29 Nov 2011

Energy-efficient Real-time Data Compression in Wireless Sensor Networks

Tommy Szalapski

Sanjay Madria
Missouri University of Science and Technology, madrias@mst.edu

Recommended Citation

T. Szalapski and S. Madria, "Energy-efficient Real-time Data Compression in Wireless Sensor Networks," *Proceedings - IEEE International Conference on Mobile Data Management*, vol. 1, pp. 236 - 245, article no. 6068443, Institute of Electrical and Electronics Engineers, Nov 2011.
The definitive version is available at https://doi.org/10.1109/MDM.2011.45

2011 12th IEEE International Conference on Mobile Data Management

Energy-efficient Real-Time Data Compression in Wireless Sensor Networks

Tommy Szalapski and Sanjay Madria
Department of Computer Science, Missouri S&T, Rolla, MO 65401, USA
T.M.Szalapski@mst.edu and madrias@mst.edu

Abstract -**Wireless sensor networks possess significant limitations in storage, bandwidth, and power. Additionally, real-time sensor networks cannot tolerate high latency. While some good compression algorithms exist specific to sensor networks, in this paper we present an energy-efficient method with high-compression ratio that reduces latency, storage and bandwidth usage further in comparison with some other recently proposed algorithms. Our Huffman style compression scheme exploits temporal locality and delta compression to provide better bandwidth utilization in the network, thus reducing latency for real time applications. Our performance evaluations show comparable compression ratios and energy savings with a significant decrease in latency compared to some other existing approaches.**

Keywords - wireless sensor network; real-time; compression

I. INTRODUCTION

Many real-time systems incorporate wireless sensors into their infrastructure. For example, some airplanes and automobiles use sensors to monitor the health of different physical components in the system, security systems use sensors to monitor boundaries and secure areas, armies use sensors to track troops and targets. It is well known that wireless sensor networks possess significant limitations in processing, storage, bandwidth, and power. Therefore a need exists for efficient data compression algorithms which do not require delays in processing or communication while still reducing memory and energy requirements.

Data compression has existed since the early days of computers [1][2][3]. Many new compression schemes [5][6][7][8][9] for wireless sensor networks have been proposed. These schemes address specific challenges and opportunities presented by sensor data and provide significant reductions in required storage, bandwidth, and power. However, most of these methods require a fair amount of data to be collected before compressing.

We propose TinyPack, a compression scheme for real-time sensor networks. TinyPack reduces the amount of data flowing through the network without introducing delays. First the data is transformed by expressing the sensed values as the change in value from the previous sensed reading. This is referred to as delta compression. We demonstrate its effectiveness for any generic real-time sampled dataset. Second, the individual delta values are then compressed using a derivative of Huffman coding [1]. Huffman codes express more frequent data values with shorter bit sequences and less frequent values with longer ones. The codes are generated and updated dynamically so no delay is needed. TinyPack is a lossless compression algorithm and the data can be decompressed at the sink or base station without any loss of granularity or accuracy.

Standard Huffman and Adaptive Huffman [2] coding have a high RAM overhead and require transmitting either the entire tree or several copies of a 'new symbol' code. We begin with a static initial code set similar to the one used in the LEC algorithm [8]. We then examine two different methods of adapting the codes. For datasets where the range of possible values is relatively low compared to the storage capability of the sensors, the actual frequencies can be counted and used to regularly update the codes. For data with a high (or unknown) variance or low RAM environments the frequencies can be approximated using running statistics on the data stream. This method easily scales to be effective on any size data set with any range of possible values. We introduce the notion of an all-is-well bit and perform initial analysis of error detection constructs.

We compare the results to the performance of the Deflate algorithm (used in gzip and most operating systems) and S-LZW [7] to measure quality of the compression. S-LZW is an adaptation of standard LZW compression specifically designed for sensor networks. S-LZW is a string based compression scheme which defines new characters for common sequences of characters. It is designed to function well for any generic sensor dataset and is very effective at compression and energy reduction. Several variations of S-LZW are developed in [7]. In an effort to be fair we have chosen the variation that performs best for each dataset studied. We also compare with the LEC algorithm [8] which supports real-time data.

In summary, this paper makes the following contributions:

- An improved set of static codes optimized for sensor data and efficiency in processing
- Hybrid adaptations of delta and Huffman compression which significantly reduce latency and RAM requirements over traditional Huffman codes while achieving comparable and improved compression ratios and energy efficiency compared to other existing methods
- An additional all-is-well bit construct that further increases compression performance and efficiency
- A novel and effective error detection method

This research is supported by DOE grant number P200A070359.

978-0-7695-4436-6/11 $26.00 © 2011 IEEE
DOI 10.1109/MDM.2011.45

II. BACKGROUND

A. Huffman Trees

Huffman-style coding [1] converts each possible value into a variable length string (sequences of bits) based on the frequency of the data. Higher frequency values are assigned shorter strings. So the more concentrated the data is over a small set of values, the more the data can be compressed. Huffman codes can be generated by building a binary tree where the nodes at each level are ideally half as frequent as the nodes at the next level up. For example, the values and frequencies in Table I generate the codes using the Huffman tree in Figure 1. Huffman codes were shown to be optimal for symbol by symbol compression in [1].

