memory Errors and Their Impact

Memories can be affected by different types of errors, causing bit flips. For example, it has been experimentally observed that radiation-induced errors have become a significant reliability issue for memories [5], [8]. This occurs because different types of radiation particles exist in almost every operational environment of ICs. If a particle hits through a circuit node of a memory cell, additional charge is generated; then, the voltage of the node is changed if the accumulated charge exceeds a threshold amount (known as critical charge [6], [7]). As a result, the value stored in this memory cell flips from "0" to "1" or the other way around. Another example of bit-flip errors is the retention errors in DRAMs caused by cell discharge [18], or errors caused by overlapping of levels in emerging memories like phase-change memory [19]. Even though the probability of error occurrence is not very high (e.g., average 0.91 radiation-induced errors per day in a 14 MB 40 nm SRAM [8] and the rate increases proportionally for a larger memory or an advanced technology), they cannot be ignored in safety-critical applications. Since single bit errors are the most common scenario [5], they are considered in this paper.

When the memory is used to store compressed data, errors can be classified in the following two types as per the impact on the final decompressed data; this is illustrated in Fig. 3 by considering again the previous example of Fig. 2.

Local errors: a local error modifies a symbol of the stored codeword to another symbol, and the data patterns related to these two symbols have the same length. In this case, the error only changes a limited number of elements in the entire decompressed data, i.e., having a local impact. For example, in Fig. 3, a local error only causes an incorrect element B after decompression, but it does not change the length of the dataset.

Global errors: a global error modifies one symbol of the stored codeword to another symbol, but the data patterns related to these two symbols have different lengths. In this

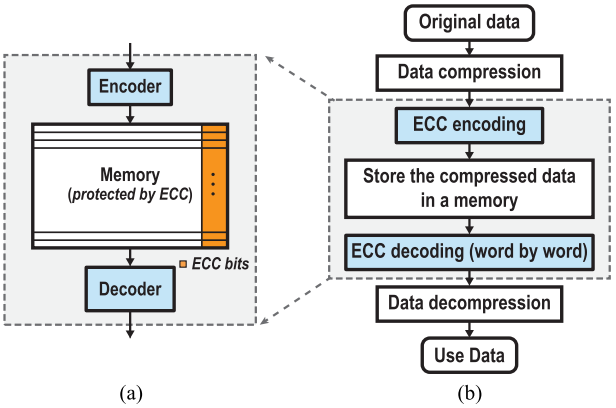

Fig. 4. Using ECCs to protect memory data: (a) diagram of the circuits; (b) flowchart of data compression/decompression with ECC protection.

case, the error is spread to the decompressed data, causing all remaining elements (starting with the element related to the incorrect symbol) to be corrupted, i.e., having a global impact. For example, in Fig. 3, a global error causes the absence of an element A after decompression, so shifting and corrupting all remaining elements starting from the fourth element.

Both types of errors cause data corruption, but global errors tend to have a significantly larger impact. For example, in some ML algorithms like classifiers, the training datasets are usually large, so they are compressed to save storage requirement. As a local error only affects one or a limited number of instances, it likely has a slight or negligible impact on the classification result. However, a global error may make features and labels to shift after decompression and affect a significant part of the training dataset, causing incorrect classification results.

C. Error Correction Codes

Error Correction Codes (ECCs) have been widely used in the past decades to protect memories against errors [11], [12], [13], [14]; they have been utilized into many custom memory chips [20]. As shown in Fig. 4 (a), when ECCs are employed, the data is encoded before writing into the memory to generate ECC bits. These bits are stored together with the original data in each word, so considered as redundancy. If bit-flip errors occur in the memory, they can be detected/corrected as per different ECC algorithms in the decoding process once the data is read.

The simplest ECC is the Single Parity (SP) check, that requires only one ECC bit and detects any single bit errors (i.e., the most common type of error). When error correction must be provided for a memory, Single Error Correction (SEC) codes (such as the Hamming code [12]) are usually selected due to the small redundancy added to the data and the low implementation complexity in hardware. By utilizing $\log_2(d)+1$ redundant bits (where d is the length of the data being protected), SEC codes correct any single bit errors in a memory word.

As introduced previously, the use of ECCs for memories storing compressed data in resource-constrained platforms or high-performance systems, presents unique challenges due to the additional ECC bits and decoding process. Hence, it is of interest to explore the inherent redundancy of the compression codes to achieve error resilience with low impact on the overall memory size and decompression latency.

III. Proposed Scheme

In this section, several observations on the data compression process are initially provided; they are used to propose an efficient error protection scheme. Then, the proposed scheme is analyzed regarding error detection and correction; an improved design is also provided. An example is then given to illustrate the proposed scheme in more detail. Finally, the decompression table required in the proposed scheme is illustrated.

A. Properties of Data Compression With Tunstall Codes

The proposed scheme is designed as per the following two observations on data compression using Tunstall codes:

Observation 1: Some data patterns have a higher probability of occurrence (i.e., those with a larger value of pattern frequency p) and some are less frequent. Moreover, many datasets have a skewed distribution, for example some features of ML datasets can only have a few values (as related to different classifications).

Observation 2: In most cases, few symbols are left unused for any pattern, because the "=" in Eq. (1) is usually not met (i.e., "<" is met in most cases). Moreover, some predetermined data patterns may not occur in the encoding procedure, and thus their related symbols are also not used.

These observations make it possible to exploit the inherent redundancy of Tunstall codes to provide error resilience; so, by carefully assigning symbols for the data patterns, most errors can be detected and corrected, as discussed next.

B. Data Compression

In the proposed data compression scheme, the data patterns are initially determined as per the process introduced in Section II-A, and five sets of Tunstall symbols {S1}, {S2}, {S3}, {S4}, {S5} are determined by using Algorithm 1. Then, a specific mapping rule between the data patterns and these sets of Tunstall symbols is implemented as per the following steps.

Step 1: assign symbols {S1} obtained in line 2 or 4 of Algorithm 1 that have at least Hamming distance[2]-3 from each other for the most frequent patterns (MFPs). This allows that in the presence of errors, a symbol associated with an MFP cannot be changed and be associated to another MFP.

Step 2: keep all symbols {S2} obtained in line 5 or 8 of Algorithm 1 that have distance-1 from each symbol of {S1} as unused. They are used for recovering the correct MFPs when any symbol of {S1} is corrupted by an error.

[2]In this paper, the Hamming distance of two symbols is defined as the number of bits that are different between these two symbols. For example, two symbols having Hamming distance-3 means that they have three bits with different values.

Step 3: assign all symbols {S3} obtained in line 10 or 12 of Algorithm 1 that have distance-2 from the symbols of {S1} for the least frequent patterns (LFPs).

Step 4: assign symbols {S4} obtained in line 17 of Algorithm 1 for the remaining patterns. Note that there may be some spare symbols {S5} obtained in line 18 of Algorithm 1 for which no pattern is assigned.

