Distributed and Parallel Databases
https://doi.org/10.1007/s10619-019-07265-y

Multi-model Z-compression for high speed data streaming and low-power wireless sensor networks

Xiaofei Cao[1] · **Sanjay Madria**[1] · **Takahiro Hara**[2]

Abstract
Wireless Sensor Networks (WSNs) have significant limitations in terms of available bandwidth and energy. The limited bandwidth in WSNs can cause delays in message delivery, which does not suit the real-time sensing applications such as a gas leak. Moreover, in such applications, there are multi-modal sensors whose values like temperature, gas concentration, location, and CO_2 levels must be transmitted together for correct and timely detection of a gas leak. In this paper, we propose a novel Z-order based data compression scheme, Z-compression, to reduce energy consumption and save bandwidth without increasing message delivery latency. Instead of using the popular Huffman tree style based encoding, the Z-compression uses Z-order encoding to map the multidimensional sensing data into one-dimensional binary stream transmitted using a single packet. Our experimental evaluations using different real-world datasets show that the Z-compression has a much better compression ratio, energy, and bandwidth savings than known schemes like LEC, Adaptive-LEC, FELACS, and TinyPack for multi-modal sensor data. Through the extensive experiments, we show that Z-compression is suitable for real-world sensing applications requiring high-stream rate WSNs and delay-tolerant low-power listening WSNs. In high-stream rate WSNs, the Z-compression can save bandwidth and increases the throughput. In low-power listening WSNs, by concatenating the Z-compressed data at selected reporting nodes, we can reduce the duty cycles of the nodes in WSNs, thus prolong the lifetime of the network, and still maintain the low distortion rate.

Keywords Sensor network · Data compression · Z-order

This research is partly supported by a NSF Grant CNS 1461914 and a DOE Grant P200A120110.

✉ Sanjay Madria
madrias@mst.edu

Extended author information available on the last page of the article

Published online: 29 March 2019

1 Introduction

Wireless sensor networks (WSNs) are being developed for a plethora of emerging applications in a wide range of disciplines. For example, there are near real-time sensor cloud applications [28] to perform multi-modal sensing tasks. Some military applications like tracking hostile objects or monitoring intruders use multiple sensing units to provide precise location and speed by applying multi-sensor data fusion. The unmanned vehicles [5] need GPS and accelerometer to locate themselves, and to track distance and height of objects using the camera, laser range meter or radar. By fusing these multi-modal sensor data, they can also predict the moving trajectory of the nearby objects. Similarly, the environmental monitoring applications need temperature, wind direction, humidity, CO_2 levels, etc. Many of these multi-modal sensor applications require data integrity (lossless data) as well as high-streaming rate, and therefore, cannot tolerate high latency due to limited bandwidth in WSNs. Since batteries are the typical power source for wireless sensors, the energy consumption is another primary constraint in the design of multi-modal WSNs. Many research efforts have shown that radio communication, including the data transmission and channel listening, is the predominant factor among all the energy consumption metrics of the WSNs.

The power model of Micaz [33] shows that the channel listening even consumes more power than the transmission. The duty cycle approaches are the most straightforward way to reduce the radio communication. The fundamental idea of these duty cycle schemes is to put the sensors to sleep periodically. When the sensors fall asleep, there is no radio communication at all which minimizes the energy consumption. We can divide these duty cycle approaches into two categories; synchronous and asynchronous. The synchronized duty cycle MAC protocols include [2,25,34] and asynchronous approaches include [3,23,37]. Among them, the low-power listening (LPL) scheme in TinyOS has proven its ability to reduce the duty cycles of the wireless sensor nodes. It works well for the applications that are without heavy loads. However, in LPL mode, sensors will wake up for a full period for sending messages. Thus, there is a need for lossless data compression algorithms which could reduce the size of multi-modal sensing data and thereby, decreasing the radio communication.

Sensor data aggregation approaches like [21,24] save energy and bandwidth by reducing the number of packets to be transmitted. However, they cannot guarantee the data integrity because of the lossy process and the outliers. Also, when aggregating, the outlier detection is expensive and introduce delays [30]. The model-based compression algorithms [14] such as APCA [16], PWLH [4], and SF [9] also have good compression ratios. However, they can not work with applications requiring lossless data as they approximate the data with temporal and spatial locality. The compressive sensing approaches like [19] have the drawback that the data may lose integrity thus, are not suitable for multi-modal lossless data compression.

Existing works [29,38,40] propose the lossless compression algorithms for sensor data using Huffman tree style coding that exploit the temporal locality of the data in WSNs. Instead of storing and transmitting the complete data, [17,29,38,40] use the difference in values, called delta value, between the two adjacent timestamp readings and usually, it needs fewer bits to represent a delta value. Based on the Shannon entropy theory [36] and the Huffman coding [12], the most frequent values are assigned a shorter

code than the less frequent values. Thus, if a dataset is drawn from a smaller set of values, the entropy will be smaller. However, the standard Huffman and adaptive Huffman [41] coding have larger overhead on RAM in storing the Huffman trees generated dynamically based on the frequency of the data. Also, to decode the compressed data, every node in the wireless sensor network needs to have a copy of the tree. However, as we know that nodes in WSNs have limited bandwidth and energy, synchronizing the Huffman tree among nodes is impractical. LEC and Adaptive-LEC [29,40] successfully adapted Huffman coding for their static initial code library which is a predefined Huffman tree. That way, WSNs do not require transmitting the entire Huffman tree. Similarly, the TinyPack [38] modified its initial code library based on LEC's and proposed algorithms adapting library to different types of sensor applications. The static initial code cannot always give the best performance because the distribution of the dataset can deviate from the optimized code tree. Further more, these schemes are designed to work for a single attribute value as the multi-dimensional sensing data can have different distributions for each attribute.

To address the problems discussed above and improve the applicability and compression ratios of existing works, we propose a Z-order [32] encoding based data compression scheme. The Z-order encoding called Z-compression can compress multi-modal sensing data at each leaf node as well as at the intermediate nodes efficiently in near real-time. The Z-compression algorithm can encode multi-modal sensor data like precipitation, water level, and wind speed (needed to detect a flood risk in a region) into a binary stream. Using our Z-compression algorithm in a WSN with a hierarchical topology [13], the nodes with limited bandwidth can tolerate higher-stream data rates coming from upstream nodes by concatenating compressed sensor data into the reduced number of packets which may be as large as permissible by the network protocol. The proposed Z-compression algorithm also uses temporal and spatial data locality and delta encoding for better performance. Instead of using Huffman style coding which requires extra bits for each delta values, we use Z-order encoding to compress the delta values of all attributes of the input data into a binary stream. When decoding we use the predefined decoding rules to decode the Z-values and extract all the values of attributes.

We conducted extensive experiments using skewed and unskewed real datasets. We found that Z-order encoding based compression performs better than Huffman tree based source coding approaches. We further optimized the original Z-order encoding, where, for skewed datasets, we proposed the initial code library to improve the compression performance further. Our experiments show that it has much better compression ratios for the multi-dimensional datasets than the previous Huffman coding based compression approaches like LEC [29], TinyPack [38], Adaptive-LEC [40] and FELACS [17]. The packet compression evaluation is done in a wireless sensor network using TelosB motes, which use the IEEE 802.15.4 radio and 8 MHz TI MSP430 microcontroller with 10KB RAM. The energy consumption rate evaluation is done in TOSSIM [22] using powerTossim-Z [33].

In this paper, we compare the performance of our work with LEC, Adaptive-LEC, FELACS and TinyPack (since they outperform other algorithms like ASTC [1], S-LZW [35] and GAMPS [11]). [31] proposed their coding dictionary based on a very specific dataset, thus, not considered in our experiments.

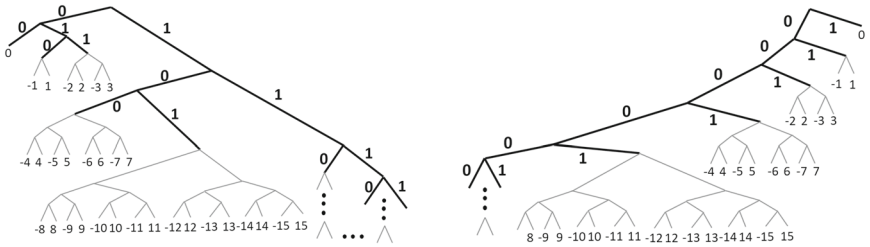
In summary, this paper makes the following contributions:

- Design of an efficient multi-modal sensor data Z-compression scheme that combines the Z-order encoding with delta compression. To our knowledge, this is the first attempt to propose a multi-modal lossless data compression algorithm for WSNs.
- Create a probability model of the sensing data which has the temporal and spatial locality such as the environmental data, location data and motion data.
- Design a data concatenation scheme which can concatenate leaf nodes data efficiently for compression. Also, improve Z-order encoding by integrating a small dictionary and an odd bit Z-order encoding for further performance improvement in WSNs.
- Integrate the proposed Z-compression scheme with low-power listening and high-streaming applications of WSNs and prove its effectiveness using the simulation experiments.
- Perform extensive simulations using TinyOS and TOSSIM and compare the performance with the recent and popular sensor data compression algorithms. The results show that our schemes outperform these other schemes in terms of compression ratio, handling high-streaming data and energy usage as the metrics.

2 Background and related work

S-LZW [35] an extension of LZW [45] compresses data by encoding and representing a common sub-string with fewer bits used for the encoded value. The encoded common sub-strings are stored in the dictionary which usually is too large to store at sensor motes. LZW and S-LZW usually need more data to create new words and update their library before compressing, which causes longer delays. Additionally, the average waiting time increases so LZW and S-LZW cannot be used for real-time wireless sensor applications as shown in our earlier work also [38].

LEC [29] and TinyPack [38] are both Huffman coding based delta compression algorithm. The Huffman coding would represent higher frequency symbols with less number of bits. However, in sensor networks, delta values can range from 0 to more than 2^{16}. Thus, the memory limitation of nodes makes creating such large Huffman coding dictionary impossible. In their improved approaches to save memory and to make them computationally efficient, instead of adapting Huffman coding to each symbol, LEC and TinyPack create the Huffman coding for the length of the delta value in a fixed pattern that matches the frequency of the data length group which they assume is decreasing when the length of delta value increases. Thus, in both LEC and TinyPack, the length of Huffman coded prefix increases with the increasing delta values length. The Huffman tree of LEC is shown in Fig. 1a and the TinyPack code is shown in Fig. 1b. These two coding focused on the different distribution of the delta values of the dataset. For example, if a dataset contains many data of only one-bit length, according to Fig. 2, TinyPack will have better compression ratio than LEC in terms of normalized compression overhead (ratio of the extra bits required to code the delta value to the data bits) whereas,



(a) Huffman tree of LEC code **(b)** Huffman tree of TinyPack's code

Fig. 1 Huffman tree of the compression code of LEC and TinyPack

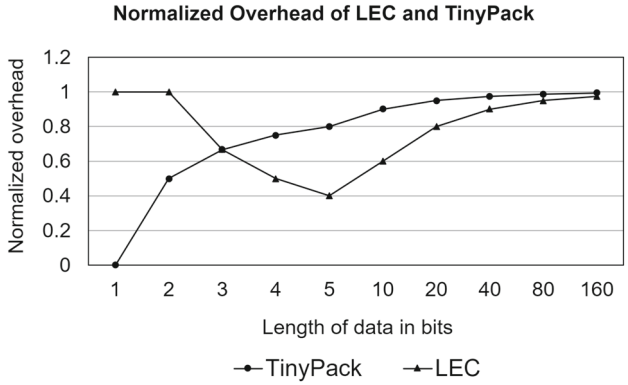


Fig. 2 Normalized compression overhead of LEC and TinyPack

for other delta values larger than 3, LEC will have better performance than TinyPack.

However, as we have discussed before, the distribution of the delta values can be varied in different applications. Even in the same application at different time periods or at different sensor nodes, the distribution may vary. Thus, in these situations, LEC and TinyPack do not show good performances as they have fixed code. To solve this problem, Adaptive-LEC [40] and TinyPack-DP(TP-DP) [38] can adapt the prefix code when the length of the distribution of the frequency of the delta value changes. Adaptive-LEC will adapt its prefix code for every new data while the TP-DP only changes its prefix code at the beginning of each new frame. They define the length of delta value with the most frequency as the frequency center of data and define the length of delta value with the minimum prefix code as the code center. When the current frequency center of data is drifted from the previous code center, the prefix code will shift to meet the current frequency center of the data. For some datasets with the two frequency centers, they proposed another initial code with the two code centers and two adaptive segments where the prefix codes will be adapted independently based on the segments. However, the drift correction, always behind the data drift, happens with a very high delay. Thus, it reduces the performance of the compression algorithm. Also, different sensor nodes may have different code centers which requires much overhead to handle the asynchronous WSNs.

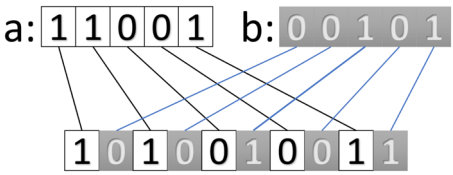


Fig. 3 Procedure of Z order encoding

Szalapski and Madria [39] algorithm takes advantage of spatial locality between the two or more nodes, and performs collaborative data compression. However, in WSNs, sensors may be sensing multidimensional data independently and waiting for other sensors to dispatch their different attribute values will increase latency in processing and delays in transmission in an intermittent environment. Thus, compressing the multi-modal sensing data locally at each node becomes essential for real-time data transmission. Z-compression does not need to correlate data with other nodes. Therefore, there is no need to wait for more data to acquire before compressing.