Table I HUFFMAN CODES

Value	*Frequency*	*Code*
-7	14653	111111
-6	16661	111101
-5	19983	111011
-4	23760	111001
-3	31124	11011
-2	35636	11001
-1	88845	101
+0	350429	0
+1	87956	100
+2	38942	11000
+3	31809	11010
+4	20563	111000
+5	17241	111010
+6	14171	111100
+7	12716	111110

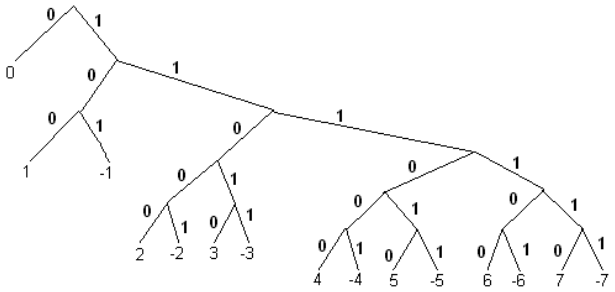

Figure 1 Huffman Tree

B. Temporal locality and delta values

Real-time wireless sensor networks generally exhibit temporal locality (data from readings taken in a small time window are correlated). Any type of data which changes in a continuous fashion will be temporally located such as temperature, location, voltage, velocity, timestamps, etc. In fact, it can be demonstrated that any sensor sensing at non-random intervals will either generate temporally located data or random noise.

Consider an arbitrary sensor sensing a stream of values $\{v_1, v_2, \ldots, v_{2N}\}$ sensed at times $\{t_1, t_2, \ldots, t_{2N}\}$ where N is an integer. Assume the values are not correlated. Then sampling at $\{t_1, t_3, \ldots, t_{2N-1}\}$ and $\{t_2, t_4, \ldots, t_{2N}\}$ would yield completely different values. So offsetting the sample period would generate entirely different data.

Therefore, excluding applications which generate pure noise, we can assume that successive readings at each sensor will be correlated. Delta compression (storing the data as the change in value from the previous reading) would then increase the frequency of certain values thus increasing the compressibility of the data.

Note that this does not apply to event driven sampling (where time between samples is random) such as a sensor that measures the speed once for each passing automobile. These applications do not necessarily exhibit temporal locality and were not included in this study.

C. Frames

In delta compression (as with most compression schemes), a dropped packet can render following packets useless or at least complicated to decompress. So in systems where data loss is probable, data should be compressed and sent in chunks (usually called frames). Additionally, in sensor networks, data characteristics can change drastically as time progresses. So sending independently compressed frames of data also allows additional flexibility for the compression to be more specific to the current state of the system.

III. RELATED WORK

A. S-LZW

In [7] an adaptation of standard LZW compression is used to address the specific characteristics of a sensor network. S-LZW compresses the data by finding common substrings and using fewer bits to represent them. S-LZW maintains two sets of up to 256 eight-bit symbols: The original ASCII characters and the set of common strings. A bit is appended to the beginning of each encoded symbol to indicate which set it is from. A dictionary is maintained that tracks which string is represented by which eight-bit sequence.

They also propose Sensor-LZW with the notion of a mini-cache to capitalize on the frequent recurrences of similar values in a short time in sensor data. Recent strings are stored with N bits in the mini-cache dictionary where $N < 8$ (for a maximum size of 2^N entries in the mini-cache). An additional bit is appended to the beginning of each symbol to note whether the symbol is from the main dictionary or the mini-cache. Different data sets had different optimal values for N. The cache is implemented as a hash table for efficient lookup times.

Table II S-LZW WITH MINI-CACHE

Encoded String	New Output	New Dict. Entry	Mini-Cache Changes	Total Bits: LZW	Total Bits: Mini-Cache
A	65,0	256-AA	0-256, 1-65	9	10
AA	0,1	257-AAA	1-257	18	15
A	65,0	258-AB	1-65,2-258	27	25
B	66,0	259-BA	2-66,3-259	36	35
AAA	257,0	260-AAAB	1-257,4-260	45	45
B	2,1	261-BC	5-261	54	50
C	67,0	262-CC	3-67,6-262	63	60
C	3,1			72	65

Table II shows S-LZW and LZW compressing the string AAAABAAABCC. Every known symbol encountered is encoded into the output stream (choosing the longest string possible from the dictionary). Then a new dictionary entry is added by concatenating the next character in the input stream to the previously encoded symbol.

B. LEC

A lightweight sensor network compression technique, LEC, is presented in [8]. LEC compresses a stream of integers by encoding the delta values with a static, predetermined set of Huffman codes shown in Table III with anything past level 7 following the pattern of the last three levels.