Finally, the original dataset is compressed by converting each pattern from the beginning to the end of the dataset to the associated symbols. The codeword with a significantly smaller size, is stored in the memory instead of the original dataset. In the proposed scheme, only the symbols of {S1}, {S3} and {S4} are used for compression, while the symbols of {S2} and {S5} are used for error protection. Algorithm 1 essentially provides an alternative to the traditional method of assigning Tunstall symbols; it is implemented in software prior to storing the symbols, so no dedicated hardware design is required.

Algorithm 1 * Process to Determine Tunstall Symbols Used in the Proposed Data Compression Scheme.

1: Prepare all possible 2^n symbols {S} as per the size of Tunstall codes n;
2: {S1} = Dis3 ({S}, all-zero symbol);
3: **if** Size ({S1}) >= num_{dp}
4: {S1} = Part ({S1}, num_{dp});
5: {S2} = Dis1 ({S}, {S1});
6: **Finish**;
7: **else**
8: {S2} = Dis1 ({S}, {S1});
9: {S_{aux}} = Setdiff ({S}, {S2});
10: {S3} = Dis2 ({S_{aux}}, {S1});
11: **if** Size ({S1}) + Size ({S3}) >= num_{dp}
12: {S3} = Part ({S3}, num_{dp} − Size ({S1}));
13: **Finish**;
14: **else**
15: {S_{aux}} = Setdiff ({S_{aux}}, {S1});
16: {S_{aux}} = Setdiff ({S_{aux}}, {S3});
17: {S4} = Part ({S_{aux}}, num_{dp} − Size ({S1}) − Size ({S3}));
18: {S5} = Setdiff ({S_{aux}}, {S4});
19: **end**
20: **end**

*Definition of functions: Dis3(input1, input2) denotes the process to obtain a set of symbols in input1 with at least Hamming distance-3 from each other, starting with input2; Dis1/Dis2(input1, input2) denotes the process to obtain a set of symbols in input1, with Hamming distance-1/2 from input2; Part(input1, input2) denotes the process to output only the first input2 members of input1; Setdiff(input1, input2) denotes the process to remove input2 from input1; Size(input) denotes the process to calculate the number of members in input;

C. Data Decompression

The specific mapping rule used in the proposed compression scheme permits a decompression of data that is resilient to errors on the stored codeword. This is achieved by employing the proposed decompression process. In particular, if a symbol in the received codeword that is being checked to recover a data pattern, belongs to {S1} or {S2}, it is decoded as the corresponding MFP; otherwise (i.e., it belongs to {S3} or {S4}), it is decoded as the corresponding LFP. If {S5} exists and one of the spare symbols is received, it is decoded as a random pattern. This decompression process is implemented using a conversion table in addition to the standard look-up table (LUT) that is used to store data patterns in the traditional scheme.

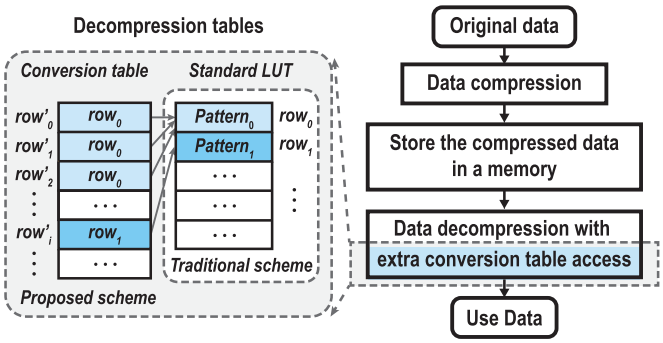

Fig. 5. Flowchart of the proposed error-resilient data compression/ decompression.

In the traditional scheme, the sequential Tunstall symbols are often used in the mapping rule. For example, if an 8-bit Tunstall code is utilized, symbols starting with 00000000 in a sequential order are used to map the data patterns as per the traditional mapping rule. In this case, a standard LUT only needs to store the data patterns for decompression. In particular, each data pattern is stored in the row associated to the binary sequence of the Tunstall symbol mapped to this pattern; for example, the pattern associated to symbol 00000000 is stored in row_0, the pattern associated to symbol 00000001 is stored in row_1, etc. Hence, the decompression is simply performed by utilizing the content of each symbol of the codeword as an address/location of the standard LUT to retrieve its associated data pattern.

In the proposed scheme, a specific mapping rule is utilized; recall that only symbols of {S1}, {S3}, and {S4} are used for compression while the remaining symbols of {S2} and {S5} are used for error correction. Therefore, the stored symbols are not sequential, so the standard LUT is not applicable. To address this issue, an additional conversion table can be simply combined with the standard LUT to retrieve the data patterns. As shown in Fig. 5, this conversion table stores the location of each pattern stored in the standard LUT. Once a symbol is received, its content is used as an address (i.e., *row'*) to access the conversion table first to obtain a location information (i.e., *row*); then, this information is considered as an address of the standard LUT to further locate the correct data pattern.

Hence, in the proposed scheme, mapping from Tunstall symbols to data patterns during decompression is rather straightforward and by introducing a conversion table, the original decompression can be transparently used. Moreover, error protection is also implemented as part of the translation (so in a single memory access). This is performed by linking the symbols used for error protection to the positions in the

standard LUT associated to the corrected data patterns. For example, consider an 8-bit Tunstall code again; 00000000 is used for mapping the first MFP; symbols with distance-1 from it (e.g., 00000001, 00000010, …, 10000000) are reserved in {S2} for correctly recovering this MFP. Therefore, the position in the LUT for this MFP (i.e., 00000000) can be also stored in row'_1, row'_2, …, row'_{128} in addition to row'_0; then, if any of these reserved symbols is received, the corrected data pattern is identified and the error is immediately corrected. As shown in Fig. 5, if $Pattern_0$ is the first MFP and stored in row_0 of the standard LUT, row_0 is stored in row'_0 of the conversion table for recovering $Pattern_0$ in the error-free case and also stored in row'_1, row'_2, etc. for correctly recovering $Pattern_0$ under errors.

As shown in Fig. 5, the proposed scheme only introduces an additional step for accessing the conversion table to the unprotected case (Fig. 1); it is more efficient than an ECC scheme (Fig. 4) in terms of latency, because a data access is significantly less complicated than the encoding/decoding process. The size of the conversion table is $n \cdot 2^n$ bits when an n-bit Tunstall code is employed because there are at most 2^n combinations of a received symbol that can be correct or incorrect. Even though this conversion table also needs to be stored in a memory together with the standard LUT, it incurs in a small memory overhead as evaluated in the next section. These tables may also be corrupted by memory errors; however, different from the memory storing the compressed data, the size of such table is small, so an ECC can be employed because its incurred overhead is acceptable. This is not further pursued in this paper, because it would be negligible compared to the overhead associated with protecting the compressed data.

D. Error Resilience of Decompressed Data

When the codeword is stored in memory, four cases that include all possible error scenarios are illustrated to show the error resilience of the data after decompression.

Case 1: an error has affected a symbol mapping to one MFP. As per steps 1 and 2 in the proposed mapping implementation, this incorrect symbol should have been moved from {S1} to {S2} that consists of symbols having distance-1 from {S1}. This is valid because a single bit error that is considered in this paper, changes the distance with a magnitude of 1. In this case, the error is detected and corrected, and the correct MFP is recovered during decompression.