FELACS [17] gives every compressing data a fixed b bits. Every value smaller than 2^b will be filled with '0's at the front to reach b bits, added '1' bit at the front then directly appended to the output stream. Every value larger than or equal to 2^b will be cut into two sections. The higher bits section is encoded using unary coding and is appended to the output stream. The lower bits section has the length b directly appended to the output stream. The fixed bits b is generated by calculating the average number of the input data.

FELACS adapts data packets by packets rather than sample by sample. It works for a single sensing attribute only. Also, it needs to wait for more data to achieve better compression performance as they have fixed indicator bits.

3 Proposed Z-compression approach

In this section, we describe the efficient Z-order encoding based multi-modal data compression scheme, Z-Compression. It is a lossless compression algorithm that exploits temporal and spatial locality. The wireless sensor nodes use the delta value of each attribute as the input of the compression algorithm. The delta value is calculated using Eq. 1 where V_c is the current delta value, V_p is the previous delta value and P is the resolution of the sensors.

$$d_\Delta = \frac{V_c - V_p}{P} \tag{1}$$

3.1 Naive multi-dimensional Z-compression for sensor values

Naive Z-compression uses Z-order encode [32] and all-is-well bit like in [38]. As shown in Fig. 3, the Z-order encode interleaves input data bit by bit and output a new binary number with a length that double of the largest input.

$$V = \begin{cases} 2 \times V_{signed}, & \text{if } V_{signed} > 0 \\ 1, & \text{if } V_{signed} = 0 \\ 1 - 2 \times V_{signed}, & \text{if } V_{signed} < 0 \end{cases} \tag{2}$$

We choose the byte array as the data structure to store the encoded data. Also, we use unsigned integer V to represent both positive, negative and zero values V_{signed} by using the Eq. 2. Next, we add '1' at the front of the output to protect the possible '0'. For example, consider three dimensional array, '10', '110' and '1'. The largest attribute is '110' which has 3 bits. So first, we need to add zeros to the other two attributes. Then applying Z-encoding on '010', '110', and '001' to get the Z-value, '010110001'. Then, we add '1' which gives the final result of '1010110001'.

In wireless sensors, as the computational and memory resources are limited, it is better to use bitwise operator rather than converting the encoding data into string format when doing Z-order encoding. We use bitwise 'shift', 'or'(|), 'and'(&) for interleaving a certain bit from the input data and append to the output. Note that the notation '<<' and '>>' means shift left and shift right for certain bits. We first need to find out the length of the largest input data as B_l. Then the length of the output array, L_Z, can be calculated using the Eq. 3. In the same way, the output length of TinyPack and LEC on compressing the two sensing attributes can be calculated using Eqs. 4 and 5 where B_i represent the number of bits of the ith attribute which needs to be compressed.

$$L_Z = N \times B_l + 1 \tag{3}$$

$$L_{TP} = \sum_{i=1}^{N} (2B_i - 1) + 1 \tag{4}$$

$$L_{LEC} = \sum_{i=1}^{N} (2B_i - T) + 1, T = \begin{cases} 0, & \text{if } 0 < B_i < 3 \\ 1, & \text{if } B_i = 3 \\ 2, & \text{if } B_i = 4 \\ 3, & \text{if } B_i = 5 \\ 4 & \text{if } B_i > 5 \end{cases} \tag{5}$$

As stated, we also integrate an all-is-well bit [38] with Z-order encoding. It sets the compressed data to be zero if all the delta values of the input attributes are zero. In Z-compression, we set the result to be an all-is-well bit when nothing changes. It is done by directly checking all the input values. If all the inputs equal to '1' then output '1' where '1' equals to zero according to the Eq. 2.

Different from the Huffman-tree based compression algorithms that need to know the probability distribution of the input data to achieve the best performance, Z-compression only exploits the relationship between the length of delta values of the attributes. We then compare the normalized overhead (in terms of extra bits over total bits) of Naive Z-compression, LEC, and TinyPack on compressing the uniformly distributed multi-modal random data by using the Eqs. 3, 4 and 5. The result is shown in Fig. 4.

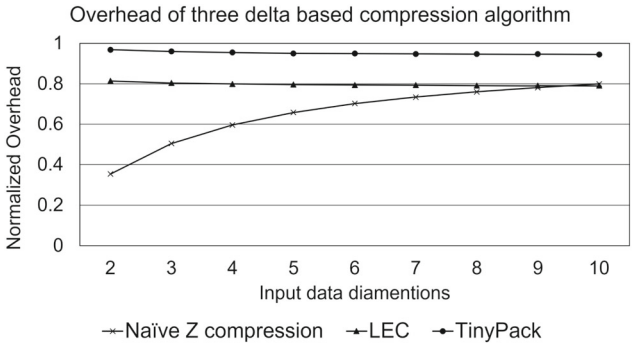


Fig. 4 Normalized compression overhead compressing multidimensional data

The result shows that Naive Z-compression performs well when the number of attributes is small. However, with the increasing number of attributes, the overhead of Z-encode increases. When the number of attributes is larger than eight, the overhead of Z-encode is greater than that of LEC. To address this drawback, we next propose an Optimized two-dimensional Z-compression scheme which guarantees to have a better compression ratio than TinyPack and LEC when compressing two-dimensional data. Thus, for multi-dimensional data, we can split it into several groups with two or more attributes to minimize the overall overhead due to extra bits.

3.2 Optimized two-dimensional Z-compression algorithm

Before improving Naive Z-compression, we need to study two important properties of two-dimensional Z-order encoding. Here, we do not consider the extra '1' bit added before the compression result.

- The number of bits in the output of Two-dimensional Z-order encoding is always even.
- There must be at least one of 1 in the first two bits of the output of the two-dimensional Z-order encoding in binary format.

The first property indicates that We can use the value with odd length to represent the 'skewed' data. Here, we define a two-dimensional dataset as skewed when the number of bits of the larger delta value is more than two times the number of bits of the smaller as in Eq. 7. When the data is skewed, in order to make the length of Z-value odd, we divide the larger value V_L with length B_l into two values V_{L1} and V_{L2} of length B_{l1} and B_{l2} using the Eq. 6a–c.

$$Mask_{l2} = (1 << B_{l2}) - 1, \tag{6a}$$

$$V_{L1} = V_L >> B_{l2}, \tag{6b}$$

$$V_{L2} = V_L \& Mask_{l2}; \tag{6c}$$

We can apply Z-order encoding on V_{L1} and V_S to get the Z-value Z_{half}. Then by appending V_{L2} on Z_{half}, shown in Eq. 8, we get the Z-value for the skewed data.

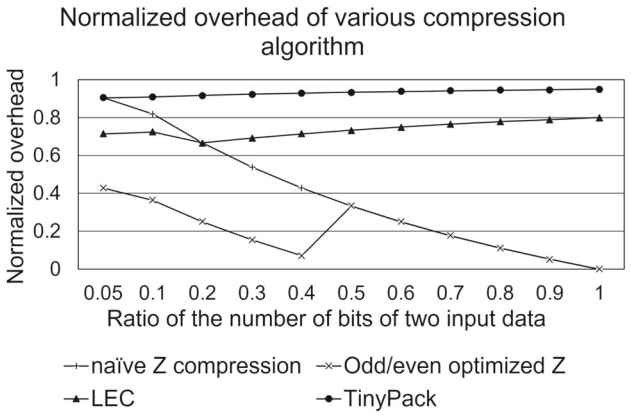


Fig. 5 Normalized overhead in compressing two attributes where the largest attribute's data is 20 bits long

The notation $\otimes$ means applying Z-order encoding and $\oplus$ means concatenating the two binary strings together. In the above procedure, we have to ensure that the Z-value has odd length L_{odd} and each different two-dimensional data maps to a unique Z-value.

Algorithm 1 shows the procedure of the Optimized two-dimensional Z-compression. It reduces the extra bits adding to the smaller delta value when the delta value pair is skewed. However, when the delta value pair is not skewed, Naive Z-encode is applied. For example, when encoding two attributes '101' and '11110000', we add two '0' bits to the first attribute '101' which makes it '00101' and set the end pointer of the second attribute to be 3 which divides it into two values, '11110' and '000'. Then we apply Naive Z-compression on '00101' and '11110' which produces '0101110110'. At the end, we append the remaining '000' to it and add '1' as the leftmost bit which gives us the encoded Z-value of '10101110110000' which is 14 bits long.

$$B_s < floor(B_l/2) \tag{7}$$

$$Z = Z_{half} \oplus V_{L2} = (V_{L1} \otimes V_S) \oplus V_{L2} \tag{8}$$

When decoding, we use the second property of two-dimensional Z-order encoding to help us figure out which value within the two decoded data is larger. The value with less '0's at the front is larger. Then we append the remaining data to it to get the final result. For example, when decoding '10101110110000', we first count the length of the Z-value after the first '1' which give us 13. This odd value 13 indicates that the delta value pair is skewed. Then using $13\%6 = 1$, we know that it belongs to the first case in Algorithm 1 that $B_L\%4 == 0$. And, using $n = 13/2 = 2$, we can divide the Z-value into two parts at $2 \times (2n + 1) = 10$ bits from left. By interleaving the bits from the first part, we get '00101' and '11110'. At the end, we append the remaining '000' to the second value as it has less '0' bits at the front than the first one so the decoder will output two binary values '00101' and '11110000'.

In Fig. 5, we compare the normalized overhead of extra bits of four different compression algorithms LEC, TinyPack, Naive Z-compression and Optimized Z-compression on compressing two-attributes data where the larger attribute is 20-bits

long. The X-axis is the ratio of the number of bits of the input delta value pairs. The Y-axis is the normalized overhead which is the extra bits over the total bits that the input delta value pair has. We can see that Optimized Z-compression dwindle the extra bits significantly. The maximum normalized overhead is 0.43 which is half of the normalized overhead of TinyPack. In Optimized Z-compression using odd/even bit optimization, we use two as the critical ratio of two attributes' length to trigger the Optimized Z-compression as it is easier to implement.

Algorithm 1: Optimized Two-dimensional Z-compression algorithm using odd/even bits optimization

input : Delta value V_L and V_S where $V_L > V_S$
output: Z value

initialization $B_L \leftarrow Length(V_L)$, $B_S \leftarrow Length(V_S)$
if $B_S \geq floor(B_L/2)$ **then**
  $Z = V_L \otimes V_S$;
else
  if $B_L\%4 == 0$ **then**
    $n \leftarrow B_L/4$;
    Divide V_L into V_{L1} and V_{L2} where $V_L = V_{L1} \oplus V_{L2}$;
    $Length(V_{L1}) = 2n + 1$ and $Length(V_{L2}) = 2n - 1$;
    $Z = (V_{L1} \otimes V_S) \oplus V_{L2}$;
  else if $B_L\%4 == 2$ **then**
    $n \leftarrow B_L/4$;
    Divide V_L into V_{L1} and V_{L2} where $V_L = V_{L1} \oplus V_{L2}$;
    $Length(V_{L1}) = 2n + 1$ and $Length(V_{L2}) = 2n + 1$;
    $Z = (V_{L1} \otimes V_S) \oplus V_{L2}$;
  else if $B_L\%4 == 3$ **then**
    $n \leftarrow B_L/4$;
    Divide V_L into V_{L1} and V_{L2} where $V_L = V_{L1} \oplus V_{L2}$;
    $Length(V_{L1}) = 2n + 2$ and $Length(V_{L2}) = 2n + 1$;
    $Z = (V_{L1} \otimes V_S) \oplus V_{L2}$;
  else
    $n \leftarrow B_L/4$;
    $V_L =' 0' \oplus V_L$;
    Divide V_L into V_{L1} and V_{L2} where $Length(V_{L1}) = 2n + 1$ and $Length(V_{L2}) = 2n + 1$;
    $Z = (V_{L1} \otimes V_S) \oplus V_{L2}$;
return $Z =' 1' \oplus Z$;

We then compare another optimization scheme that can apply Optimized Z-compression on two input attributes with any ratio of lengths. We use a fixed small bit as a ratio indicator to indicate the actual ratio of two attributes' lengths. However, extra control bits can cause extra overhead. For example, when using two bits indicator, four different ratio which are $\frac{3}{2}$, $\frac{3}{1}$,$\frac{2}{3}$, and $\frac{1}{3}$, can be indicated using '00', '01', '10', '11'. We can then interleave binary values in a fixed ratio. For example, when the ratio is set to be $\frac{3}{2}$, we should interleave 3 bits at a time for the first delta value and 2 bits at a time for the second delta value.