Table III LEC CODES

Level	Bits	prefix	suffix range	values
0	2	00		0
1	4	010	0...1	-1.1
2	5	011	00...11	-3,-2,2,3
3	6	100	000...111	-7,...,-4,4,...,7
4	7	101	0000...1111	-15,...,-8,8,...,15
5	8	110	00000...11111	-31,...,-16,16,...,31
6	10	1110	000000...111111	-62,...,-32,32,...,63
7	12	11110	0000000...1111111	-127,...,-64,64,...,127

C. GAMPS

Many lossy compression schemes have also been proposed such as [9]. GAMPS compresses the data from multiple sensors which sense correlated data using mathematical techniques to group the sensors which have highest correlation to each other. One sensor in each group is selected as the baseline and the rest of the sensors in the group report the difference in their sensed values from the baseline. The values are rounded based on an error threshold parameter to achieve compressed sizes under 1% of the original size.

D. Routing methods

Other schemes have been introduced which depend on the network topology and routing [5][6]. In this paper, we focus on methods to perform lossless compression at a single sensor.

IV. EXPERIMENTAL DATA SETS USED

The data sets used for simulation were pulled from a wide variety of domains which utilize wireless sensor networks including environment monitoring, tracking, structural health monitoring, and signal triangulation. All except the environment monitoring data are from applications where low latency is critical. All are from real deployments of wireless sensors for academic, military, and commercial purposes. In every experiment, the entire datasets were used.

Environment monitoring data was drawn from the Great Duck Island [10] and Intel Research Laboratory [12] experiments. On the island 32 sensors monitored the conditions inside and outside the burrows of storm petrels measuring temperature, humidity, barometric pressure, and mid-range infrared light. The Intel group deployed 54 sensors to monitor humidity, temperature, and light in the lab. Approximately 9 million sensed values were generated on the island and over 13 million from the lab.

For tracking, data was taken from two different studies. Princeton researchers in the ZebraNet project [11] tracked Kenyan zebras generating over 62,000 sensor readings. The U.S. Air Force's N-CET [13] project tracked humans and vehicles moving through an area.

The structural health data is comprised of nearly half a million packets send by a network of 8 sensors fused to an airplane wing in a University of Colorado study [14]. Half the data was generated by a healthy wing and the other half by a wing with simulated cracking and corrosion.

Signal triangulation data came from another portion of the N-CET project, in which a network of sensors mounted on unmanned aerial vehicles intercepted and collaboratively located the sources of RF signals.

V. OUR PROPOSED APPROACH

We propose multiple versions of our TinyPack compression algorithm. First we introduce a static set of initial codes which are used as a starting point for the other methods. These codes by themselves provide good compression with excellent efficiency. Next we achieve greater compression at the cost of some RAM and processing by maintaining dynamic frequencies of the streamed values. The third approach approximates the frequencies with running statistics on the data, significantly decreasing the RAM requirements while only slightly increasing the size and processor utilization. We modify each of the above approaches by adding an all-is-well bit that gives a small boost to the compression ratio. We conclude by discussing error detection, how to adjust for real numbers instead of integers, and experimental results.

A. TinyPack initial frame static codes (TP-Init)

We begin with a set of initial codes similar to those used in LEC; however, the static codes used in LEC were optimized for jpeg compression whereas the TinyPack initial codes are designed to perform well on time-sampled sensor data with absolute minimum processing time required.

Since we are using delta compression, the data is expressed as the change in value from the previous sample. The reported values can be positive or negative. In many applications such as temperature sensing the values are cyclic so the frequency of positive changes is similar to the frequency of negative changes. In general highest frequencies appear in the smaller values (e.g. temperature usually changes fairly slowly so most changes reported are small). Also the set needs to scale to any number of values. Based on these characteristics, we construct an initial set of codes as follows:

Table IV INITIAL DEFAULT CODES

Value	+0	-1	+1	-2	+2	-3	+3
Code	1	011	010	00101	00100	00111	00110

With all other values continuing the pattern: Define B as the base of the delta value d where

$$B = \begin{cases} floor(\log_2(|d|)) & |d| > 0 \\ -1 & d = 0 \end{cases}$$

The code C is constructed as a string of $2B + 3$ bits. The first $B+1$ bits are 0s followed by the binary representation of $|d|$ (which will be B+1 bits), and a sign bit. For example, if d is 57 then B is 5. So C is constructed as 6 0 bits, followed by the binary representation of |57| (111001), followed a 0 sign bit since 57 is positive. So C is 0000001110010.

If the minimum and maximum allowed for the value are known, then the 1 bit in the center can be removed for the longest set of codes. For example, in the codes for -3 to +3 above, if the 1 bit in the center of the codes for -2,+2,-3, and +3 was removed, the leading 00 would be enough for the decoder to accurately decode those symbols. The initial static codes for values ranging from -127 to 127 are shown in Table V. The leading 1 bit in the number is considered to be part of the prefix since it is static for the entire level of the tree.

Table V DEFAULT CODES

Level	Bits	prefix	suffix range	values
0	1	1		0
1	3	01	0...1	-1.1
2	5	001	00...11	-3,-2,2,3
3	7	0001	000...111	-7,...,-4,4,...,7
4	9	00001	0000...1111	-15,...,-8,8,...,15
5	11	000001	00000...11111	-31,...,-16,16,...,31
6	13	0000001	000000...111111	-62,...,-32,32,...,63
7	14	0000000	0000000...1111111	-127,...,-64,64,...,127

Using bitwise operators the floor (round down) of log base 2 can be calculated in logarithmic time with respect to the maximum value of d using Algorithm 1. The example shows getting the base for a one byte value. The notation bxxxx is used to indicate a binary number so b10000 = 16.