Case 2: an error has affected a symbol mapping to one LFP considered in step 3 of the proposed mapping scheme. Similar to case 1, this symbol should have been changed from {S3} to {S2}, {S4} or {S5} if there are some spare symbols left. If it belongs to {S2}, the error can be detected but a miscorrection occurs, because it is decoded as an MFP during decompression. If it belongs to {S4}, the error cannot be detected. If it belongs to {S5}, the error can be detected but may not be corrected, because a different LFP may be output. However, most of the patterns associated to the symbols of a compressed data are always MFPs, therefore the case in which an error affects one LFP has a smaller probability of occurrence than case 1.

Case 3: an error has affected a symbol mapping to one LFP considered in step 4 of the proposed mapping scheme, but it still belongs to {S4}, so the error cannot be detected. However, this case also has a small probability of occurrence.

Case 4: an error has affected a symbol mapping to one LFP considered in step 4 of the proposed mapping scheme, and it is changed from {S4} to {S5}. In this case, the error can be detected, because a spare symbol of {S5} is found, but it may not be corrected in some cases, because a random pattern will be output during decompression. Again, this case has a small probability of occurrence.

Overall, the proposed scheme can detect and correct most of the errors (i.e., those affecting a symbol mapping to an MFP, which are more likely to occur). As for some errors that cannot be detected or corrected, their impact is significantly reduced by utilizing an improved mapping; this will be presented next.

E. Improved Mapping Scheme

To reduce the impact of uncorrectable errors, the proposed mapping scheme can be improved by converting global errors into local errors because they have a smaller impact on data integrity than global errors. This is performed by assigning a number of symbols to data patterns with the same sequence length, which has been initially discussed in [3] in the transmission of compressed data over a communication channel. To further explore this approach, in particular, the following conditions are employed: i) when assigning symbols {S3} for LFPs in step 3 of the proposed mapping scheme, each symbol that has distance-2 from a symbol in {S1} (i.e., assigned for one MFP) is assigned for one LFP with the same length as the related MFP; ii) in step 4 of the proposed mapping scheme to assign symbols {S4}, data patterns with the same length are mapped to symbols with at least distance-1 and patterns with different lengths are mapped to symbols with at least distance-2; iii) if {S5} exists and one of the spare symbols is received, then it is decoded as a random pattern related to a symbol with distance-1 from it. These conditions guarantee that once a miscorrection occurs, an incorrect pattern with the same length as the correct pattern is provided as output in most cases, so only causing a local error with no additional consequences.

Moreover, as per the method of generating Tunstall codes reviewed in Section II-A, patterns predicted in the same iteration have the same length and they only differ in the last element. This property can be exploited to further reduce the impact of local errors; by assigning distance-1 symbols for the patterns only differing in the last element, local errors then only affect a single element, likely having a negligible impact.

F. Example

An example shown in Fig. 6 is used to discuss the proposed scheme in more detail. In this case, a dataset that consists of elements A, B, C is considered again; an 8-bit Tunstall

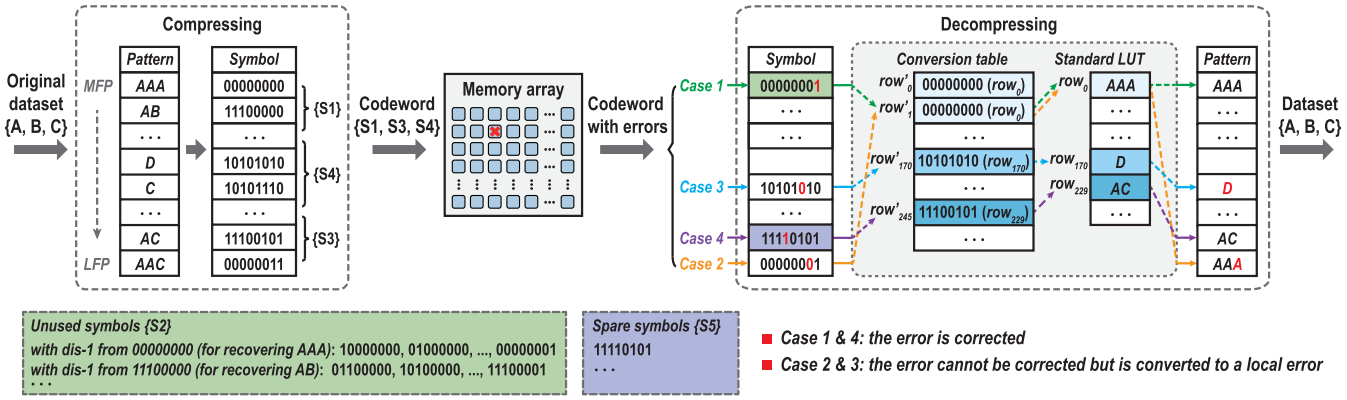

Fig. 6. Example to illustrate the proposed scheme (incorrect symbols or patterns due to memory errors are marked in red).

code is utilized for compression. By employing the proposed compression process, the following steps are countered:

Step 1: 00000000 is assigned for the first MFP, then 11100000 that has distance-3 from 00000000, is assigned for the second MFP, and so on; these symbols form {S1}.

Step 2: symbols with distance-1 from 00000000 (i.e., 1000000, 01000000, ..., 00000001), from 11100000 (i.e., 01100000, 10100000, ..., 11100001), and from all other symbols as assigned in step 1 are not used for any pattern; they are reserved in {S2}.

Step 3: 00000011 that has distance-2 from 00000000, is assigned for one LFP that has the same length with the first MFP, and 11100101 that has distance-2 from 11100000, is assigned for another LFP that has the same length with the second MFP, and so on; these symbols form {S3}.

Step 4: assign the remaining symbols that have at least distance-1 from each other for patterns with the same length (e.g., 10101010 for pattern D and 10101110 for pattern C) and those have at least distance-2 for patterns with different lengths, until all patterns are mapped. These symbols form{S4}; {S5} consists of the spare symbols (if any). Once the above steps are completed, the codeword that is the compressed data is obtained and stored in the memory.

The decompression process is implemented by using the conversion table and standard LUT. If a symbol belonging to {S1} is found in the received codeword, the corresponding MFP is recovered; this case should be error free, because all symbols for the MFP have distance-3 with each other and all symbols with distance-1 from them have also been reserved for unused, so no single error can change a symbol to one of {S1}.

Instead, for a symbol belonging to other sets, an error may have happened. If the symbol is found in {S2} as consisting of unused symbols, then it should have been affected by an error; moreover, it must be associated to an MFP or an LFP in the error-free case, because the symbols for these patterns have a distance-1 from {S2}. This is illustrated for cases 1 and 2 in Fig. 6; an error changes the original symbol 00000000 associated to an MFP AAA or the original symbol 00000011 associated to an LFP AAC, to 00000001 in {S2}. In these cases, the corresponding MFP is provided as output, because it has a larger probability to be corrupted. This permits error correction for case 1: once the incorrect symbol 00000001 is received, row'_1 of the conversion table is located and its content 00000000 that refers to row_0 of the standard LUT is established; thus, the error is corrected, and the MFP AAA is correctly recovered. For case 2, outputting AAA results in a miscorrection, because the corrupted pattern was an LFP AAC. However, this is only a local error, because only one element is corrupted between AAC and AAA; this is achieved due to the additional constraint in step 3 of the compression process analyzed in the improved scheme, which guarantees that the output MFP has the same length as the affected LFP.