Algorithm 2: Optimized Z-compression algorithm with input length ratio indicator

input : Delta value V_L and V_S where $V_L > V_S$
output: Z value
initialization $B_L \leftarrow Length(V_L)$, $B_S \leftarrow Length(V_S)$
if $CalculateOverhead(B_L, B_S)$ *prefer Naive* **then**
  $Z =' 1' \oplus V_L \otimes V_S$;
else
  $Z =' 0'$;
  $Z = Z \oplus FindBestRatio(B_L, B_S)$; $Z = Z \oplus FindZValue(V_L, V_S, ratio)$;
return $Z =' 1' \oplus Z$;

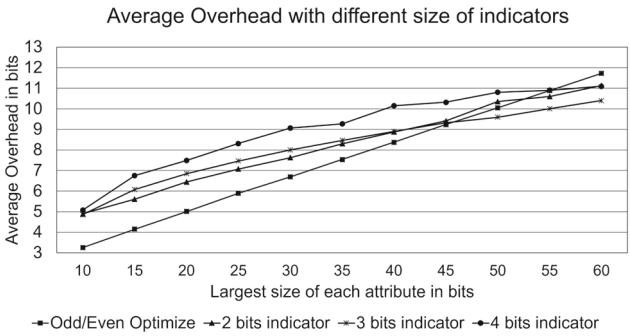


Fig. 6 Average overhead in compressing data with two attributes

We changed the ratio indicator bits from 2 to 4 and tested it on a uniformly distributed random data set. Figure 6 shows that with the increasing length of input data, the average overhead of different optimized Z-compression also increases. However, the odd/even checking based optimization performs better than others when the input's maximum length is fewer than 45 bits which is larger than the size of most wireless sensor's data sensing output. Thus, when compressing real-time sensing data packets, we suggest only to use Algorithm 1.

3.3 Small code library add-on

We can further improve Optimized two-dimensional Z-compression scheme by integrating a small code library. This library will enhance the compression performance without affecting the correctness. The code library is shown in Table 1. For entries in the small library, one of the input delta value pairs should have '0' or '1'. We assign the output a smaller value which never appears in the result of Naive or Optimized Z-compression. For example, when we compress two delta values where both the values are 0, the output of encoding value will be 1-bit '1' instead of 2-bits '11' giving 50% improvement. Also, this small code library will not affect the correctness of the encoding and decoding procedures given in Algorithm 1 because it will not generate Z-values that the small library has. When decoding, we can check the small library before checking the length of Z-value to get the correct decoding result.

Table 1 Initial small code library

Value 1	Value 2	Z value
0	0	1
0	1	11
1	0	10
0	− 1	111
− 1	0	110
0	$V_2 > 31 \,\|\|\, V_2 < -31$	10,000 ⊕ V
$V_1 > 31 \,\|\|\, V_1 < -31$	0	100,000 ⊕ V

3.4 Optimized N-dimensional Z-compression algorithm

To improve Naive Z-compression for more than two-dimensional data, we proposed the optimized N-dimensional Z-compression that exploit the data correlation between close attributes. As the number of the sensing attributes are fixed based on the sensing hardware, we suppose both the encoder and decoder know how many attributes they are going to compress and decode.

The optimized N-dimensional Z-compression algorithm combines the procedures discussed in 3.A, 3.B, and 3.C using a predefined rule which is generated in Algorithm 3. When compressing multidimensional sensing data, we can either use Naive Z-encoding based compression on all the attributes or use Optimized Z-compression on the pairs of attributes and then merge the result. Testing all the combinations of the input attributes with the above two encoding methods is an NP-complete problem. Thus, we propose an approximate algorithm using two pointers and a local greedy approach to find the approximate combination result given in Algorithm 3. We use a two-dimensional array as **GroupMember** to represent the attribute IDs and their length. We use another array as **Group** to store the list of **GroupMember**s. The input of the algorithm is a list of **GroupMember**s which represent all the attributes. The output of the algorithm is a list of **Group**s which instructs the encoding and decoding procedure.

The encoder will use the **Group** information to help them encoding. If a **Group** contains more than three entries, the encoder will use the Z-order encoding to compress the attributes represented by the group. If a **Group** contains only two attributes, the encoder will use Algorithm 1 and Table 1 to compress them. At last, the sensor will further encode all the encoded values and output the result. For example, if there is a **Group** [{1,7},{2,5},{4,9}], the wireless sensor node will do a Z-order encoding on the attributes with field IDs 1, 2 and 4. If there is another **Group** [{3,3},{5,10}], the sensor will do the optimized two-dimensional Z-compression on the attributes with field ID 3 and 5. Then the node will compress the two encoded values using optimized Z-compression as there are only two groups. After transmitting the compressed value to the decoder, the decoder will decode the Z-value reversely based on the number of groups and the field IDs in each group.

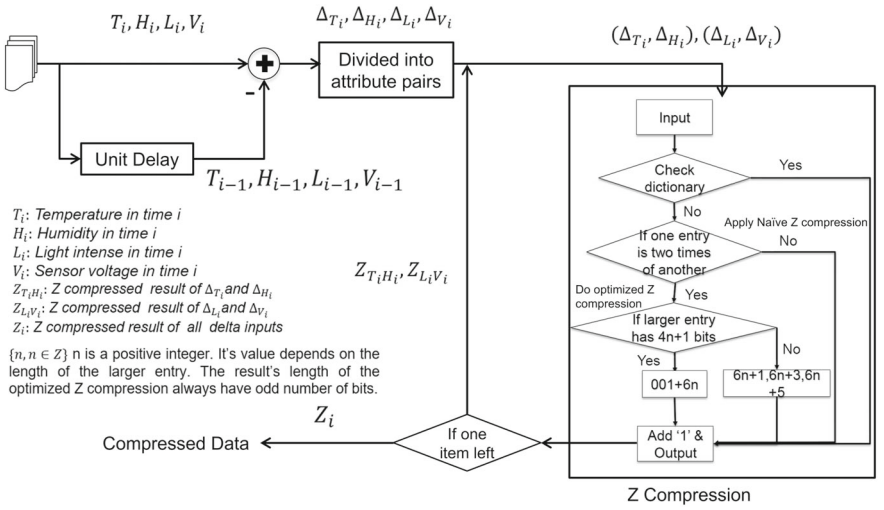


Fig. 7 System model of optimized Z compression

Algorithm 3: Rule Generation Algorithm

input : List of **GroupMember**s: GML
output: List of **Group**s: GL

```
Sort(GML); //sorting based on the length of attributes
LP← 0; //left pointer starting from the left end
RP ← GML.getSize(); //right pointer initialization
while (length(GML[LP])<length(GML[RP-1])/2&&LP<RP) do
    LP++,RP- -; //find pairs meeting equation 2
//add groups of the pairs to the output list
while RP<GML.getSize() do
    GL.add(new Group{GML[RP++],GML[GML.getSize()-RP]}
bufGroup=new  Group{}//initial a new group
//add rest GroupMembers to the Group
for (i=LP;i<GML.getSize()-LP;i++) do
    bufGroup.add(GML[i]);
GL.add(bufGroup);
return GL;
```

For example, for compressing the Intel Berkeley Lab data, which includes the following four attributes: temperature, humidity, Light tense, and voltage of sensors, the system model of optimized Z-compression is shown in Fig. 7.

First, calculate the delta values of these four attributes using Eq. 1. Second, we do learning based on the first 100 packets. Find the compression rules using Algorithm 3. Third, group the attributes based on the compression rules. In this example, the best compression strategy is to group the temperature data with humidity data and group the light intensity data with the sensor voltage data, then group the compressed data of these two groups. Next, compress data in each group using the optimized Z-compression

Algorithm 4: Lossless data concatenating algorithm

```
input : Q, N, b_max, b_min, sum, L
output: payload: out[]
Index=new byte[N], ind=1, prev=L[0][1];
Sort(L); //sorting is based on L[][0]
for (i = 0;i<N;i++) do
    Index[L[i][1]]=ind;
    ind+=prev;
    prev=L[i][1];
prev=L[0][1];
for (i=0;i<N;i++) do
    buf=Q.poll();
    for (j=prev-L[i][1];j<L[i][1];j++) do
        out[Index[i] + j]=buf[j]
    prev=L[i][1];
out[0]=N;
return out;
```

Table 2 Fields of experimental dataset

Data set name	Number of fields	Fields label
Intel Berkeley Lab data	6	epoch, node ID, Temperature, Humidity, Light, Voltage
Accelerator in moving car	5	epoch, node ID, X, Y, Z
ZebraNet data	5	epoch, node ID, Longitude, Latitude, Voltage
Vehicle trace data (V to V)	10	epoch, node ID, (Longitude, Latitude, Altitude, speed)× 2

Algorithm 1 until there are no more groups and then output the compressed data (Table 2).

3.5 Lossless data concatenating algorithm

The main reason to concatenate the locally compressed packets is to save the energy and bandwidth usage further. In our experimental results, also shown in experimental analysis of radio performance in [18,26], we found that reducing the payload size of the packets at the leaf node will not give us much energy saving. Also, a leaf node is not the bottleneck in WSNs as there is not much traffic via them. However, we found by the experiment that the intermediate nodes are the bottleneck as they not only need to sense data but also need to route packets from the lower level nodes to the higher level nodes in the tree. The energy consumption rate and bandwidth occupation by the intermediate nodes are much more critical than the leaf nodes. Thus, to save energy and prolong the lifetime of WSNs, we only need to consider the energy consumption

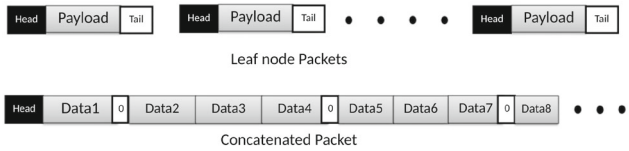


Fig. 8 Concatenate packets based on packets' size

of the node with the most radio connections. To do that, we need to concatenate the upstream data to decrease the number of packets each intermediate node will transmit.

Next, we discuss how to concatenate the locally compressed data packets. The compressed data are in the byte arrays of variable length. It has the node ID and timestamp as the primary key. It is not practical to decode and re-encode all the data at the intermediate nodes as it will increase the RAM and CPU load and thus, will cause delays. Here, we propose a data concatenating algorithm which will concatenate byte array input data efficiently. The idea is that in a packet transmitted by the intermediate node, the compressed data with a larger number of bytes is always in front of the compressed data with a smaller number of bytes. However, the compressed data with a smaller number of bytes will be filled with zero bytes to make them have the same number of bytes as their neighbor in front of them. At the end, we set the first byte of the output packet as the number of bytes of the largest data. When decoding, we first read the length of the largest packet b_{max} at the first byte. Then we read the following $2b_{max}$ bytes to extract the first and the second data. If there are empty bytes in front of the second data, we need to update the b_{max} by subtracting the number of empty byte from b_{max}. Next, we use the updated b_{max} to extract the third data and update the b_{max} again. Repeating this process until the last byte of the packet, we can extract all the data samples in the packet. Figure 8 shows how we concatenate small size packets into large size packets by sorting the size of the packets in the descending order.

$$PacketSizeLimit < (sum + b_{max} - b_{min}) \tag{9}$$

Note that we use a queue Q to store the upstream packets' payload, a two-dimensional array L to store the size of each payload and their index in Q, an index array $Index$ to help append the data in the queue to the output byte array. We also need to track the largest packet length b_{max}, the smallest packet length b_{min} and the total length of all the packets $sum = \sum_{i=1}^{n} b_i$ in the queue. Once the criteria in Eq. 9 is satisfied, we use Algorithm 4 to concatenate the elements in the payload queue and create a larger size packet. The total number of data needs to concatenate is 'N' which exclude the last item in the queue. Then the intermediate node will transmit the large size packet to its downstream node.

4 Entropy and data distribution model

Based on Shanon's entropy theory, the minimal number of bits needed to represent each value of the source data can be calculated using Eq. 10 where p_i is the probability of the appearance of each symbol.

$$H(X) = -\sum_{i=0}^{N-1} p_i log_2 p_i \tag{10}$$

The entropy indicates how much we can compress the input sensing data. The Eq. 10 shows Shannon entropy determined by the probability of the data elements in the set. When we compress multi-dimensional data, the sum of the entropy of all sensing attributes cannot be used as the entropy of the whole data due to different temporal and spatial correlations. Since the distribution of the sensing data is not known before sensing and therefore, to evaluate the compression algorithms' performance, we propose a temporal and spatial data model that could simulate the delta (Δ) values of the real-world sensing data.

4.1 Temporal and spatial data approximation and regression

The temporal property means, in a short sensing period, two consequent sensor data have similar values. The spatial property means the nearby sensor or the same sensor moving nearby, the sensor data will have the similar values. Below, we discuss the data approximation using regression, the probability mass function of the bits in the compressed sensor data and the possible compression strategies.

For data with temporal locality, we find that the sensing period can directly affect the statistical distribution of the delta values. Intuitively, if the sensing period is small, the delta values of the sensing data poses to be small because the data do not have much time to change. We describe the distribution of the delta values of the sensing data with the temporal locality mathematically in the following sections.

Assumptions and preconditions: When the sensor data have the temporal locality property periodically, we only consider data from one period only. In that period, the data have three stages; the ascending stage, the stable stage and the descending stage. We perform linear regression at each stage and use a straight line with a fixed slope β_1 as the linear model. Note that in the following equations, x_i refers to the time i and y_i refers to the observed sensing value at time i .

$$y_i = \beta_0^i + \beta_1 \times x_i$$

We can find the β_0 and β_1 by minimizing the Pearson's correlation coefficient at each stage.

Regression and approximation: In each period, we approximate the data using the following three linear regression models; the ascending linear line model (11), the descending linear line model (12) and the static linear line model (13).

$$y_i = \beta_0^{asc} + \beta_1^{asc} \times x_i \text{ where } \beta_{asc} > 0; \tag{11}$$

$$y_i = \beta_0^{des} + \beta_1^{des} \times x_i \text{ where } \beta_{des} < 0; \tag{12}$$

$$y_i = \beta_0^{sta} \tag{13}$$

After proper slicing of the dataset and regression, the average residual of the linear regression should fall in the 10% criteria which indicates a very small correlation. We assume that the residual is white noise with the mean value zero.