Algorithm 1 FloorLog2Byte(d)

```
Objective: Calculate the base of a value
Input: Delta value d
Output: The base B of value d
    B = 0
    If d = 0
        B = -1
    Else
        d := |d|
        If d >= b10000
            rightBitShift(d, 4)
            B := B bitwiseOr b100
        End If
        If d >= b100
            rightBitShift(d, 2)
            B := B bitwiseOr b10
        End If
        If d >= b10
            B := B bitwiseOr 1
        End If
    End If
```

The value is then bit shifted to fill in the $B + 1$ prefix bits and appended to the output stream.

In order to test the validity of this initial default set, we compressed each of the datasets using only these codes. Figure 2 shows the results of the TinyPack initial codes (TP-Init) compared to the standard Deflate algorithm, S-LZW, and the LEC codes. For all the datasets our initial codes actually compressed slightly better than any of the other methods except for the N-CET Track dataset where S-LZW, LEC, and our initial codes had nearly identical performance. As expected, the Deflate algorithm, which does not specifically target sensor network data, performed significantly worse for most of the datasets. The ZebraNet and aircraft health datasets both contain significant runs of unchanging data which the Deflate algorithm takes advantage of so it performed relatively well on those datasets compared to the sensor network specific algorithms.

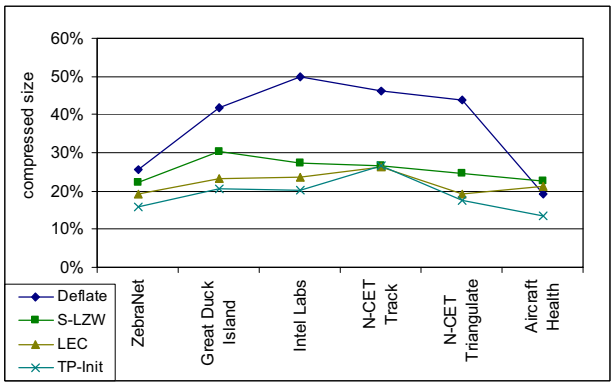

Figure 2 Initial Codes compared to Deflate, S-LZW, and LEC

B. *TinyPack with Dynamic frequencies (TP-DF)*

In order to use Huffman-style compression, the frequencies of the different data values must be known. However, in real-time systems there is often no time collect all the data to count the total frequencies of all the values before sending the currently collected data. So the frequencies from the last frame of data can be used. The frequencies are calculated both at the source and the destination to avoid the need to transmit the frequency tables. The trees and codes are updated at the beginning of each frame. Naturally, values that are in the possible range but do not appear in a frame are assigned a frequency of zero.

Since the values are typically densely clustered around 0 and sparsely scattered far from 0, the frequencies are stored in a hash table. The hash for the value is the last eight bits using 2's compliment for negative numbers so the values from -128 to 127 fit neatly into the table. The hash table is chained so that colliding values are stored in a list in the hash table bucket. This keeps the RAM requirements reasonably low while still allowing for fast lookups.

In order to capitalize on the dynamic characteristics of sensor data we add weight to the most recent values so recent occurrences have a higher impact than past occurrences but the history is not entirely forgotten. We replace the frequency table with a weighted frequency table and define a weighting factor

M such the occurrence of a new value is given twice the weight of the value observed M samples ago. So the weighted frequency $F[d]$ for a value d appearing in the n^{th} sample is updated by the following equation:

$$F[d] = F[d] + 2^{\frac{n}{M}}$$

In our experiments we set M equal to the one quarter of the frame size. At the end of a frame when the tree is updated, the weighted frequencies are normalized to reset n to 0 and prevent overflow. Also any values with a normalized frequency less than .001 are assigned a frequency of 0 and removed from the list of counted values.

So Algorithm 2 runs for each delta value in a sensed vector.

Algorithm 2 CountAndEncode(d, n, M, S, F)

```
Objective: Maintain count of frequencies and encode data
Input: Delta value d, count n, weighting factor M
    frame size S, frequency table F
Output: Frequency table updated and code appended to stream
    If Hash(d) in F
        F[d] := F[d] + 2^(n/M)
    Else
        F[d] := 2^(n/M)
    End If
    C := LookupCode(d)
    AppendToStream(C)
    n := n + 1
    If n = S //New frame
        n = 0
        For every F[x] in F
            F[x] := F[x]/(2^(S/M))
            If F[x] < .001
                F[x] := 0
            End If
        End For
        UpdateCodes(F)
    End If
```

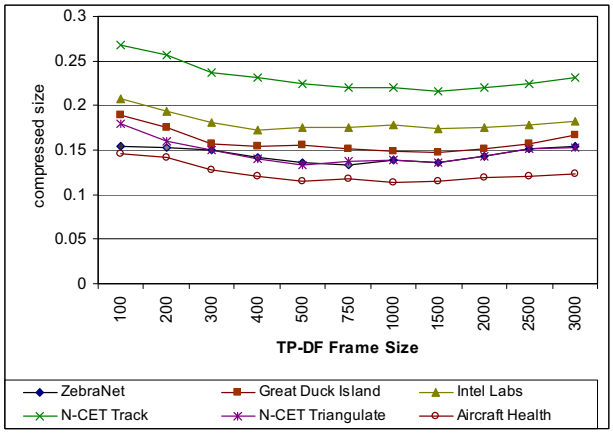

Figure 3 Frame size analysis for TinyPack with dynamic frequencies

We ran TP-DF on all the datasets with a varying frame size. Results are shown in Figure 3. When the frame size was small, the overhead for creating a new frame had a significant impact on the compressed size. When the frame size was very large, the codes were not updated frequently enough to keep up with the dynamic characteristics of the data.