If the received symbol belongs to {S3} or {S4} (e.g., symbol 10101010 in case 3 of Fig. 6), it is decoded as the corresponding pattern (i.e., pattern D); even if an error has occurred and cannot be detected, it only affects one element of the decompressed dataset (e.g., changing pattern C to D) as per the improved design discussed previously. If a spare symbol in {S5} (e.g., symbol 11110101 in case 4 of Fig. 6) is found, an error must have changed another symbol with distance-1 from it (e.g., symbol 11100101 has been changed), so the error can be detected. In this case, the pattern related to a random symbol with distance-1 from this spare symbol is output; however in this way, the error may be corrected (e.g., pattern AC in Fig. 6), or a miscorrection may occur (e.g. another pattern with two elements instead of AC is output in Fig. 6).

IV. Evaluation

In this section, the effectiveness of the proposed scheme is evaluated by covering three widely used large ML datasets. Traditional error protection schemes including Single Parity (SP) check and Single Error Correction (SEC) codes are used for comparison. Moreover, the case for uncompressed data with different error protection schemes is also evaluated to show the advantage of data compression by using the proposed scheme.

A. Experimental Setup

1) Datasets: The training set of three datasets taken from a public repository [21] are used: Record Linkage Comparison Patterns [22] (3,200,000 samples with 81 different numerical features), Census Income (520,976 samples with 99 different

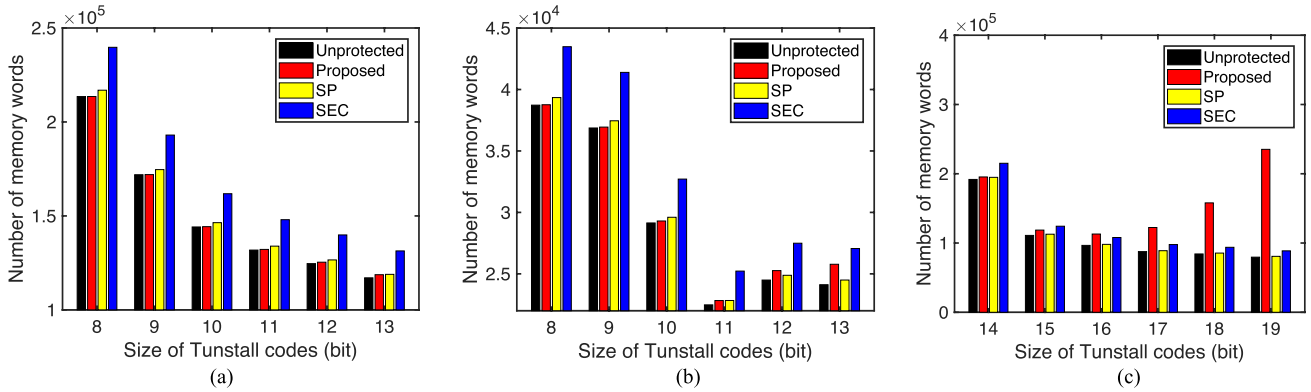

Fig. 7. Total number of memory words (including that for the decompression tables) required by different schemes for (a) Record Linkage Comparison Patterns; (b) Census Income; (c) Bank Marketing dataset.

TABLE I
DATA COMPRESSION FOR DIFFERENT DATASETS

Dataset	Tunstall code*	Compre. ratio	# of possible patterns	# of used patterns	# of patterns can be protected
Record Linkage Comparison Patterns ($N = 81$)	8-bit	26.69%	241	213	5
	9-bit	21.48%	481	380	14
	10-bit	18.00%	961	634	39
	11-bit	16.46%	2,001	964	98
	12-bit	15.54%	4,081	1456	220
	13-bit	14.58%	8,161	2061	469
Census Income ($N = 99$)	8-bit	29.72%	197	190	8
	9-bit	28.29%	295	206	34
	10-bit	22.32%	981	459	56
	11-bit	17.19%	1,961	519	139
	12-bit	18.75%	4,091	416	306
	13-bit	18.42%	8,135	636	581
Bank Marketing ($N = 6315$)	14-bit	50.00%	12,629	8,632	553
	15-bit	28.43%	31,571	12,984	1,318
	16-bit	24.37%	63,141	16,123	3,088
	17-bit	21.83%	126,281	17,631	6,673
	18-bit	20.58%	258,875	21,566	13,183
	19-bit	19.52%	524,063	18,439	26,623

* The size of Tunstall codes starts with the minimum value as per Eq. (1).

numerical features), and Bank Marketing [23] (1,519,152 samples with 6315 different numerical features).

2) Compression Codes: Tunstall codes with different size (i.e., n) are utilized for data compression. A numerical element is represented by 16 binary bits: the most significant bit is for the sign, seven bits for the integer part and eight bits for the decimal part. Table I shows the compression information of employing Tunstall codes with different sizes, including the compression ratio, the number of possible patterns (as per Eq. (1)), the number of patterns used in the compression process, and the number of patterns that can be theoretically protected by employing the proposed scheme. As per Table I, the number of bits n of the Tunstall code symbols increases, so does the compression ratio and also the fraction of symbol values that are not used; thus, the proposed scheme tends to be resilient to more errors for larger values of n. Note that both existing SP/SEC schemes and the proposed schemes do not change these compression ratios or the efficiency of Tunstall codes; however, an additional memory overhead may be incurred. Such evaluation is pursued in the subsequent subsection.

3) Error Injection: Single bit errors, which are the most common class of errors in a memory, are considered as error model in the experiments. The error position is randomly selected with a uniform distribution and then the related bit is flipped. 100,00 trials of error injection were performed in different schemes to evaluate the average impact.

4) Evaluation Metrics: Prior to evaluating the hardware overhead required for different schemes in subsequent subsections, the main hardware modules in each case are summarized next.

- *Unprotected scheme:* a memory storing compressed data and a memory storing a LUT for decompression.
- *SP/SEC scheme:* a memory storing compressed data and ECC bits, a memory storing a LUT for decompression, an encoder circuit for generating the ECC bits, and a decoder circuit for detecting or correcting errors.
- *Proposed scheme:* a memory storing compressed data, a memory storing a standard LUT and an additional conversion table for decompression.

Since memory accounts for the largest part of the area/power of the entire implementation in all cases, the memory size that is proportional to these overheads is evaluated in Section IV-B; moreover, the error resilience that can be achieved by utilizing such memory size for different schemes is also compared. Then, Section IV-C evaluates the latency introduced to compression and decompression; since the memory read/write latency is the same in all schemes, only the additional latency required for performing error protection is evaluated and compared.

B. Error Resilience & Memory Size

The percentage of errors that are handled in different schemes is given in Table II, in which the memory overhead required for storing the compressed data and the decompression tables in different cases is also compared in terms of number of words (assuming that a memory with 64 bits/word is utilized).