4.2 Probability distribution of Δ of the sensor data

The Δ is the difference of two sensor data in the adjacent time period, $\Delta = Value_i - Value_{i-1}$ where $Value_i$ is the sensing value at time i. In sensing period, say T_s, the expected value changes can be calculated with Eqs. 11, 12 and 13. We calculate the average value changes E_Δ in the sensing period using the Eq. 14. As each sensing is independent of others, the distribution of each sensing period follows the Poisson distribution shown in Eq. 15 where k indicates the steps of the data change in each sensing period. For example, consider a temperature sensor in the sensing period T_s where the temperature of this sensor is 0.1 degrees higher than the sensing data in the last period. Suppose the sensor has the sensing ranges from -40 degree to 123.8 degrees and the sensing result is in 14 bits binary format. In that sensing period, the total steps of the temperature changes can be given as $\frac{2^{13}\times 0.1}{123.8-(40)} \approx 10$ which results in a 5 bits delta value of the temperature. Note here that we need to consider the positive or negative delta values. For example, the positive 10 steps can be represented as 10100 which ends with 0 while the negative 10 steps can be represented as 10101 which ends at 1.

$$E_\Delta = \begin{cases} \beta_{asc} \times T_s & \text{When in ascending stage} \\ \beta_{des} \times T_s & \text{When in descending stage} \\ 0 & \text{When in static stage} \end{cases} \tag{14}$$

$$P_\Delta(k_\Delta) = \begin{cases} \frac{\beta_{asc}^{k_\Delta} e^{-\beta_{asc}}}{k_\Delta!} & \text{When in ascending stage} \\ \frac{|\beta_{des}|^{k_\Delta} e^{-|\beta_{des}|}}{k_\Delta!} & \text{When in descending stage} \\ 0^{k_\Delta} & \text{When in static stage} \end{cases} \tag{15}$$

Considering the accuracy range of the sensors, based on the $3-\sigma$ rule, 99.7% of the sensing data should within the accuracy's upper-bound $3\sigma'$ and the accuracy's lower-bound $-3\sigma'$. The PMF distribution should follow the discrete approximation of the normal distribution with $\mu = 0, \sigma = \sigma'$ which is the binomial distribution Eq. 16 with $n = 4 \times \sigma^2$, $p = 1/2$:

$$P_{bino}(k) = \binom{n}{k + n/2} p^{k+n/2} \times (1-p)^{n/2-k} \tag{16}$$

The noise of the sensor is independent from the sensing attributes so we can join these two PMFs Eqs. 15 and 16 using the following equation:

$$P_{XY}(x, y) = P(X = x, Y = y) = P_X(x) \times P_Y(y) \tag{17}$$

Thus, we can get :

$$P_{k=k_\Delta + k_{bino}}(k_\Delta, k_{bino}) = P_\Delta(k_\Delta) \times P_{bino}(k_{bino}) \tag{18}$$

Next we want to find the PMF in terms of $k = k_\Delta + k_{bino}$ at each stage. For the convenience of the calculation, we assume the noise, within the sensor accuracy, is in the range of $[-3\sigma, 3\sigma]$. For stage 1, $k \in [-3\sigma, n+3\sigma]$ where n is the positive sensing limitation. We get:

$$P_k(k) = \sum_{k_{bino}=-3\sigma}^{3\sigma} P_{bino}(k_bino) * P_\Delta(k - k_{bino})$$

For stage 2, $k \in [-n-3\sigma, 3\sigma]$ where -n is the negative sensing limitation. We get:

$$P_k(k) = \sum_{k_{bino}=-3\sigma}^{3\sigma} P_{bino}(k_{bino}) * P_\Delta(k - k_{bino})$$

For stage 3, k is ranging from $[-3\sigma, 3\sigma]$. We get:

$$P_k = \binom{n}{k+3\sigma+n/2} p^{k+3\sigma+n/2} \times (1-p)^{n/2-(k+3\sigma)}$$

As the number of bits of the compressed data is only determined by the number of bits in the delta value $K = log_2(k)$, we only need to consider the PMF of K. Thus, we can represent each stage's PMF function P_K using each model's P_k as follows.

$$P_K(K) = \sum_{k=2^{K-2}}^{2^{K-1}-1} P_k(k) \tag{19}$$

Let's assume that the number of ascending sensing periods in stage 1 be np_1, the number of descending sensing periods in stage 2 be np_2, and the number of sensing periods in stage 3 be np_3. As discussed above, we assume that the expected delta value is zero over the long period. Thus, we have the following derivation:

$$np_2 = \frac{np_1 * \beta_{as}}{\beta_{de}} \tag{20}$$

So the distribution of the number of bits in the delta value becomes:

$$\begin{aligned} PMF(K) = &\frac{np_1 * P_K(K)_{stage1}}{np_1 + np_2 + np_3} + \frac{np_2 * P_K(K)_{stage2}}{np_1 + np_2 + np_3} \\ &+ \frac{np_3 * P_K(K)_{stage3}}{np_1 + np_2 + np_3} \end{aligned} \tag{21}$$

Consider Intel-Lab environmental data [27] which records the temperature, humidity, and light intensity; here we only consider the temperature and humidity as both of which have the temporal and spatial locality. From [20], in a single day, the indoor

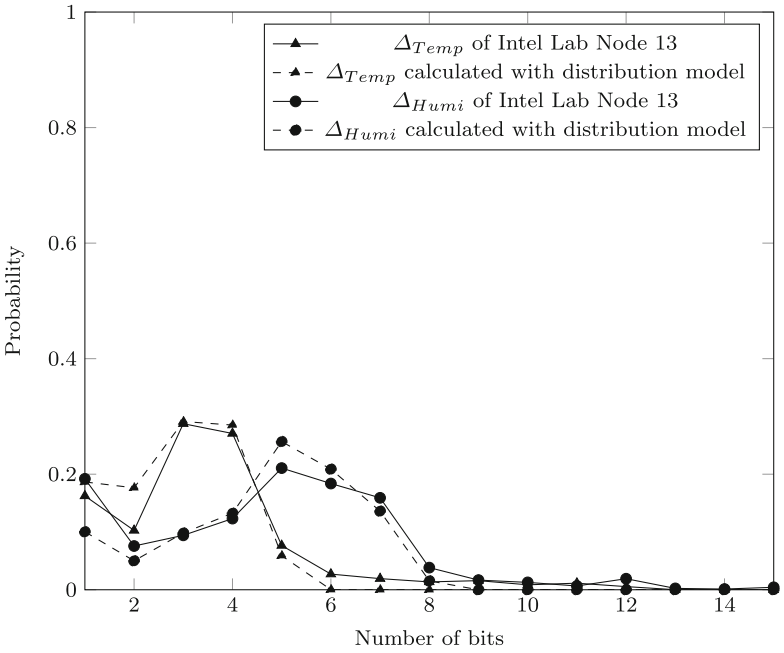
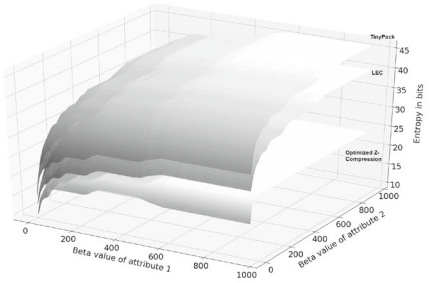


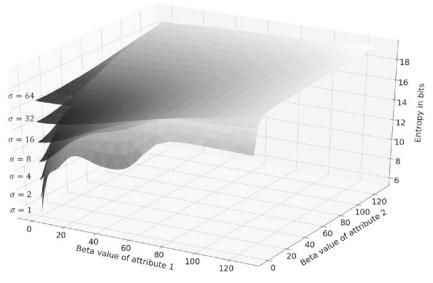
Fig. 9 Calculated PMF of the number of bits in the Intel lab data

temperature rising period is about 6 h when it increases by 1.6°, descending temperature period is about 2 h when it decreases by 1.6° and the constant temperature period is about 16 h. The ratio of these three time periods is the 3:1:8. The three periods of the humidity has the value change of + 36%, − 36% and 0% whose ratio is the same as the ratio of the temperature. Thus, for distribution (Eq. 15), stage 1 is for the rising period of the temperature and humidity where $\beta_{temp} = 1$ and $\beta_{humi} = 7$ based on the approximate sensing period of 2 min. Stage 2 is for the temperature and humidity descending period. Stage 3 is for the temperature and humidity holding period where $\beta_{temp} = 0$ and $\beta_{humi} = 0$. In Eq. 20, we set $np_1 = 3, np_2 = 1, np_3 = 8$, thus, we can get $\beta_{de} = \frac{np_1 * \beta_{as}}{np_2}$. After studying the sensor's datasheet [6], we find the accuracy for the temperature is 0.5 degree, and the accuracy of the humidity is 3.5%. Consider the sensor resolution of the temperature as 0.01 and the sensor resolution for the humidity as 0.03%. As the accuracy means the sensing value should have high probability to fall in the accuracy range, we set $6\sigma = \frac{Accuracy}{resolution}$ which gives us $\sigma_{Temperature} = 3.1$ and $\sigma_{Humidity} = 7.3$. Then we can calculate $n = 4 \times \sigma^2$ by the distribute (Eq. 16) postulate which is the discrete approximation of the normal distribution. Then after joining of Eqs. 15 and 16 using Eq. 21, we get the approximate distribution of the number of bits of the indoor temperature and humidity as shown in Fig. 9.

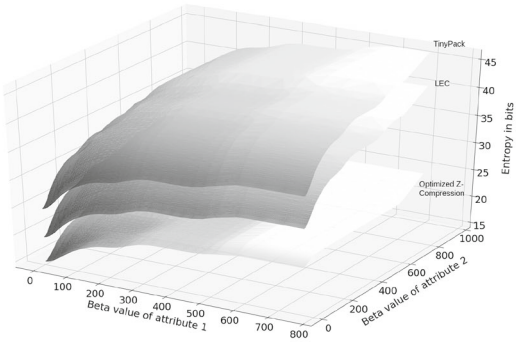
Our distribution model represents the number of the bits of the sensing data correctly with less than 10% distortion. Using this data model, we can simulate different temporal and spatial sensing data by modifying the five arguments defined in this data



(a) Average number of bits in the compressed data with different β values



(b) Average number of bits in the compressed data with different β and σ values



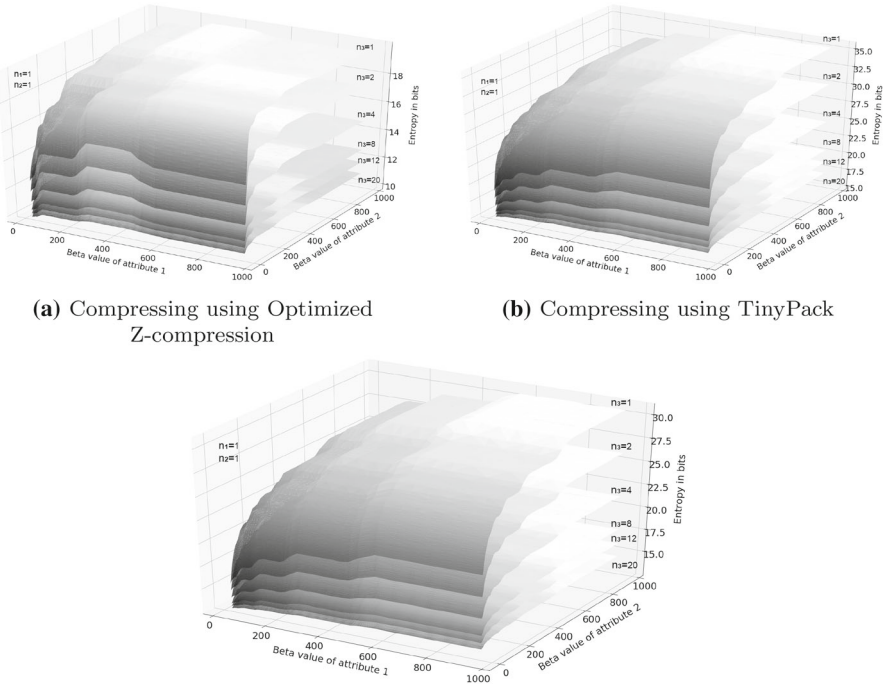
(c) Average number of bits in the compressed data with $\sigma = 64$

Fig. 10 Average number of bits in the compressed data with different β and σ values for optimal Z-Compression, LEC, and TinyPack

model. We then evaluate the performance of different sensor compression algorithms on compressing the data generated using this data model.

4.3 Performance evaluation using temporal and spatial data model

Recall the data model mentioned in Sect. 4.2 where five attributes which affect the distribution of the sensing data with the temporal and spatial property are given. The conjunction of the PMF of the sensing data model can be seen as the combination of three Poisson distribution with variance σ. The β values of stage 1 and stage 2 of the sensing period determine the center of the PMF of the data distribution. The stage period $n1, n2, n3$ determines the proportion of each Poisson stage in the whole sensing period. Suppose we have two independent randomly selected sensing attributes say attribute 1 and attribute 2 which need to be compressed with 300,000 samples. To study the effectiveness of optimized Z-Compression, TinyPack, and LEC algorithms on these two attributes' sensing data, we vary the β value, the σ value and the ratio of each stage period. We use the Eqs. 3, 4, and 5 given in Sect. 4.2 to estimate the average number of bits in the compressed data. The result is shown in Figs. 10a–c, and 11.