Frame sizes between 500 and 1500 samples per sensor had roughly the same impact. For our experiments, we set the frame size to 512 samples.

C. *TinyPack with Running statistics (TP-RS)*

In cases where the number of possible values is very high or memory is very limited, storing the frequency table can be too costly since a standard Huffman tree on that much data would require more RAM than many sensors have available. For example, storing the frequency table for a single 4-byte integer if the values covered the entire possible range would require over 8MB of RAM while Crossbow Technology's [15] popular Mica2 and MicaZ motes have less than 1MB of total memory. In these cases the frequencies can be approximated by maintaining running statistics such as the mean and standard deviation. Because we use delta values, it is not necessary to know the distribution of the data. Only the distribution of how the data changes is important. This remains much more consistent in all of our datasets.

Beginning with the average and standard deviation that the default codes would produce the running average and standard deviation can be calculated over a window of size W. The running average $E(d)$ updates when the nth value d is sampled by the simple equation:

$$E(d)_n = \frac{1}{W} d_n + \frac{W-1}{W} E(d)_{n-1}$$

In the same way, the average of the squares of the values can be maintained. So we can compute the standard deviation σ using the well known formula:

$$\sigma = \sqrt{E(d^2) - (E(d))^2}$$

The frequency of a value occurring in a stream divided by the total number of values in the stream is referred to as the probability of that value. In a Huffman tree the probability of each leaf node is the probability of that value occurring in the stream and the probability of a non-leaf node is the sum of the probabilities of each child node. So the probability of the root is 1. The probability of each node was shown by Shannon [4] to be ideally half the probability of its parent so the level of a node in the tree should be $-\log_2(P)$ where P is the probability of the node. Using the statistics calculated the probabilities of each value can be approximated. Then the tree can simply be expressed as a table containing the number of leaf nodes that should be at each level. So the Huffman tree in Figure 1 can be compressed into Table VI where the table is stored on the sensor as an array 1-indexed on the tree level.

The code strings for the values can then be generated in logarithmic time.

Table VI COMPRESSED TREE

Level	Count
1	1
2	0
3	2
4	0
5	4
6	8

These codes are generated by creating a base code similar to a prefix for each level in the tree and using the position of each node at its level. The binary base for all nodes at a level in the tree is generated by adding the base and count of the previous level and multiplying by 2 (appending a 0) with the base for the root initialized to 0. For example, suppose the statistics approximated a tree with one node at level 1 and 1, 3, 4, and 4 nodes at levels 3, 4, 5, and 6 respectively for values of 0 to 12. The base generation for these values is shown in Table VII.

Table VII BASE GENERATION

Level	Count	Binary	Generation	Base
1	1	1	0	0
2	0	0	(0+1)*10	10
3	1	1	(10+0)*10	100
4	3	11	(100+1)*10	1010
5	4	100	(1010+11)*10	11010
6	4	100	(11010+100)*10	111100

The code for a value is generated by adding the value's position in the level to the group's base. Again, all the arithmetic is done in binary. Continuing the above example, the generation for the codes of these values is shown in Table VIII.

Table VIII CODE GENERATION

Value	Level	Position	Base	Generation	Code
0	1	0	0	0+0	0
1	3	0	100	100+0	100
2	4	0	1010	1010+0	1010
3	4	1	1010	1010+1	1011
4	4	2	1010	1010+10	1100
5	5	0	11010	11010+0	11010
6	5	1	11010	11010+1	11011
7	5	2	11010	11010+10	11100
8	5	3	11010	11010+11	11101
9	6	0	111100	111100+0	111100
10	6	1	111100	111100+1	111101
11	6	2	111100	111100+10	111110
12	6	3	111100	111100+11	111111

The probability of a level is computed as the sum of the probabilities of the nodes in the level. Since the probability of a node at level L is ideally 2^{-L}, the probability of a level is defined by:

$$P(L) = (Count(L))(2^{-L})$$

The probability of the table $P(T)$ is defined as the sum of the probabilities of all the levels. So for the table to generate accurate codes, $P(T)$ must be less than one; however, the higher it is, the more compact the code are. So the following relationship should hold (where H is the height of the tree):

$$P(T) = \sum_{L=1}^{H} (Count(L))(2^{-L}) = 1$$

Events such as changes in values are often assumed to follow exponential distributions. Experiments confirmed this in our datasets. So confidence intervals can then be used to approximate the ideal number of nodes at each depth of the tree. The values are assigned to their ideal levels rounding down so that $P(T)$ remains less than 1. Then the table is adjusted from the top down using Algorithm 3 so that nodes are pushed upward in the tree until $P(T) = 1$.