Table II show that the proposed scheme can detect/correct most of the errors and the percentage of detected errors is larger than corrected in all cases. This is expected because only errors affecting symbols in {S1} can be corrected, while those affecting {S3} or generating symbols in {S5} can also be

TABLE II
COMPARISON FOR ERROR RESILIENCY AND MEMORY SIZE FOR DIFFERENT SCHEMES

Dataset	Scheme	Data compressed*				Data uncompressed		
		% of errors detected	% of errors corrected	Memory words required for compressed data	Memory words required for decompression tables	% of errors detected	% of errors corrected	Memory words required
Record Linkage Comparison Patterns	Unprotected	0	0	116,642 to 213,481	27 to 419	0	0	800,000
	Proposed	87.21% to 95.51%	84.12% to 90.34%	116,642 to 213,481	59 to 2,083	-	-	-
	SP	100%	0	118,493 to 216,869	27 to 419	100%	0	812,699
	SEC	100%	100%	130,966 to 239,697	27 to 419	100%	100%	898,246
Census Income	Unprotected	0	0	22,386 to 38,705	24 to 130	0	0	130,244
	Proposed	78.37% to 99.88%	76.40% to 97.88%	22,386 to 38,705	56 to 1,794	-	-	-
	SP	100%	0	22,742 to 39,320	24 to 130	100%	0	132,312
	SEC	100%	100%	25,135 to 43,459	24 to 130	100%	100%	146,239
Bank Marketing	Unprotected	0	0	74,130 to 189,925	1,889 to 5,475	0	0	379,788
	Proposed	94.50% to 99.98%	88.62% to 99.14%	74,130 to 189,925	5,473 to 161,123	-	-	-
	SP	100%	0	75,307 to 192,940	1,889 to 5,475	100%	0	385,817
	SEC	100%	100%	83,233 to 213,249	1,889 to 5,475	100%	100%	426,429

* Different Tunstall codes as per Table I are used for data compression, which generates a range of values for some metrics.

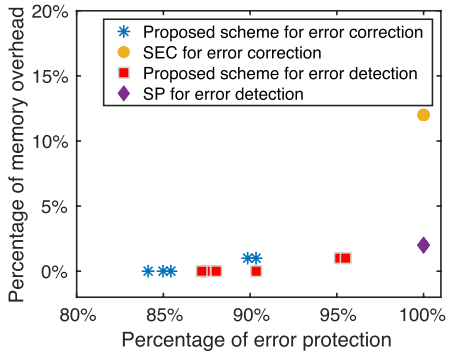

Fig. 8. Percentage of error protection versus the extra memory overhead for Record Linkage Comparison Patterns dataset.

detected. As for traditional schemes, all errors can be detected (corrected) by employing the SP check (SEC codes); however, these schemes require additional memory to store the ECC bits.

As per Table II, when considering the entire memory size, the proposed scheme incurs in a negligible overhead for small Tunstall codes; the overhead is significant when a large Tunstall code is employed, but this is still lower than SEC in most cases. Fig. 7 (a) shows the total number of memory words required for the Record Linkage Comparison Patterns dataset in different cases; the proposed scheme (that can correct more than 84.12% errors) only requires a small memory overhead compared with the unprotected scheme, and smaller than the SP check as well as the SEC. This is also the case for the Census Income dataset (Fig. 7 (b)) when n is smaller than 11; for this dataset, the proposed scheme requires a larger memory overhead than the SP scheme when larger Tunstall codes are used, it can still correct 76.40% to 97.88% errors, while of course SP cannot correct any. Moreover, the SEC code still incurs in the largest memory size for all cases. For the Bank Marketing dataset (Fig. 7 (c)), the proposed scheme is not efficient when large Tunstall codes are used, because the required conversion table is also large; however, for the other cases, it corrects more than 90% errors with a lower memory size than the SEC code.

To better show the overhead for error resilience in terms of memory size, results for the Record Linkage Comparison Patterns dataset are plotted in Fig. 8 as an example; a stronger error resilience tends to be achieved at the cost of a larger memory size. Therefore, this plot shows that the proposed scheme is very attractive for resource-constrained platforms because it requires an extremely low overhead while correcting most errors, even though not all. Moreover, error protection can be implemented as part of the translation in a single memory access during the decompression process as discussed previously; hence, the proposed scheme is also more efficient in terms of latency than ECC techniques that require an additional decoding process to correct errors.

It is of interest to also consider the case of uncompressed data for comparison purposes. The comparison results are also given in Table II. For example, compared with compressed data protected by the proposed scheme, an additional 2.77 to 5.96 times (3.17 to 6.70 times) the memory overhead is introduced by the SP (SEC) scheme to detect 4.49% to 12.79% more errors (correct 9.66% to 15.88% more errors) for the uncompressed Record Linkage Comparison Patterns dataset.

Next, Table III shows the impact of uncorrectable errors in the unprotected and the proposed schemes, i.e., the percentage of uncorrectable global errors and local errors (the results for the SP scheme are the same as for the unprotected schemes, because no error can be corrected). As per Table III, most of the errors in the unprotected memory have a global impact on the decompressed data; in the proposed scheme, the percentage of uncorrectable errors is significantly reduced, and these errors tend to have a local rather than a global impact by carefully considering the mapping in the improved design. Finally, for the SEC scheme, the rather large memory overhead is incurred even though all errors can be corrected.

Overall, the proposed scheme can correct most errors with an extremely small memory overhead; even if some errors cannot be corrected, by improving the mapping between data patterns and Tunstall symbols, local errors are dominant, thus the impact of errors is reduced. These advantages make the proposed scheme more attractive to resource-limited platforms. It should be also noted that under a constant error rate, a large

TABLE III
PERCENTAGE OF ERRORS THAT CANNOT BE CORRECTED IN DIFFERENT SCHEMES

Dataset	Tunstall code	Unprotected			Proposed		
		Total % of uncorrectable errors	% of uncorrectable global errors	% of uncorrectable local errors	Total % of uncorrectable errors	% of uncorrectable global errors	% of uncorrectable local errors
Record Linkage Comparison Patterns	8-bit	100%	68.96%	31.04%	12.64%	5.83%	6.81%
	9-bit	100%	79.91%	20.09%	15.02%	7.01%	8.01%
	10-bit	100%	83.07%	16.93%	14.57%	5.49%	9.08%
	11-bit	100%	75.87%	24.13%	15.88%	7.58%	8.30%
	12-bit	100%	79.29%	20.71%	9.66%	3.89%	5.77%
	13-bit	100%	76.11%	23.89%	10.14%	3.83%	6.31%
Census Income	8-bit	100%	56.69%	43.31%	22.63%	10.37%	12.26%
	9-bit	100%	62.03%	37.97%	15.53%	4.22%	11.31%
	10-bit	100%	71.16%	28.84%	23.60%	8.26%	15.34%
	11-bit	100%	79.13%	20.87%	19.47%	3.28%	16.19%
	12-bit	100%	68.54%	31.46%	8.35%	0.12%	8.23%
	13-bit	100%	72.88%	27.12%	2.12%	0.03%	2.09%
Bank Marketing	14-bit	100%	37.54%	62.46%	4.03%	0.98%	3.05%
	15-bit	100%	81.97%	18.03%	6.67%	3.22%	3.45%
	16-bit	100%	83.11%	16.89%	11.38%	4.99%	6.39%
	17-bit	100%	85.99%	14.01%	9.10%	2.62%	6.48%
	18-bit	100%	79.54%	20.46%	7.52%	1.89%	5.63%
	19-bit	100%	79.04%	20.96%	0.86%	0.03%	0.83%

TABLE IV
ADDITIONAL LATENCY (*ns*) INCURRED FOR ERROR PROTECTION

Scheme	In compression	In decompression
Proposed	0	0.03 to 0.06
SP	0.43	0.47
SEC	0.53	0.73

size memory may have more cells to be affected by the errors. Hence a smaller memory size required by the proposed scheme may also have this inherent advantage, i.e., a larger probability of having at most a single error in the entire memory.