(a) Compressing using Optimized Z-compression

(b) Compressing using TinyPack

(c) Compressing using LEC

Fig. 11 Average number of bits of the compressed values against different ratio of stage 3 (n_3) in the whole sensing period. Stage 1 and stage 2 has ratio $n_1 : n_2 = 1 : 1$, $\sigma = 5$

In Fig. 10a, we can notice that when compressing two independent attributes with the distribution model (mentioned in Sect. 4.2) that has $\sigma = 5$ and $1 : 1 : 0$ stages ratio, the optimized Z-Compression could always beat the TinyPack and LEC on the compression ratio independent of the β values of two attributes. We can also see that closer the two β values are, the better the compression ratio is. It is because a fewer number of extra bits will be added if two attributes are similar in the number of bits when using the optimal Z-Compression.

In Fig. 10b, we can notice that when we increase the σ value of the sensing data distribution model, the average number of bits of the compressed data increases. The effect is more significant when the β value is small for the two compressing attributes. When the β values are more than twice of the σ value, the effect of the σ value on the number of bits of the compressed data is trivial.

Figure 10c shows although larger σ value will introduce extra bits for the optimal Z-Compression, it is still better than the TinyPack and LEC for sensing data with *beta* values ranging from 1 to 1000. The reason is that TinyPack and LEC create more overhead in coding larger values.

The stage ratio of the sensing data, mentioned in Sect. 4.2, will affect the average number of bits of the compressed value for Optimized Z-compression, TinyPack, and LEC. When the ratio of stage 3 in the sensing period is higher, suggesting the data

has a high probability to be unchanged in the sensing period, fewer average bits are needed for the compressed value. Figure 11 shows for each ratio of stage 3 in the sensing period, Optimized Z-compression performs better than LEC and TinyPack. Thus, the stages' ratios of the sensing period will not affect the rank of these three compression algorithms in the evaluation.

4.4 Observation

In this section, we proposed a data model with less than 10% data distortion to simulate the Δ values of the temporal and spatial sensing data. Next, we evaluated three different compression algorithms against this model which show that when the β value is small, meaning the data with a lower number of bits appear more frequently, Optimized Z-compression, LEC, and TinyPack work well. The reason is that LEC and TinyPack both use the codebook which assumes the higher occurring probability of lower bit symbols like 0,1, and -1, so when the number of bits of the sensing data decreases, their compression ratio is close to each other (including of Z-compression). That is, LEC and TinyPack are designed for the data distribution with decreasing probability of occurrence when the number of bits of the data increases. However, when the β value increases, meaning the number of bits of the sensing attributes increases, the performance of LEC and TinyPack decrease significantly. It is because When the β value is small, the code of LEC and TinyPack fits the distribution model better, so the compression overhead is smaller. However, when the β or σ value increases, the distribution of the data diverges from the distribution model of LEC and TinyPack, thus the performance degrades. Whereas Optimized Z-compression does not depend on the data distribution only. It exploits the known information on how many attributes appear in compressing the sensing data and their average number of bits (better than estimating the bits using Shannon's entropy model or Huffman coding), therefore achieving better compression performance than the LEC and TinyPack.

5 Z-compression in high stream rate WSNs

In a wireless sensor network, there can be hundreds of wireless sensors nodes. As the number of sinks is limited, each data link to the sink is limited by the bandwidth of that sink. The bottleneck of the bandwidth is at the last hop of the data link which is the hop after the sink.

Both Naive Z-compression and Optimized Z-compression can be easily applied to different applications of WSNs. The best strategy to improve the throughput in a WSN that has high data streaming rate is to compress the data locally and concatenate the compressed data at the intermediate node.

To show, we assume a wireless sensor network organized as a tree structure where data will be transmitted from lower level to higher level in the tree. We assume that the root of the wireless sensor network is the base station. In such systems, the intermediate nodes not only need to perform sensing but also need to do routing from the lower level to higher level nodes. As stated earlier, the energy consumption rate and bandwidth

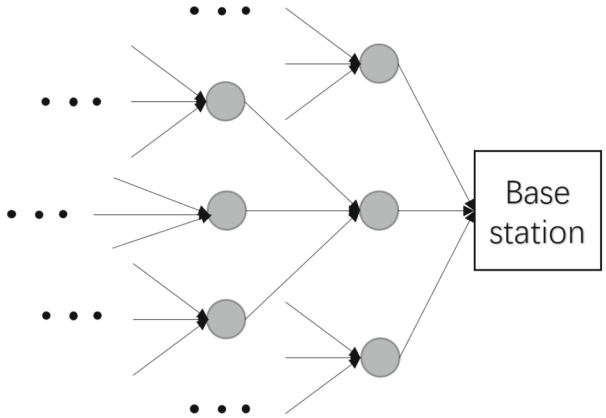


Fig. 12 Tree structure with R = 3

occupation by the intermediate nodes are much more critical than the leaf nodes. The lifetime of a wireless sensor network organized as a tree is the time elapsed until the first node in the network depletes its energy (like an intermediate node). In our experiments, we consider the intermediate node as the bottleneck for both energy and bandwidth consumption. Figure 12 shows a WSN organized as a tree structure with $R = 3$ where R stands for the number of children of each intermediate node. Every node will sense, compress and transmit local data at the same rate. The intermediate nodes will also forward the upstream data from their children to their parent node. In this case, the leaf nodes only need to handle the data generated by themselves while the intermediate nodes not only need to compress and transmit the data produced not only by themselves but also need to forward their R children's data to downstream nodes.

Figure 13 shows how small energy saving locally at a node can provide a considerable benefit to the intermediate nodes with more than one children in the tree structure. In the WSNs with a tree structure and R greater than 1, the packets the downstream nodes receive will increase exponentially when the number of hops increases. Also, with the help of the local compression algorithm, upstream nodes will generate fewer packets which will save the bandwidth at the intermediate nodes and mitigate the potential network congestion in WSNs.

6 Z-compression in low-power listening WSNs

Low-power listening WSNs are widely used in real-world applications because they could extend the lifetime of the network by reducing the duty cycle of the sensors in the network. In low-power listening WSNs, nodes periodically fall asleep to save the energy used while listening to the channel. In each period, the sensor nodes will wake up for a short-time to listen to the channel. If the node receives a packet, it will wake up for a longer-time to receive additional packets and to compute and perform routing. However, if the node does not receive any packets, it goes back to sleep after

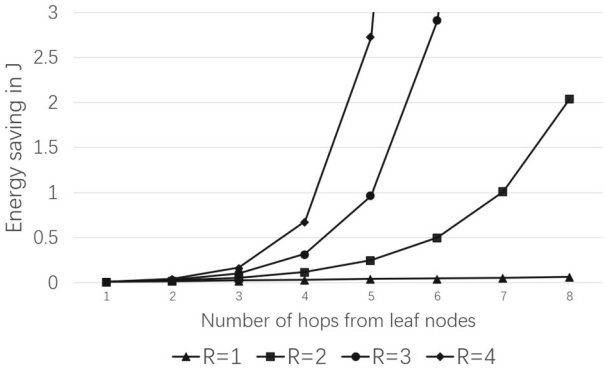


Fig. 13 Energy savings at intermediate nodes with different number of hops and children when compressing ZebraNet data using Optimized Z-compression versus no compression

the waiting time is over. In low-power listening WSNs, the only criteria to judge the energy consumption of the network is the duty cycle of all the sensor nodes in the WSNs. In this section, we are going to discuss how the Z-compression could reduce the duty cycles in the low-power listening WSNs as well as reduce the distortion (the difference between real data and sensing data) of the sensing data.

In a wireless sensor network with one sink node, we assume that the physical event delta value S_i has spatial correlation with the interested region S. The previous paper [42] modeled the physical phenomenon as joint Gaussian random variables (JGRVs) at each observation point i with zero mean and with σ_S as the variance. The observed sample at node i will be the sum of the physical event's value S_i plus the observation noise N_i which has zero mean and variance σ_N. The measured distortion can be calculated using the following equation:

$$D_E(M) = E[(S - \hat{S}(M))^2]$$

Here S is the true value and S(M) is the reporting value. Based on [43], we define the distortion function (22) where $D_E(M)$ is the distortion of all the reporting message M.

$$\begin{aligned} D_E(M) = \sigma_S^2 + \frac{\sigma_S^4}{M(\sigma_S^2 + \sigma_N^2)} \left(2 \sum_{i=1}^{M} \rho_{(S,i)} - 1 \right) \\ + \frac{\sigma_S^6}{M(\sigma_S^2 + \sigma_N^2)^2} \sum_{i=1}^{M} \sum_{j \neq i}^{M} \rho_{(S,i)} \end{aligned} \tag{22}$$

From the Eq. 22, we find that it is not necessary to collect all the data from the location of interest to keep the distortion low. If we choose several reporting nodes wisely from the area of interest, we can reduce the number of sensing nodes and still get highly reliable data.

In [43], the reporting nodes are selected using Lloyd's algorithm. The data from the reporting nodes send back to the sink node directly. However, in the low-power

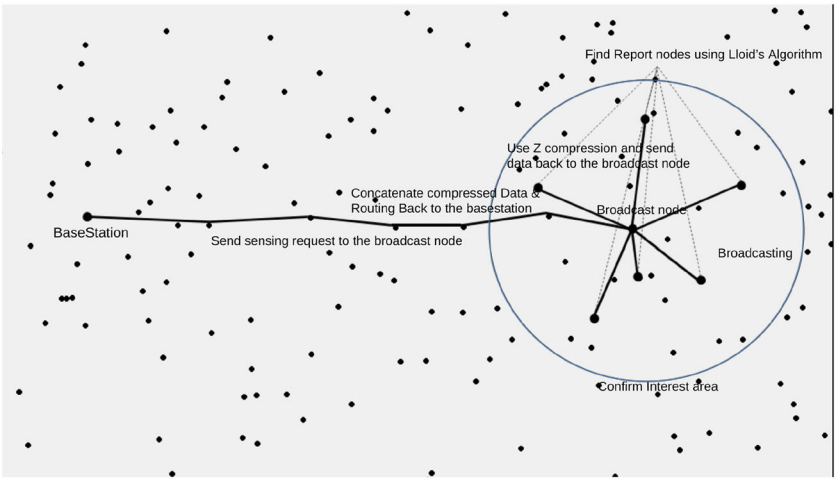


Fig. 14 Data collection in low power listening WSN using Z-compression along with data concatenation

listening WSNs, the routing nodes need to wake up for a period doing the radio transmission. More the routing path we use, more the energy will be used as the routing node will not go to sleep until it receives the acknowledgment from its successor. To reduce the usage of energy on the routing path, we can compress the sensing data using Z-compression and concatenate the reporting packets. Thus, we can reduce the duty cycles by reducing the wake-up time of the nodes on the routing path.

For example, in WSN in Fig. 14, the base station will select the reporting sensors using the Lloyd's algorithm. After broadcasting the request to the areas of interest, the reporting sensors will report the sensing data as follows. First, the reporting node will sense and compress the data locally using Z-compression. Second, the reporting nodes will send the compressed data back to the broadcast center which was selected by the base station. Next, the sensing data will be concatenated in the broadcast center and then send back to the base station through the reliable route (which uses the same route the request message used).

7 Experimental evaluations

We evaluated and compared compression algorithms using the following three types of experiments involving high-stream rate WSNs' data compression, local block data compression, and data compression in low-power listening WSNs.

7.1 Z-compression in high stream rate WSNs

The size of the data packets in wireless sensors is usually restricted to 128 bytes with 250kbps bandwidth, but the data streaming rate of each packet can range from seconds to hours. For example, the ZebraNet senses location data every few

minutes. The Intel Berkeley Lab sensor network application collects temperature, humidity, and light lumen at different periods whereas the accelerometer on the vehicle collects 3-axis acceleration values, and the vehicle tracking application collects communicating vehicle's location, speed, and altitude data much more frequently.

7.1.1 Experimental setup and configurations

To demonstrate the effectiveness of our proposed Z-compression scheme in real-world situation, we tested it against different types of real-world multi-modal data sets such as GPS data [44], environmental data [27], accelerometer data [15], and vehicle tracking data [10]. The attributes in each dataset are shown in Table 4. Two common attributes which these data sets share are timestamp and node ID. These two attributes are used as the primary key of the local packet. We cannot compress the primary key because the intermediate nodes need to identify where the packet has come from as well as when the sampling starts. We assign the timestamp as fixed two bytes and the node ID as fixed one byte in the local packet, and we use a variable length packet for the compressed sensing values. The compression ratio CR in the performance matrix is defined as the compressed data length over the size of uncompressed data as shown in Eq. 23. We use a base station to accumulate the number of bytes of all n compressed packets. $\sum_1^n L_{compressed}$ is the compressed data size and $\sum_1^n L_{original}$ is the original data size.

$$CR = \frac{\sum_1^n L_{original}}{\sum_1^n L_{compressed}} \tag{23}$$

To find the energy cost of the intermediate nodes, we use PowerTOSSIM-Z to simulate the energy consumption in WSN. The tool will calculate the CPU cycles and radio usage at each node. It then uses the predefined power model to generate the power consumption at each node in the experiment. Data is inserted into the leaf nodes using the python script.