Algorithm 3 FilterUp(T, H)

Objective: Produce optimal codes by getting $P(T) = 1$
Input: Table T where T is simply the array of the counts
    Height of tree H
Output: T adjusted so that $P(T) = 1$
    $P(T) := 0$
    For L From 1 to H
        $P(T) := P(T) + T[L]*2^{(-L)}$
    End For
    For L From 1 to H-1
        //Get the highest number that can possibly move
        move_count := Floor((1- $P(T)$)/(2^(-L-1)))
        //Don't move more than are there
        move_count := Max(move_count, $T[L]$)
        //If move_count is 0 the next two lines do nothing
        $T[L] := T[L]$ + move_count
        $T[L+1] := T[L+1]$ – move_count
    End For

The window size analysis for the running statistics was almost identical to the frame size results using dynamic frequencies (shown in Figure 3). So again the experiments were run with a window size of 512.

Figure 4 shows the results of running both the dynamic frequencies (TP-DF) and running statistics (TP-RS) over the datasets compared to the other methods.

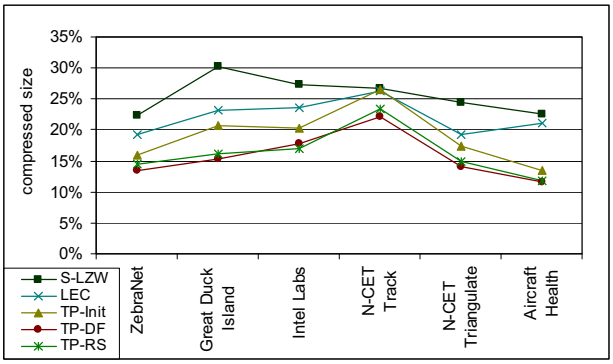

Figure 4 TinyPack with dynamic frequencies and running statistics

The running statistics generally performed slightly poorer than dynamic frequencies except on the Intel Labs dataset. The

data in this set is more precise and follows a cleaner statistical pattern than the others.

D. All-is-well bit

Most sensor applications send a vector of values (e.g., timestamp, temperature, humidity) at each sampling interval. Often in the data sets studied all the values in a sample were exactly equal to the previous corresponding value. Similar to the methods in [18], a bit can be appended to the beginning of the packet indicating whether or not this has occurred (obviously if it has, no more data needs to be sent for that packet). In protocols with variable sized packets or packets that are small compared to the size of a vector of readings, this could introduce additional savings.

The datasets were affected differently by adding this. Figure 5 shows the effects of the all-is-well bit (AIW). TP-DF and TP-RS were very similar, so TP-RS was removed to avoid cluttering the graph. In each of the TinyPack algorithms the all-is-well bit improved performance for all the datasets except the aircraft health and N-CET tracking sets. This is due to the higher level of precision in those datasets. The datasets had a very small number of packets where all the values were identical to the previous packet. In general, if the application is designed such that sensed values will rarely be exactly equal to the previous value (as in high precision data), the all-is-well bit should not be used.

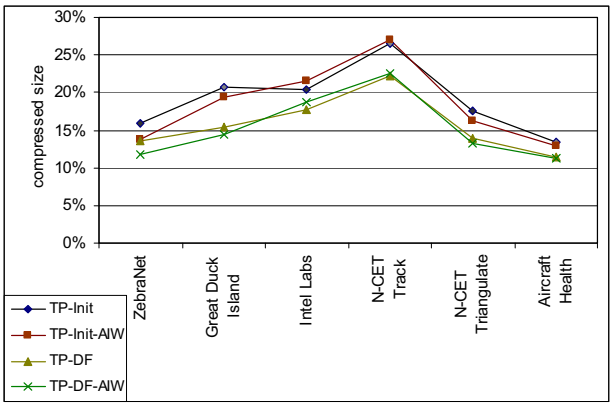

Figure 5 Effects of all-is-well bit

Additionally, if the sensors send on a predetermined schedule or if the packet headers contain consecutive sequence numbers, simply refraining from sending data could be used to indicate the same thing as the all-is-well bit. This would remove the overhead so no decision would need to be made whether or not to use it. These intentionally unsent packets would be easily differentiated from actual drops based on the sequence numbers or the error detection discussed in the next section.

E. Error detection

The first packet in a new frame is sent with uncompressed values. Each additional packet is sent using the delta (change) values. If the last value is repeated in the first packet of the next frame, the values can be compared to check for the presence of errors due to dropped packets or corrupted values in the packets.

For example, suppose a temperature sensor sensed values at 23, 25, 28, and 29 with a frame size of 4. The first frame contains [23, +2, +3, and +1]. Assuming packet corruption changed the +3 to -3, the receiver would read the values as 23, 25, 22, and 23. When the second frame was sent with 29 as the first value the receiver could see that an error had occurred since the last value (23) does not equal the first value of the next frame (29).