C. Latency for Error Protection

To show the efficiency of the proposed scheme in terms of also latency required for error protection, the additional latency introduced to data compression/decompression by different protection schemes is evaluated and compared next. For the proposed technique, such overhead is only required during decompression for accessing the conversion table; for SP and SEC, an additional encoding latency is required once data is compressed, and an additional decoding latency is required prior to data decompression (Fig. 4 (b)).

The latency overhead is evaluated by implementing different schemes at the RTL level and automatically mapping the design to a 32 nm technology library [24] using the Synopsys Design Compiler. The constraint of delay optimization is set during synthesis to evaluate the best timing performance that can be achieved; this is performed for all circuits for fair comparison. Since the memory used for storing the compressed data is assumed to have 64 bits per word, an SP (SEC) requires 1 (7) parity bit to protect these data bits [12]. As per the evaluation results given in Table IV, the proposed scheme incurs in a significantly smaller latency for error protection compared to existing techniques; moreover, such advantage will become evident when the compressed data has a large size and needs to be stored in a large number of memory words, because error protection is performed word by word and all latency results reported in Table IV must be multiplied by the number of memory words. A pipelined scheme can be implemented to reduce the impact of ECC decoding on the decompression delay for the entire system; however, it still may increase either the delay per clock cycle or the number of clock cycles, which can also degrade the operating frequency. In any case, even when the use of pipelining can reduce the impact of the ECC on the decompression latency, the area benefits of the proposed scheme still make it of interest for resource constrained applications. Compared to the SP/SEC, the proposed scheme also does not require the encoding and decoding circuits in addition to any parity bits; hardware metrics (such as for area and power) required for these circuits are not further discussed because they are negligible compared to the memory overhead.

V. APPLICATIONS

The proposed error-resilient data compression scheme can be beneficial in many applications that process a large volume of data and perform compression on the stored data to reduce storage requirements, such as ML [25], image/video processing [26], digital signal processing [27], wireless systems [28], and automatic test equipment [29]. Next, two ML applications are taken as examples to evaluate the effectiveness of the proposed scheme. In ML inference, the training data samples (like for the k Nearest Neighbors (kNNs) classifier), or the parameters of a trained model of large size are often stored in a compressed version [30]; this also applies to ML training because the training set usually has a very large size. In all these cases, the compressed data is possibly accessed many times for frequently performing classification tasks or conducting the complex training for a single or several combined algorithms [31]. An error (especially a global error) in the compressed data can result in incorrect classification

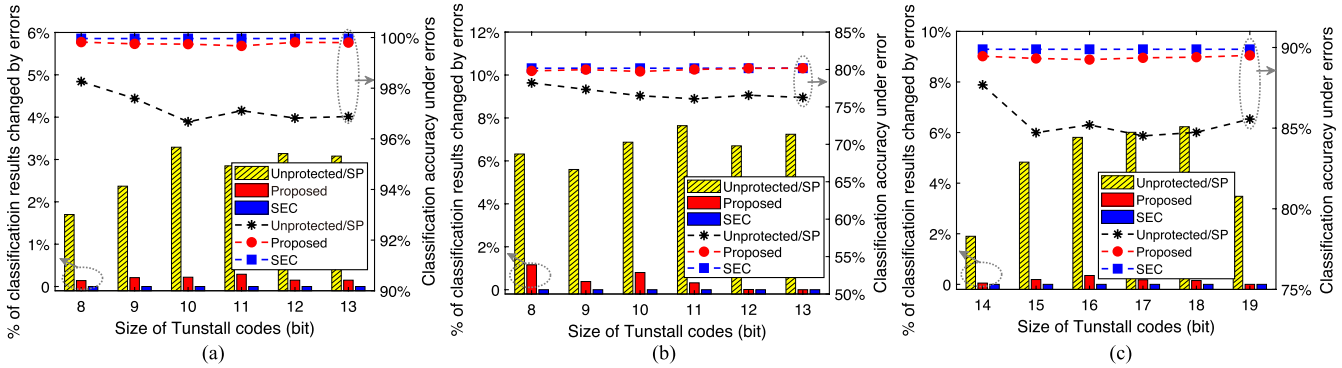

Fig. 9. Impact of single errors on kNNs in different schemes for (a) Record Linkage Comparison Patterns (with optimal $k = 3$ to achieve a classification accuracy of 99.96% in the error-free case); (b) Census Income (with optimal $k = 19$ to achieve a classification accuracy of 80.17% in the error-free case); (c) Bank Marketing dataset (with optimal $k = 17$ to achieve a classification accuracy of 89.89% in the error-free case).

results or hyperparameters. Therefore, an efficient error-resilient data compression scheme like the proposed approach is of interest, especially for hardware-constrained platforms or systems with limited computing resources.

Next, the inference process of a kNNs classifier [16] and the training process of a Stochastic Computing-based Neural Network (SC-NN) classifier [17] are considered, because in both cases, the training set that is compressed and stored in a memory, needs to be accessed. The three ML datasets considered in Section IV are used again. The following evaluation metrics are assessed for each application.

i) Since the paper focuses on memory errors occurred in hardware, the percentage of classification results changed are evaluated to reflect the impact of errors in different schemes at a higher system level, i.e., the change on each final result computed by using the hardware with errors. Moreover, the classification accuracy of different models in each scheme is also evaluated to assess the impact of errors on the classification performance.

ii) Moreover, the memory size that is proportional to the hardware area and power required for different schemes, is also evaluated and compared.

A. Application 1: kNNs Inference

kNNs is one of the simplest but powerful classifiers; the training set of a kNNs classifier is used to determine the optimal number of the nearest neighbors (i.e., the value of k) and then stored in memory for inference. The class of a coming data sample is predicted by performing a majority voting among the classes of its neighbors once its distance to all stored data samples is computed. Therefore, errors on the stored features and labels of training samples may modify the classification result, because the error may change one or more neighbors.

Before evaluating the impact of different error-resilient data compression schemes on the classification results of the kNNs classifier, the value of the optimal k and associated classification accuracy are determined first as per the error-free data. Then, errors are injected (also by using the method explained in Section IV-A) in the memory that stores the compressed training set unprotected or protected by different techniques; the decompressed data is then used to evaluate the classification results for the testing set.