We notice that when we increase the sampling rate close to a certain interval of two consecutive sensing samples, the packets drop start happening at some intermediate nodes, and at the sink node. Here, we define the sampling rate as the total number of sensing samples per second. To find the effectiveness of the compression algorithms on the sampling rate, and the maximum sampling rate a WSN can have, we define Eq. 24 which outputs the approximate maximum sampling rate of the WSN. Here $T_{ap} = 30.31\%$ is the maximum experimentally normalized throughput of IEEE 802.15.4 radio in application layer [8], $V_{ch} = 250kbps$ is the channel speed of the radio [7], S_{data} is the uncompressed size of an original sensing sample and CR is the compression ratio of respective compression algorithm.

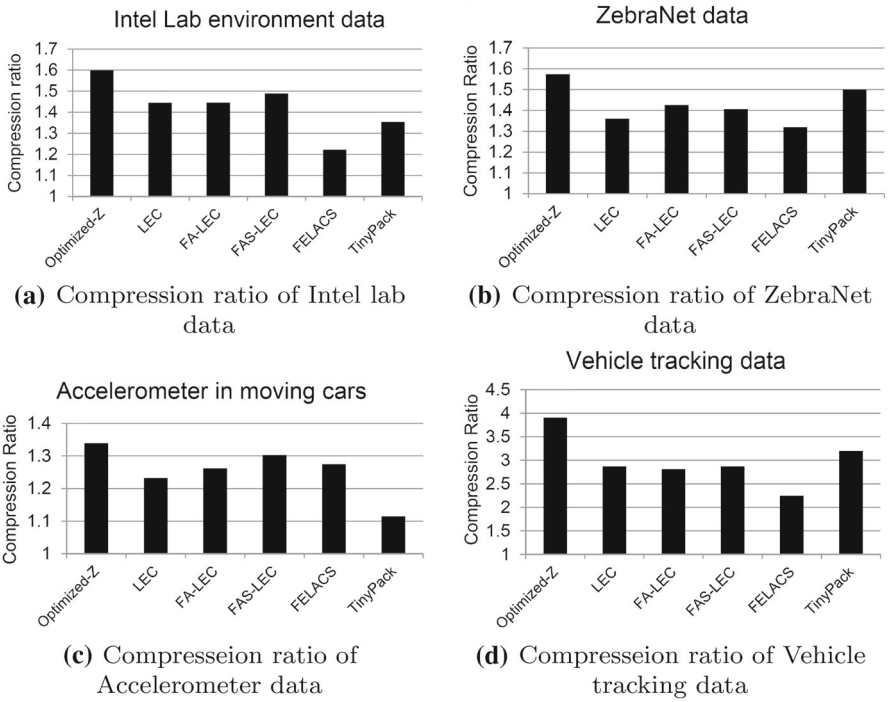
$$sample\ Rate_{max} \approx \frac{CR \times T_{ap} \times V_{ch}}{S_{data}} \tag{24}$$

7.1.2 Compression performance comparison

This experiment evaluates the average compression ratio in compressing 5000 data items from each of the four different datasets listed before. The leaf nodes will do the compression locally. Then the compressed packets are concatenated at the intermediate nodes using the Algorithm 4. The evaluation results are shown in Fig. 15. The compression ratios in Y-axis is calculated using the Eq. 23. Note that a higher compression ratio means better compression performance.

On compressing vehicle trace dataset, Z-compression has more than 30% improvement over other compression algorithms. On compressing other three datasets, Z-compression has between 5% to 30% improvement over other compression algorithms. Z-compression has better compression ratio improvement when using the vehicle trace dataset because this dataset has more unchanged samples than others. Z-compression also has an all-is-well function that can reduce the multiple zero delta values into a single Z-value of zero. Also, the performance of Z-compression is stable whereas we can see that other compression algorithms, for example, TinyPack and LEC, have large performance variation when the input data distribution changes. LEC performs better for the dataset with smaller delta values. However, TinyPack has a better compression ratio over the datasets which have more smaller delta values. Adaptive-LEC improved the compression performance by using the real-time adaption. However, when adapting multi-modal sensing data, there can be more than two frequency center exists like what Fig. 9 shows. That limits the performance of Adaptive-LEC. FELACS has the worst performance because it is not good at compressing multi-modal sensor data. It has the same drawback as Naive Z-compression. That is, when compressing skewed data, both Naive Z-compression and FELACS need to add '0' at the front of the smaller delta values to make their length equal to the length of the largest delta value. For example, in the Intel Berkeley lab environment dataset, the length of the delta values of the humidity is always larger than the length of the delta values of the light intensity. For the multi-modal sensing data, the distribution of the length of each attribute can be different. That means the dataset is prone to be skewed like Fig. 9. That is the reason FELACS can not perform well on compressing real-world multi-modal datasets.

The result shows that the TinyPack achieves a higher compression ratio in compressing dataset with many "zero" delta values like in ZebraNet data in Fig. 15b and Vehicle tracking data in Fig. 15d. However, in compressing ZebraNet dataset, Optimized Z-compression algorithm achieves a higher compression ratio than Naive Z-compression and has about 15% improvement over the next best TinyPack algorithm. On compressing Intel Berkeley dataset, LEC achieves a higher compression ratio than Naive Z-compression. The reason is that the light lumen values in the IntelLab dataset do not change as frequently as the temperature and humidity values do, so the dataset is skewed. Thus, the delta values of temperature and humidity are much higher than delta values of the light lumen. However, here our Optimized Z-compression groups temperature and light lumen together and then compressed with the humidity data. Thus, the result in Fig. 15a shows that optimized Z-compression gains about 10% improvement over the next best LEC.



(a) Compression ratio of Intel lab data

(b) Compression ratio of ZebraNet data

(c) Compresseion ratio of Accelerometer data

(d) Compresseion ratio of Vehicle tracking data

Fig. 15 Compression ratio of real-time datasets

On compressing Vehicle tracking data in Fig. 15d, we use the small code library and odd/even Optimized Z-compression. We notice that the attributes of the vehicle tracking dataset are evenly distributed with the average number of bits of the delta value close to 1. So for the eight attributes of the dataset, we generate a rule to compress pairs of attributes first; then compress pairs of encoded values recursively till only one Z-value is obtained. From Table 1, we can compress two zero delta values, which is represented as '1', into one bit Z-value of '1'. If all eight attributes of vehicle tracking data are zero, the final encoded result will be '11' (The first '1' is the placeholder). Therefore, Optimized Z-compression gets about 50% improvement over next best TinyPack.

Accelerometer dataset has an evenly distributed delta values. Therefore, Fig. 15c shows that Naive Z-compression achieves good performance on the compression ratio with an improvement of about 18.4% over LEC and 30% over TinyPack. However, as we discussed in the last section, the Optimized Z-compression uses the same compression rule as Naive Z-compression so the performance of Optimized Z-compression in Fig. 15c is same as of Naive Z-compression.

7.1.3 Energy usage comparison

These experiments are performed with TOSSIM simulator using PowerTOSSIM-Z [33]. We inserted data at the leaf nodes using the python script again. For some datasets

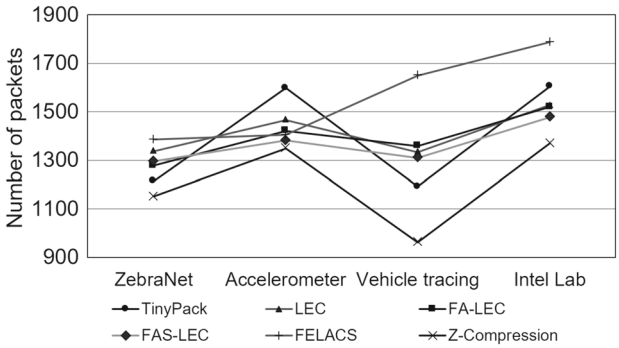


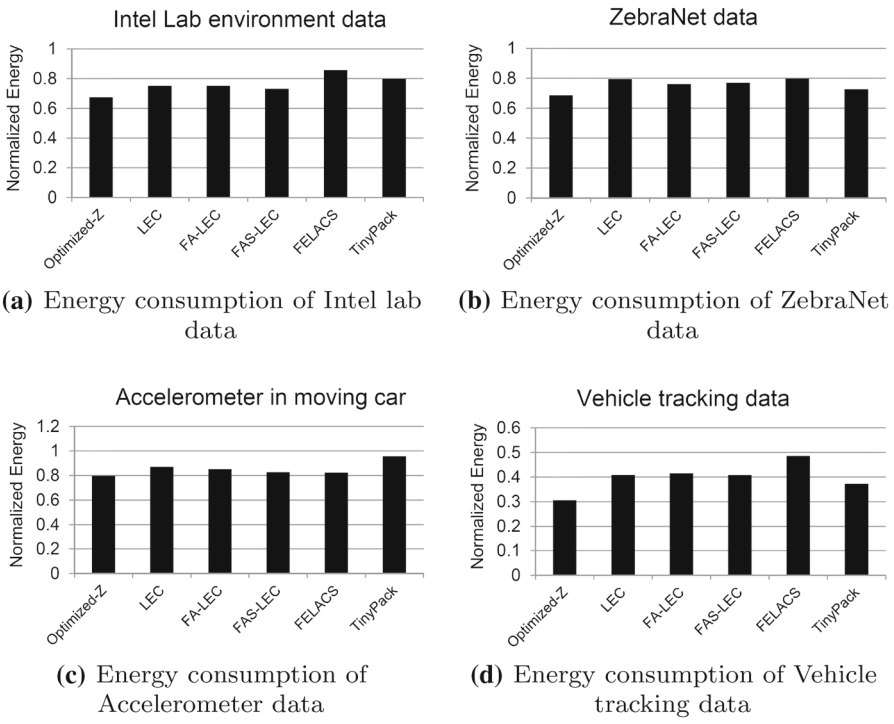
Fig. 16 Total packets after compression and concatenating for 20,000 data samples

such as Intel Lab and vehicle trace dataset, we inserted about 20,000 samples each time. After compression and concatenating, the number of packets forwarded to the sink is much smaller than the original 20,000 samples. The result is shown in Fig. 16. As the compression ratio of Z-compression is better than the other compression algorithms for all the four datasets, the Z-compression reduces more packets than all others and thus, saving more energy and also it reduces the bandwidth usage in the network. Note that as we are comparing different datasets and each dataset has a different number of samples in the energy comparison results, thus we use the normalized energy instead the real energy cost to show the effectiveness of each compression algorithm. The normalized energy is the ratio of the energy consumed by compressing or concatenating data and transmitting the fused packets over the energy consumed by only transmitting the fused packets. The experimental results are shown in Fig. 17; the result shows that Z-compression provides the best energy saving for the WSN. It is because with the better compression ratio achieved with Z-compression, the intermediate nodes can concatenate much more leaf node data into a larger packet that reduces the radio usage which saves battery. The fact that the intermediate nodes do not perform the compression, and the overhead due to concatenating leaf node's payload is negligible, thus, we do not show the CPU energy usage in the result.

7.1.4 Approximate maximum sampling rate

As we discussed in Sect. 5, the maximum sampling rate is the rate at which the maximum throughput of the sink node can be supported without dropping packets. It mainly depends on the compression ratio of the leaf nodes. The compression time will only determine the minimum sample interval of the leaf nodes in the WSN. The maximum sampling rate will not be affected by the compression time as we can increase the number of leaf nodes. Also, the time complexity of all these compressing algorithms is O(n). It takes about 30–60 milliseconds for compressing each sample including the sampling time.

Table 3 shows the approximate maximum sampling rate for compressing and concatenation of different data sets versus direct forwarding. The optimal Z-compression has the best performance according to the results.



(a) Energy consumption of Intel lab data

(b) Energy consumption of ZebraNet data

(c) Energy consumption of Accelerometer data

(d) Energy consumption of Vehicle tracking data

Fig. 17 Energy consumption of real-time datasets

Table 3 Maximum approximate sampling rate using different compression algorithms with data concatenating

Data set name	Z-comp	LEC	FA-LEC	FAS-LEC	FELACS	TinyPack	No concat
Intel Lab environment data	1376	1244	1245	1282	1053	1165	131
Accelerator in moving car	1409	1297	1328	1371	1341	1172	136
ZebraNet data	1656	1431	1500	1479	1389	1578	136
Vehicle trace data (V to V)	1947	1430	1403	1430	1119	1594	117

7.2 Local block data compression

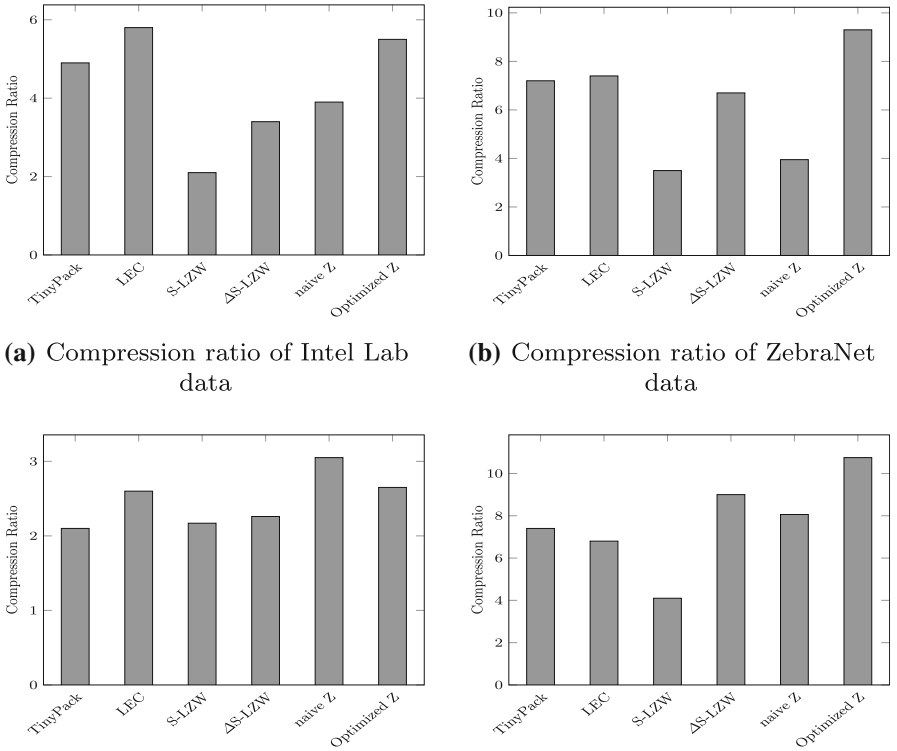
In wireless sensor networks, energy is the most critical factor for the lifetime of the network. Most of the power consumed by wireless sensor nodes is due to data transmissions using radio communication. The time radio is on mainly depends on the number of packets to be transmitted, which in turn depends on the packet size. Thus, the compression algorithm which achieves a better compression ratio will usually have a better energy saving due to the reduced radio transmission time as it will send

fewer packets. However, the microcontroller also consumes energy in compressing data as well. Thus, it is essential that we not only consider the compression ratio as the performance metric but also compare the energy savings of compression methods.