This successfully detects all single bit errors and single dropped packets; however, it is possible that multiple errors could cause the values of the compared packets to actually be equal although the errors existed. For example a +2 and a -2 could both be dropped. In this case the drops would be undetected.

Since the codes are dynamic, the chances of undetected error constantly changes but the codes in all cases were consistently distributed similarly to the static default codes so those were used for error analysis.

Assuming the values occur with the probability expected by the default codes, the probability of a bit error occurring in the base (prefix) of a code can be determined by calculating the expected number of prefix and suffix bits in a code.

From Table V it can be seen that a code at level L has a prefix length L+1 and suffix length L. The count of nodes at that level is 2^L so the probability of a random sampled value being on that level is $2^{-(L+1)}$. Therefore the expected number of prefix bits $E(P)$ for an arbitrarily large set of possible values is:

$$E(P)=\sum_{L=0}^{\infty}\left(\frac{L+1}{2^{L+1}}\right)=\frac{1}{2}+\frac{2}{4}+\frac{3}{8}+\frac{4}{16}+\ldots$$

$$2E(P)-E(P)=2$$

Similarly, the expected number of suffix bits $E(S)$ is:

$$E(S)=\sum_{L=0}^{\infty}\left(\frac{L}{2^{L+1}}\right)=\sum_{L=0}^{\infty}\left(\frac{L+1}{2^{L+1}}-\frac{1}{2^{L+1}}\right)$$

$$=E(P)-\sum_{L=0}^{\infty}\left(\frac{1}{2^{L+1}}\right)=1$$

So as the height of the tree approaches infinity, $E(P)$ approaches 2 and $E(S)$ approaches 1. So the probability of a bit errors occurring in the prefix for large trees approaches 66.67%. Calculating for the case where the values can range from -127 to 127 gives 66.98%. Such errors would change the expected length of the code and would be detected at the end of the packet transmission.

For bit errors in the suffix of a code and for drops the probability of a subsequent error "correcting" the value and causing the errors to be undetected is roughly 3.57%. This was calculated by an extensive state transition diagram and a transition matrix which were excluded due to space constraints. Since most sensors send a vector of values at each sample the

probability of detecting multiple errors from dropped packets is $(.0357)^{|V|}$ where $|V|$ is the vector size of the sample.

For example, the Intel Labs dataset contains 2.3 million samples with six values in each sample so $|V| = 6$. In the worst case there will be exactly two drops per frame. So assuming 10% packet loss, there would be approximately 115,000 frames each containing two dropped packets. The chance of detecting every drop would be

$$\left(1-(.0357)^6\right)^{115000} \approx 99.976\%$$

The worst case probabilities are shown for each of the datasets in Table IX.

Table IX PROBABILITY OF DROP DETECTION

Dataset	|V|	frames	probability
ZebraNet	6	284	99.9999%
Great Duck Island	8	38226	>99.9999%
Intel Labs	6	115123	99.9762%
N-CET Track	4	23143	96.3106%
N-CET Triangulate	6	11123	99.9977%
Aircraft Health	2	22937	<0.00001%

Experiments were conducted with errors generated assuming Poisson inter-arrival times and results were consistent with the above analysis.

The aircraft health data has only two values per vector and so in the worst case, at 10% drop rate, errors would undoubtedly go undetected. For such datasets, it would be effective to define a smaller frame size to reduce the probability of multiple errors occurring in the same frame or to send error detection packets in the middle of the frame instead of always sending them at the end.

F. *Working with real values*

TinyPack works most effectively with integers. Our approach could fairly intuitively be extended into the real numbers; however, for simplicity in our experiments, we expressed reals as integers. In the case where the real values were rounded in the dataset to some low number of decimal places, we simply shifted the decimal point. In the case of higher precision reals, we split the values into the exponent and mantissa and compressed them separately.

VI. EXPERIMENTAL RESULTS

Experiments were performed using TOSSIM [16], which simulates the open source TinyOS operating system that runs on many sensors. TOSSIM simulated Crossbow Technology's MicaZ motes [15] and was used to test performance of compression as well as accuracy, RAM usage, and processor utilization. In addition to TOSSIM the PowerTOSSIM [17] simulator was used. PowerTOSSIM is built on top of TOSSIM and is capable of also measuring simulated energy consumption and latency.

A. *Compression*

To summarize, we calculate the entire compression of all the data across every dataset. Figure 6 shows the compressed size of all the data using the standard Deflate algorithm used in most operating systems, S-LZW, LEC, and our approaches: The static initial codes (TP-Init), dynamic frequencies (TP-DF), running statistics (TP-RS), and each of the TinyPack methods with the all-is-well bit added (-AIW).