Fig. 9 shows the percentage of classification results changed by a single error and the classification accuracy under errors in different schemes. Results show that when the traditional SEC codes are employed, no classification results are changed because errors can be fully corrected; however, this is achieved by requiring 1.11 to 1.12 times the original memory size required to store the unprotected compressed data (as evaluated in Section IV-B and Fig. 7). When the proposed scheme is employed, the percentage of classification results changed due to an error is extremely low (i.e., below 0.38% in most cases, and in some cases 0) and it only requires a small memory overhead in most cases. For a few cases (i.e., when large Tunstall codes are employed), the proposed scheme has no impact on the classification results, but it incurs in large memory size, so an SEC code is more efficient for these cases.

B. Application 2: SC-NN Training

An NN is another widely used ML technique to perform classification over a wide range of applications [32]. It includes one input layer, one or more hidden layers and one output layer, consisting of neurons per layer. By providing the input neurons to the NN, the features are then activated by using the activation function (e.g., tanh or ReLU [33]) in the next layers; this offers the NN a better classification accuracy than for example kNNs in most cases. Since NNs usually deal with large volumes of data, the required hardware (e.g., memory and arithmetic logics) may not be acceptable in resource-constrained platforms; thus in this case, data compression that reduces the memory and stochastic computation that reduces the arithmetic circuits complexity are attractive for implementing NNs. In stochastic computation, the basic arithmetic operation of NNs (e.g., multiplication, addition and activation functions) can be implemented at a significantly low complexity, for example only one AND gate is required to perform multiplication in a unipolar SC design. Additionally, stochastic computation inherently provides error protection during computation, so efficient error protection is usually required for only the memory that stores the training dataset and the NN parameters.

Next, an SC-NN of [17] is used to evaluate the impact of errors on the classification result. Errors are injected in the

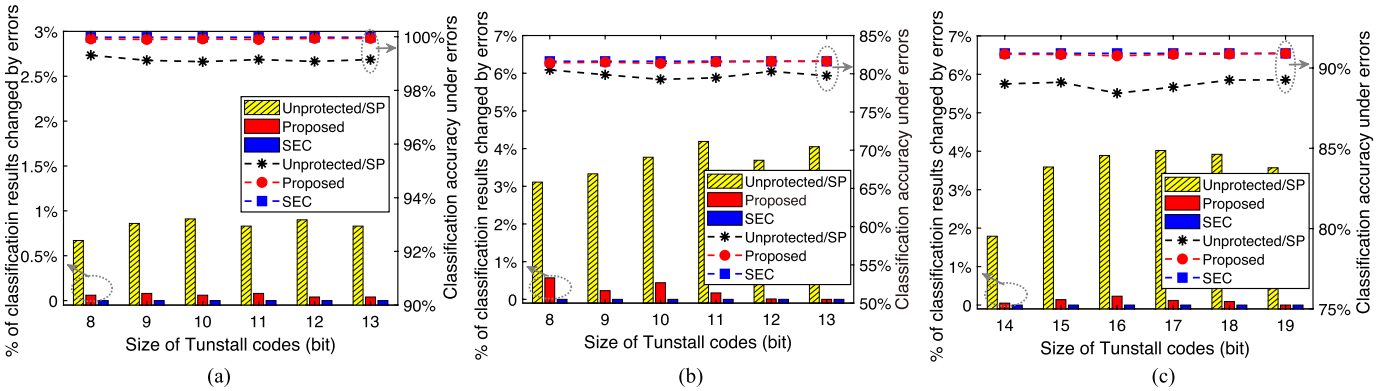

Fig. 10. Impact of single errors on SC-NNs in different schemes for (a) Record Linkage Comparison Patterns (with a hidden layer of 16 neurons to achieve a classification accuracy of 99.98% in the error-free case); (b) Census Income (with a hidden layer of 31 neurons to achieve a classification accuracy of 81.63% in the error-free case); (c) Bank Marketing dataset (with a hidden layer of 96 neurons to achieve a classification accuracy of 90.91% in the error-free case).

memory storing the compressed training set and the network is retrained; then the percentage of classification results of the testing set modified due to the error in different cases, as well as the classification accuracy of different schemes, are compared in Fig. 10. Results have the same trend as for the kNNs; the impact of errors on the classification results in the proposed and SEC schemes are similar or even the same (in particularly, close or equal to zero) in most cases, but SEC requires a larger memory overhead. Again in a few cases when large Tunstall codes are employed, an SEC code is more efficient because in these cases, the proposed scheme requires a large conversion table that increases the memory size.

Overall, compared with existing memory protection techniques, the proposed scheme is more attractive when these classifiers are implemented in resource-constrained platforms (e.g., Internet of Things devices) because it can achieve a nearly complete error protection at a lower hardware overhead.

VI. Conclusion and Future Work

In this paper, an efficient error-resilient data compression scheme using variable-to-fixed length codes (namely Tunstall codes), is proposed against single bit-flip memory errors. Since the 2^n symbols of the n-bit Tunstall coding are usually not fully assigned during compression, the mapping between Tunstall symbols and data patterns in the proposed scheme has been carefully designed for error resilience of the memory storing the compressed data. Moreover, an enhanced design has been presented to reduce the impact of uncorrectable errors.

Compared to existing ECC techniques that are widely used to provide error protection for memories, the proposed technique does not need to store any ECC redundancy bits in each memory word; it only needs a conversion table for decompression. The scheme has been evaluated for several ML datasets. The results have shown that the proposed scheme can correct up to 97.88% errors, with almost the same memory size as the unprotected scheme in most cases; by comparison, the Single Error Correction (SEC) code corrects 2.12% more errors, but at an overhead of 1.12 times the memory size. However, for larger Tunstall codes, SEC is more efficient, because the conversion table required by the proposed scheme is also large. In addition to the memory size requirement, it has also been shown that the proposed scheme is attractive in terms of latency for error resiliency, because it does not require any ECC encoding/decoding process, but it controls errors as part of the translation in a single memory access during the decompression process. Two ML classifiers (i.e., the kNNs and SC-NNs) have been utilized to evaluate the impact of the proposed scheme on the classification results. Results show an extremely low (nearly zero in few cases) impact on the classification results in the proposed scheme for all datasets considered. Therefore, when Tunstall codes with smaller than 16-bits symbols are employed for data compression, the proposed scheme is best suitable for resource-constrained platforms, because it provides an almost complete error resilience at a significant lower hardware overhead compared with existing techniques.

Even though single bit-flip errors are considered in this paper, the proposed scheme would also be applicable to handle single stuck-at errors, as another common type of errors in both conventional and emerging memories. Future works could be pursued in the following aspects: (i) further extend the proposed scheme for different variants of Tunstall codes (such as the codes of [34]), and (ii) efficient protection schemes for multiple bit errors are also of interest, because ECCs with a powerful correction ability typically incur in extremely large memory redundancy and decoding latency.

References

[1] K. Saygood, *Introduction to Data Compression*, 4th ed. Newnes, MA, USA: Morgan Kaufmann, 2017.