For the experiment here, different from the last which uses real-time datasets, the system model of the evaluation here is an only one-hop delay-tolerant wireless sensor network. We initialize two blocks of data as byte arrays. Each block is 528 bytes containing a sequence of continuous timestamp data. Each data item is composed of timestamp and sensing values. To exploit the temporal locality property of sensing attributes, We regrouped data by sensor category and types before compressing. For example, when compressing ZebraNet data, we group based on timestamp, the longitude, the latitude and the boolean values separately based on the sensor types. For the case without compression, local energy consumption is the energy required to transmit the whole block of data. For the node doing compression, the energy consumption is the sum of the energy needed to transmit the compressed data and the energy used in compressing. We define the energy saving as the energy consumed when compressing and transmitting data over the energy consumed in transmitting uncompressed data immediately.

The compression algorithms we evaluated in this part are S-LZW (as it works traditionally on data blocks), *Delta-S-LZW*, LEC, TinyPack, Naive Z-compression and *Three-round Z-compression*. The size of the output packet is 114 bytes which are suitable for the TelosB mote. For S-LZW and *Delta-S-LZW*, we use the dictionary with 1024 entries. We propose here Delta-S-LZW that compresses the delta values rather than the actual values. It exploits the temporal locality and gives us a higher compression ratio as shown in Fig. 18 and better energy saving as shown in Fig. 19 than S-LZW. For TinyPack and LEC, Huffman code is generated respectively based on Fig. 1. For Naive Z-compression and Three-round Z-compression, to reduce the compression overhead, we split the output packet into smaller blocks of the same size. The number of blocks under a packet is predefined. In this experiment, we set this to be 6. After generating all the blocks of a packet using Naive Z-compression, we then merge those blocks to get the output packet. Note that each block needs extra one byte to indicate how many data items are in the block. Next, *Three-round optimized Z-compression* is proposed to improve compression performance on the datasets with many delta values as '0'. In this case, we compress a sequence of continuous '0' delta values together using LEC to encode the number of '0's. Then we apply Naive Z-compression on the newer dataset generated in the first round. We call this optimization Three-round Z-compression algorithm because it uses three rounds to get the result. Note that the first round that handles a sequence of '0' bits will add extra one bit to the delta values which are not 0s.

We regrouped sensor data by category and types before compressing. For example, for ZebraNet data, we group timestamp, the longitude, the latitude and the boolean values separately based on sensor types. For each group, temporal locality property is exploited. To reduce the compression overhead when using Z-compression, we set the sub-group size first and then compress data into groups with the size less than the sub-group size. After generating all the sub-groups, we apply Naive Z-compression on those sub-groups to get the final output. We also refer to this as Two-round Naive Z-compression to indicate that it uses two rounds of compression.



(a) Compression ratio of Intel Lab data

(b) Compression ratio of ZebraNet data

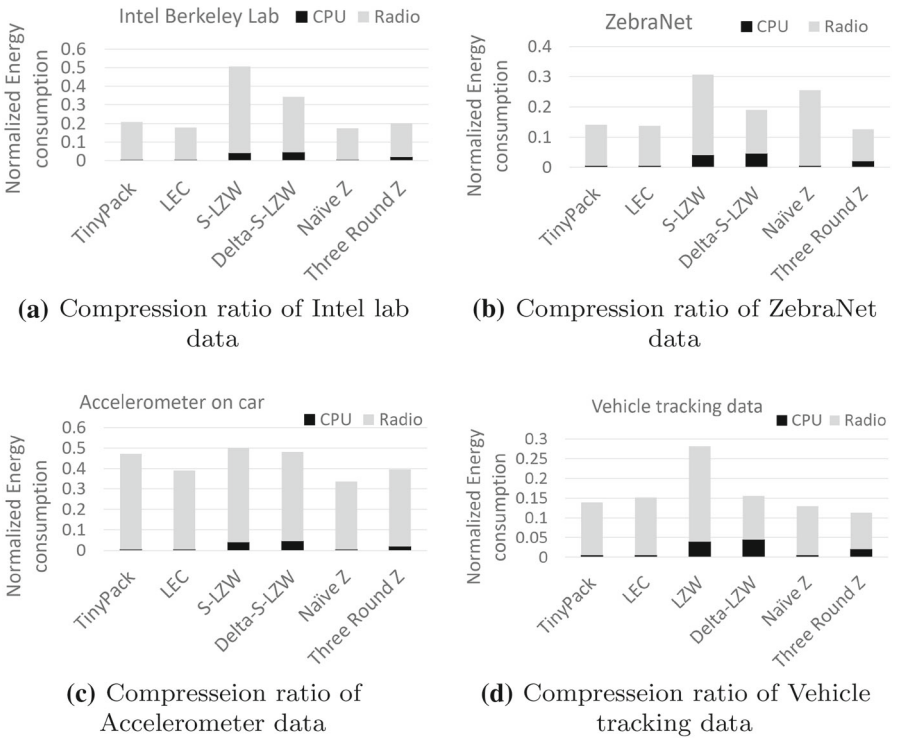
(c) Compresseion ratio of Accelerometer data

(d) Compresseion ratio of Vehicle tracking data

Fig. 18 Compression ratio of different local datasets

Compressing large blocks of local data is different from compressing smaller data packets. First, delays are not considered in this case. Second, the compressing produces data packets which have size limitations. Third, the time stamp is included. Fourth, the format of input data is the byte array.

Figure 18 shows the local compression ratio of four different kinds of datasets with five different compression algorithms. And, Fig. 19 shows the normalized energy consumption for different compression algorithms using four different datasets. Naive Z-compression beats all the other compression algorithms on compression ratio and energy saving in compressing Intel lab and Accelerometer datasets shown in Figs. 18a, c and 19a, c for the reason that Naive Z-compression has less overhead in compressing evenly distributed dataset. Three-round Z-compression beats all the other compression algorithms on compression ratio and energy saving for compressing ZebraNet and Vehicle tracking datasets showing in Figs. 18b, d and 19b, d for the reason that it will decrease overhead by encoding sequence of '0' delta values into a single LEC code. Three-round Z-compression has better performance in compressing Vehicle tracking dataset than compressing ZebraNet dataset because Vehicle tracking dataset contains



(a) Compression ratio of Intel lab data

(b) Compression ratio of ZebraNet data

(c) Compresseion ratio of Accelerometer data

(d) Compresseion ratio of Vehicle tracking data

Fig. 19 Energy consumption in compressing different datasets

many '0' delta values than ZebraNet. The more '0' delta values a dataset contains, the more better Three-round Z-compression performs.

Datasets such as Accelerometer and Intel lab which contain fewer '0' bits items are not suitable for Three-round Z-compression algorithm, which is also validated by Fig. 19a, c. As explained above, for these two datasets, Naive Z-compression is the best. Thus, our proposed Z-compression schemes also perform well for not so real-time time case. Although Delta-S-LZW always achieves better compression ratio, it is still not as good as other delta compression algorithms. It shows that delta compression has an advantage over local sensor data compression.

7.3 Experiments and evaluations of optimal Z-compression in LPL WSNs

7.3.1 Experimental setup and configurations

In this experiment, we use TOSSIM to simulate the wireless sensor networks. We use PowerTOSSIM-Z to measure the energy consumption of each node. The sensors are deployed in a rectangle area with length $\times$ width equal to 3600×2000. We assume that the area is homogeneous and the sensors hold the same radio range in the area. The network is not sparse, and the sensors are uniformly deployed in the rectangle

Table 4 Sensors deployment parameters

Type	Experiment 1	Experiment 2
Number of nodes	990	2500
Area to deploy	3600 × 2000	3600 × 2000
Radio range	300	200
Energy model	micaz	micaz
Sleep period	2 second	2 second
Duty cycle	10%	10%
Area of interest	Radius = 1000	Radius = 600
# of reporting nodes	10	10
# of sensing attributes	10	10

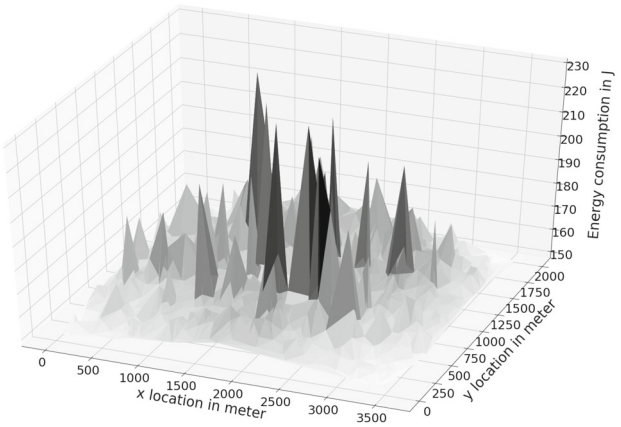


Fig. 20 Energy usage of 990 node WSN in 10,000 s with 1250 sensing requests with no compression

area. By configuring the range of the radios of the sensors, we can make sure that each sensor node has an average of six to ten direct neighbors. The sensor deployment rule is shown in Table 4. We did two experiments with a different number of nodes. In each experiment, we compare the energy consumption at each node using direct sensing and reporting the energy consumption using the compression and concatenation.

As TOSSIM does not perform real sensing, to simulate the data collection and compression, we insert data at each node. One way is to use the data injection in TOSSIM. However, this data injection will increase the duty cycle of the sensor's which makes the simulation results incorrectly. So we hard-coded the sensing data in the header file and refer the header file from the NesC code of the sensors based on the node ID. Also, to ensure the routing accuracy, each sensor has also hard-coded its neighbors' ID in their source code. The 990 sensor nodes' result is shown in Figs. 20 and 21. And, the 2500 sensor nodes' result is shown in Figs. 22 and 23.

The simulation results using 990 nodes show that the optimal Z-compression along with the data concatenation reduces the average energy consumption by about 9%. It also balances the load of the network significantly by reducing the load on the

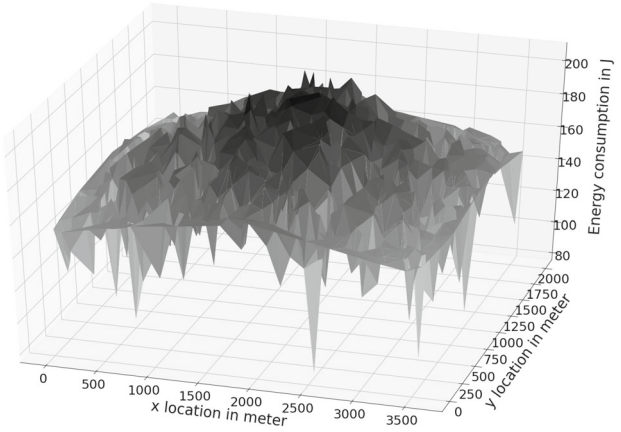


Fig. 21 Energy usage of 990 nodes WSN in 10,000 s with 1250 sensing requests with optimal Z-compression along with data concatenation

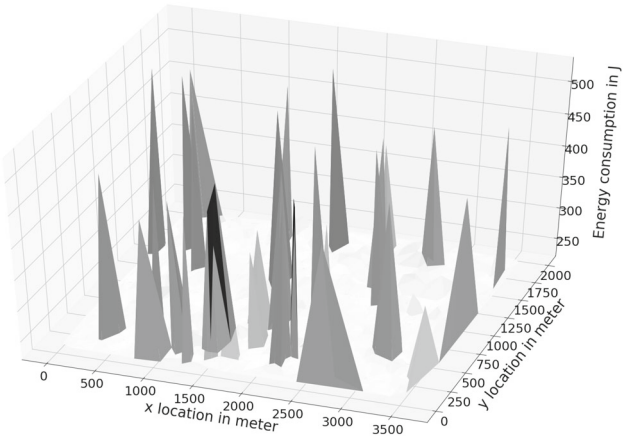


Fig. 22 Energy usage of 2500 nodes WSN in 10,000 s with 1250 sensing requests with no compression

intermediate nodes. Figure 21 is smoother than Fig. 20 which directly transmits the reporting message back to the base station without compression. The improvement reaches about 24% at the nodes with peak loads. It significantly increases the lifetime of the whole network.