[2] Y. Xie, W. Wolf, and H. Lekatsas, "Code compression for embedded VLIW processors using variable-to-fixed coding," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 14, no. 5, pp. 525–536, May 2006.

[3] J. A. Briffa and V. Buttigieg, "Investigation of the error performance of Tunstall coding," in *Proc. IEEE Int. Symp. Inf. Theory*, Aug. 1998, p. 202.

[4] H. Hashempour, L. Schiano, and F. Lombardi, "Error-resilient test data compression using Tunstall codes," in *Proc. 19th IEEE Int. Symp. Defect Fault Tolerance VLSI Syst.*, Oct. 2004, pp. 316–323.

[5] E. Ibe, H. Taniguchi, Y. Yahagi, K. Shimbo, and T. Toba, "Impact of scaling on neutron-induced soft error in SRAMs from a 250 nm to a 22 nm design rule," *IEEE Trans. Electron Devices*, vol. 57, no. 7, pp. 1527–1538, Jul. 2010.

[6] R. C. Baumann, "Soft errors in advanced computer systems," *IEEE Des. Test. Comput.*, vol. 22, no. 3, pp. 258–266, May/Jun. 2005.

[7] R. C. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Trans. Device Mater. Rel.*, vol. 5, no. 3, pp. 305–316, Sep. 2005.

[8] J. L. Autran et al., "ASTEP (2005–2015): Ten years of soft error and atmospheric radiation characterization on the Plateau de Bure," *Microelectron. Rel.*, vol. 55, nos. 9–10, pp. 1506–1511, Aug. 2015.

[9] G. Li et al., "Understanding error propagation in deep learning neural network (DNN) accelerators and applications," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2017, pp. 1–12.

[10] P. Koopman and M. Wagner, "Autonomous vehicle safety: An interdisciplinary challenge," *IEEE Intell. Transp. Syst. Mag.*, vol. 9, no. 1, pp. 90–96, Jan. 2017.

[11] C. L. Chen and M. Y. Hsiao, "Error-correcting codes for semiconductor memory applications: A state-of-the-art review," *IBM J. Res. Develop.*, vol. 28, no. 2, pp. 124–134, Mar. 1984.

[12] S. Lin and D. J. Costello, *Error Control Coding*, 2nd ed. Englewood Cliffs, NJ, USA: Prentice-Hall, 2004.

[13] A. Dutta and N. A. Touba, "Multiple bit upset tolerant memory using a selective cycle avoidance based SEC-DED-DAEC code," in *Proc. 25th IEEE VLSI Test Symmposium (VTS)*, May 2007, pp. 349–354.

[14] S. Liu, P. Reviriego, and F. Lombardi, "Codes for limited magnitude error correction in multilevel cell memories," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 67, no. 5, pp. 1615–1626, May 2020.

[15] M. Augustin, M. Gossel, and R. Kraemer, "Reducing the area overhead of TMR-systems by protecting specific signals," in *Proc. IEEE 16th Int. Line Test. Symp.*, Jul. 2010, pp. 268–273.

[16] S. Zhang, X. Li, M. Zong, X. Zhu, and R. Wang, "Efficient kNN classification with different numbers of nearest neighbors," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 29, no. 5, pp. 1774–1785, May 2018.

[17] Y. Liu, S. Liu, Y. Wang, F. Lombardi, and J. Han, "A stochastic computational multi-layer perceptron with backward propagation," *IEEE Trans. Comput.*, vol. 67, no. 9, pp. 1273–1286, Sep. 2018.

[18] M. Patel et al., "Bit-exact ECC recovery (BEER): Determining DRAM on-die ECC functions by exploiting DRAM data retention characteristics," in *Proc. 53rd Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2022, pp. 282–297.

[19] S. Swami and K. Mohanram, "Reliable nonvolatile memories: Techniques and measures," *IEEE Design Test*, vol. 34, no. 3, pp. 31–41, Jun. 2017.

[20] Intel. *Safeguard Important Data With ECC Memory*. [Online]. Available: https://www.intel.com/content/www/us/en/processors/xeon/the-value-of-ecc-memory-for-servers.html

[21] D. Dua and C. Graff, "UCI machine learning repository," School Inf. Comput. Sci., Univ. California, Irvine, CA, USA, 2019. [Online]. Available: http://archive.ics.uci.edu/ml

[22] M. Sariyar, A. Borg, and K. Pommerening, "Controlling false match rates in record linkage using extreme value theory," *J. Biomed. Informat.*, vol. 44, no. 4, pp. 648–654, Aug. 2011.

[23] S. Moro, P. Cortez, and P. Rita, "A data-driven approach to predict the success of bank telemarketing," *Decis. Support Syst.*, vol. 62, no. 1, pp. 22–31, Jun. 2014.

[24] (2021). *Synopsys 14 nm, 32/28 nm and 90 nm Generic Libraries*. [Online]. Available: https://www.synopsys.com/community/university-program/teaching-resources.html

[25] D. Han, J. Lee, J. Lee, and H. Yoo, "A low-power deep neural network online learning processor for real-time object tracking application," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 66, no. 5, pp. 1794–1804, May 2018.

[26] K. Pexaras, I. G. Karybali, and E. Kalligeros, "Optimization and hardware implementation of image and video watermarking for low-cost applications," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 66, no. 6, pp. 2088–2101, Jun. 2019.

[27] B. Sanches and W. Van Noije, "An optimized radiation tolerant baseline correction filter for HEP using AI methodologies," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 68, no. 5, pp. 1789–1799, May 2021.

[28] S. Mehfuz and U. Tiwari, "A Tunstall based lossless compression algorithm for wireless sensor networks," in *Proc. Annu. IEEE India Conf. (INDICON)*, Dec. 2015, pp. 1–4.

[29] S. Seo, Y. Lee, and S. Kang, "Tri-state coding using reconfiguration of twisted ring counter for test data compression," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 2, pp. 274–284, Feb. 2016.

[30] C. Chen, Z. Wang, X. Chen, J. Lin, and M. M. S. Aly, "Efficient Tunstall decoder for deep neural network compression," in *Proc. 58th ACM/IEEE Design Autom. Conf. (DAC)*, Dec. 2021, pp. 1021–1026.

[31] S. B. Kotsiantis, I. D. Zaharakis, and P. E. Pintelas, "Machine learning: A review of classification and combining techniques," *Artif. Intell. Rev.*, vol. 26, no. 3, pp. 159–190, Nov. 2006.

[32] S. Das, A. Dey, A. Pal, and N. Roy, "Applications of artificial intelligence in machine learning: Review and prospect," *Int. J. Comput. Appl.*, vol. 115, no. 9, pp. 31–41, Apr. 2015.

[33] E. A. M. Shenouda, "A quantitative comparison of different MLP activation functions in classification," in *Proc. 3rd Int. Conf. Adv. Neural Netw.*, 2006, pp. 849–857.

[34] S. T. Klein and D. Shapira, "On improving Tunstall codes," *Inf. Process. Manag.*, vol. 47, no. 5, pp. 777–785, Sep. 2011.