In the result with 2500 nodes in Figs. 22 and 23, when using the optimal Z-compression, the average energy saving reaches 26%. The nodes with peak load have about 24% energy saving. We notice that the average energy saving for the 2500 nodes is more significant than that for the 990 nodes. The reason is that for both of these WSNs, the number of reporting nodes is the same. However, the number of hops in the path from the reporting nodes to the sink is more for the 2500 nodes than that of 990 nodes. Thus, Optimal Z-compression saves much more energy as it reduces the

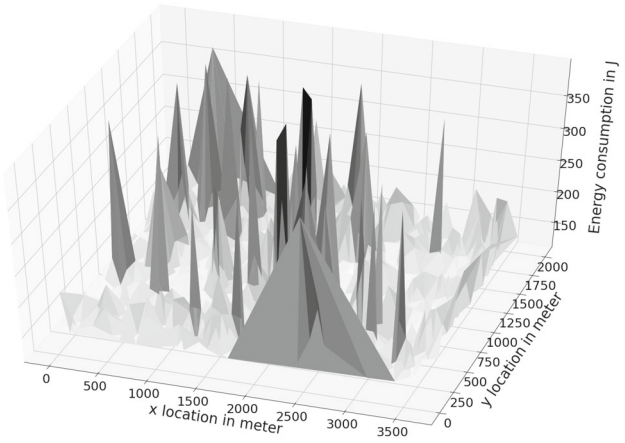


Fig. 23 Energy usage of 2500 nodes WSN in 10,000 s with 1250 sensing requests with optimal Z-compression along with data concatenation

duty cycle of the sensors on the routing path of the data which reduces the number of reporting nodes.

8 Conclusion and future work

In this paper, we proposed, based on Z-order, a multi-model Z-compression scheme for sensor data in wireless sensor networks to conserve bandwidth as well as energy. It compresses multi-dimensional data by exploiting the temporal and spatial locality. It reduces the packets size and allows the intermediate nodes to transmit less number of packets and thus, save energy, and being able to reduce the packet drops when streaming rate is high. We have performed several ToSSIM and TinyOS based experiments using four real-world sensor datasets. It performs much better when compared with well-known compression schemes like LEC, Adaptive-LEC and TinyPack using the compression ratio, energy usage and sampling rate as performance metrics. The Z-compression algorithm compresses multi-model sensing data locally in a real-time fashion, and thus, it can work with different MAC protocols to achieve further efficiency in WSNs. In the high-stream rate WSNs, Z-compression improves the throughput thus increase the maximum stream rate of the network. In the low-power wireless sensor networks using asynchronous MAC protocol such as low-power listening (LPL), Z-compression could reduce the duty cycle of the nodes in the path from where the reporting data routes back. The experimental results also demonstrated that Z-compression could not only save the energy and bandwidth but also balances the load in the WSNs which could prolong the lifetime of the sensor nodes.

In future, we plan to implement the Z-compression in a sensor cloud [28], which provides on demand sensing as a service to users satisfying the QoS. With the help of lossless Z-compression, we can handle the maximum sensing request rate from many different clients without additional delays, and also maintain the QoS requests of users. Similarly, in IoT-based applications such as smart-city, Z-compression can

adpat according to the device type and can handle the network heterogeneity as well to meet the application demands.

Funding Funding was provided by Directorate for Computer and Information Science and Engineering.

References

1. Ali, A., Khelil, A., Szczytowski, P., Suri, N.: An adaptive and composite spatio-temporal data compression approach for wireless sensor networks. In: Proceedings of the 14th ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems, pp. 67–76. ACM (2011)
2. Anchora, L., Capone, A., Mighali, V., Patrono, L., Simone, F.: A novel mac scheduler to minimize the energy consumption in a wireless sensor network. Ad Hoc Netw. **16**, 88–104 (2014)
3. Buettner, M., Yee, G.V., Anderson, E., Han, R.: X-mac: a short preamble mac protocol for duty-cycled wireless sensor networks. In: Proceedings of the 4th International Conference on Embedded Networked Sensor Systems, pp. 307–320. ACM (2006)
4. Buragohain, C., Shrivastava, N., Suri, S.: Space efficient streaming algorithms for the maximum error histogram. In: 2007 IEEE 23rd International Conference on Data Engineering, pp. 1026–1035. IEEE (2007)
5. Burgess, J., Zahorjan, J., Mahajan, R., et al.: CRAWDAD dataset umass/diesel (v. 2008-09-14) (2008)
6. Crossbow, M.: Telosb v2 data sheet. www.memsic.com/userfiles/files/Datasheets/WSN/telosb_datasheet.pdf (2008)
7. Datasheet, T.: Crossbow Inc. http://www.memsic.com/userfiles/files/Datasheets/WSN (2013)
8. Dopico, N.I., Gil-Soriano, C., Arrazola, I., Zazo, S.: Analysis of ieee 802.15. 4 throughput in beaconless mode on micaz under tinyos 2. In: Vehicular Technology Conference Fall (VTC 2010-Fall), 2010 IEEE 72nd, pp. 1–5. IEEE (2010)
9. Elmeleegy, H., Elmagarmid, A.K., Cecchet, E., Aref, W.G., Zwaenepoel, W.: Online piece-wise linear approximation of numerical streams with precision guarantees. Proc. VLDB Endow. **2**(1), 145–156 (2009)
10. Fujimoto, R.M., Guensler, R., Hunter, M.P., Wu, H., Palekar, M., Lee, J., Ko, J.: CRAWDAD dataset gatech/vehicular (v. 2006-03-15). http://crawdad.org/gatech/vehicular/20060315 (2006). https://doi.org/10.15783/C74S3Z
11. Gandhi, S., Nath, S., Suri, S., Liu, J.: Gamps: Compressing multi sensor data by grouping and amplitude scaling. In: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data, pp. 771–784. ACM (2009)
12. Huffman, D.A., et al.: A method for the construction of minimum-redundancy codes. Proc. IRE **40**(9), 1098–1101 (1952)
13. Hull, B., Jamieson, K., Balakrishnan, H.: Mitigating congestion in wireless sensor networks. In: Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems, pp. 134–147. ACM (2004)
14. Hung, N.Q.V., Jeung, H., Aberer, K.: An evaluation of model-based approaches to sensor data compression. IEEE Trans. Knowl. Data Eng. **25**(11), 2434–2447 (2013)
15. Jain, M., Singh, A.P., Bali, S., Kaul, S.: CRAWDAD dataset jiit/accelerometer (v. 2012-11-03) (2012). https://doi.org/10.15783/C76K58
16. Keogh, E., Chakrabarti, K., Pazzani, M., Mehrotra, S.: Locally adaptive dimensionality reduction for indexing large time series databases. ACM SIGMOD Rec. **30**(2), 151–162 (2001)
17. Kolo, J.G., Shanmugam, S.A., Lim, D.W.G., Ang, L.M.: Fast and efficient lossless adaptive compression scheme for wireless sensor networks. Comput. Electr. Eng. **41**, 275–287 (2015)
18. Krishnamachari, B., Estrin, D., Wicker, S.: The impact of data aggregation in wireless sensor networks. In: Distributed Computing Systems Workshops, 2002. Proceedings. 22nd International Conference on, pp. 575–578. IEEE (2002)
19. Kumar, G., Baskaran, K., Blessing, R.E., Lydia, M.: A comprehensive review on the impact of compressed sensing in wireless sensor networks. Int. J. Smart Sens. Intell. Syst. **9**(2), 818 (2016)
20. Künzel, H., Holm, A., Zirkelbach, D., Karagiozis, A.: Simulation of indoor temperature and humidity conditions including hygrothermal interactions with the building envelope. Sol. Energy **78**(4), 554–561 (2005)

21. Kuo, T.W., Lin, K.C.J., Tsai, M.J.: On the construction of data aggregation tree with minimum energy cost in wireless sensor networks: Np-completeness and approximation algorithms. IEEE Trans. Comput. **65**(10), 3109–3121 (2016)
22. Levis, P., Lee, N., Welsh, M., Culler, D.: Tossim: Accurate and scalable simulation of entire tinyos applications. In: Proceedings of the 1st International Conference on Embedded Networked Sensor Systems, pp. 126–137. ACM (2003)
23. Lim, J.B., Jang, B., Sichitiu, M.L.: Mcas-mac: a multichannel asynchronous scheduled mac protocol for wireless sensor networks. Comput. Commun. **56**, 98–107 (2015)
24. Liu, C.X., Liu, Y., Zhang, Z.J., Cheng, Z.Y.: High energy-efficient and privacy-preserving secure data aggregation for wireless sensor networks. Int. J. Commun. Syst. **26**(3), 380–394 (2013)
25. Long, J., Dong, M., Ota, K., Liu, A.: A green tdma scheduling algorithm for prolonging lifetime in wireless sensor networks. IEEE Syst. J. **11**(2), 868–877 (2017)
26. Lymberopoulos, D., Priyantha, N.B., Goraczko, M., Zhao, F.: Towards energy efficient design of multi-radio platforms for wireless sensor networks. In: International Conference on Information Processing in Sensor Networks, IPSN'08
27. Madden, S.: Intel berkeley research lab data (2003)
28. Madria, S., Kumar, V., Dalvi, R.: Sensor cloud: a cloud of virtual sensors. Softw. IEEE **31**(2), 70–77 (2014)
29. Marcelloni, F., Vecchio, M.: An efficient lossless compression algorithm for tiny nodes of monitoring wireless sensor networks. Comput. J. **52**(8), 969–987 (2009)
30. McDonald, D., Sanchez, S., Madria, S., Ercal, F.: A survey of methods for finding outliers in wireless sensor networks. J. Netw. Syst. Manag. **23**(1), 163–182 (2015)
31. Medeiros, H.P., Maciel, M.C., Demo Souza, R., Pellenz, M.E.: Lightweight data compression in wireless sensor networks using huffman coding. Int. J. Distrib. Sens. Netw. (2014)
32. Morton, G.M.: A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing. International Business Machines Company, New York (1966)
33. Perla, E., Catháin, A.Ó., Carbajo, R.S., Huggard, M., Mc Goldrick, C.: Powertossim z: realistic energy modelling for wireless sensor network environments. In: Proceedings of the 3nd ACM Workshop on Performance Monitoring and Measurement of Heterogeneous Wireless and Wired Networks, pp. 35–42. ACM (2008)
34. Rhee, I., Warrier, A., Aia, M., Min, J., Sichitiu, M.L.: Z-mac: a hybrid mac for wireless sensor networks. IEEE/ACM Trans. Netw. (TON) **16**(3), 511–524 (2008)
35. Sadler, C.M., Martonosi, M.: Data compression algorithms for energy-constrained devices in delay tolerant networks. In: Proceedings of the 4th International Conference on Embedded Networked Sensor Systems, pp. 265–278 (2006)
36. Shannon, C.E.: A mathematical theory of communication. ACM SIGMOBILE Mob. Comput. Commun. Rev. **5**(1), 3–55 (2001)
37. Sun, Y., Gurewitz, O., Johnson, D.B.: Ri-mac: a receiver-initiated asynchronous duty cycle mac protocol for dynamic traffic loads in wireless sensor networks. In: Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems, pp. 1–14. ACM (2008)
38. Szalapski, T., Madria, S.: On compressing data in wireless sensor networks for energy efficiency and real time delivery. Distrib. Parallel Databases **31**(2), 151–182 (2013)
39. Szalapski, T., Madria, S.: Energy efficient distributed grouping and scaling for real-time data compression in sensor networks. In: 2014 IEEE 33rd International Performance Computing and Communications Conference (IPCCC), pp. 1–9. IEEE (2014)
40. Vecchio, M., Giaffreda, R., Marcelloni, F.: Adaptive lossless entropy compressors for tiny iot devices. IEEE Trans. Wirel. Commun. **13**(2), 1088–1100 (2014)
41. Vitter, J.S.: Design and analysis of dynamic huffman codes. J. ACM (JACM) **34**(4), 825–845 (1987)
42. Vuran, M.C., Akan, Ö.B., Akyildiz, I.F.: Spatio-temporal correlation: theory and applications for wireless sensor networks. Comput. Netw. **45**(3), 245–259 (2004)
43. Vuran, M.C., Akyildiz, I.F.: Spatial correlation-based collaborative medium access control in wireless sensor networks. IEEE/ACM Trans. Netw. (TON) **14**(2), 316–329 (2006)
44. Wang, Y., Zhang, P., Liu, T., Sadler, C., Martonosi, M.: CRAWDAD dataset princeton/zebranet (v. 2007-02-14). http://crawdad.org/princeton/zebranet/20070214 (2007). https://doi.org/10.15783/C77C78
45. Welch, T.A.: A technique for high-performance data compression. Computer **17**(6), 8–19 (1984)

Affiliations

Xiaofei Cao[1] · Sanjay Madria[1] · Takahiro Hara[2]

Xiaofei Cao
xc9pd@mst.edu

Takahiro Hara
hara@ist.osaka-u.ac.jp

[1] Department of Computer Science, Missouri University of Science and Technology, Rolla, MO 65401, USA

[2] Department of Multimedia Engineering, Osaka University, Osaka, Japan