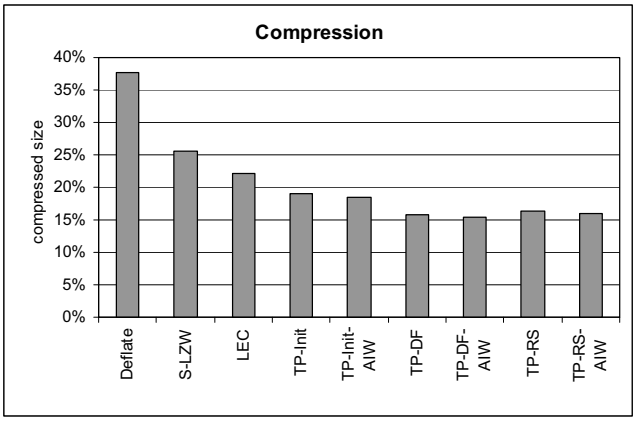

Figure 6 Compression summary

B. *Accuracy*

Since the TinyPack algorithms produce approximations of the frequencies of the values, a measure of accuracy can be calculated by comparing the lengths of the generated codes for each frame to the optimal code lengths determined by generating standard Huffman codes. Figure 7 shows the performance of the TinyPack algorithms compared to the performance of a theoretical optimal algorithm. It should be noted that while standard Huffman coding would produce optimal codes, the overhead for sending the new tree at every frame would cause the algorithm to perform much worse than any of the others. No algorithm currently exists which produces optimal codes with no overhead.

The data in both Intel Labs and aircraft health remains fairly consistent throughout the entire dataset so the approximated codes almost reached the optimal level.

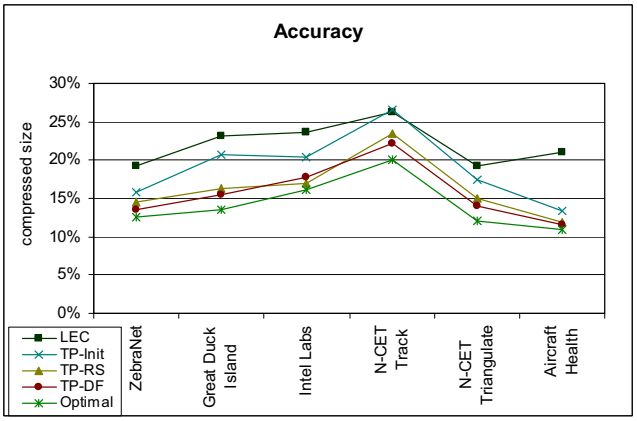

Figure 7 Accuracy

C. *Latency*

Sending the uncompressed data takes less time in processing but more time in transmission so the latency

depends on the motes used. In general, however, processor speed is exponentially faster than radio data rate for wireless sensors (for example, the MicaZ mote [15] has a 7 MHz processor and a 250 kbps high data rate radio). So for the MicaZ motes latency is decreased proportionally to the compressed size of the data. So TinyPack has a decrease in latency of 80-85% compared to uncompressed data.

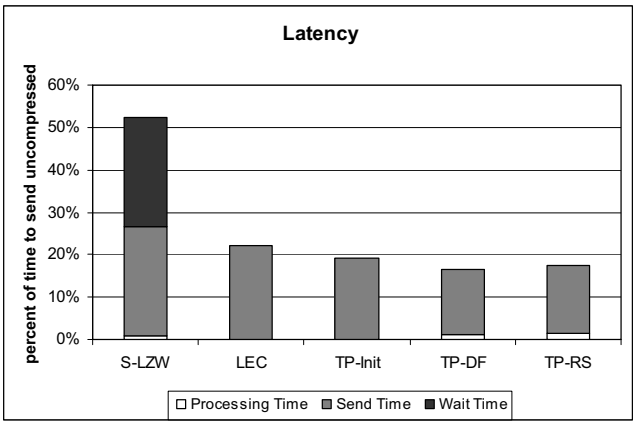

Figure 8 Latency

For comparison, the S-LZW algorithm was modified to send data as soon as possible and it was assumed packets were sent in a constant stream. Figure 8 shows the relative latencies scaled to the uncompressed data. In each version of TinyPack adding the all is well bit decreased the latency by less than half a percent and so data for the all-is-well bit is not shown separately. Deflate is not shown since it requires collecting all of the data prior to compressing.

D. Energy

Energy consumed for compressing, writing to memory, and transmitting was measured using PowerTOSSIM. Results are shown in Figure 9. Results are again scaled to uncompressed and averaged over the datasets. As with latency, the all-is-well bit in each case decreased the energy usage by less than half a percent. Deflate was used only as a compression benchmark and was not implemented in PowerTOSSIM so energy usage data was not collected for the Deflate algorithm.

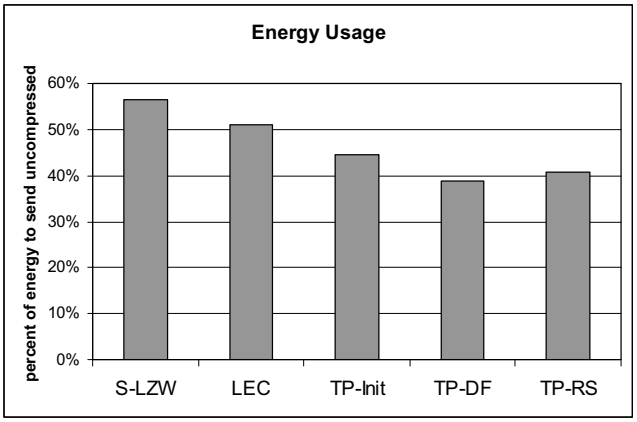

Figure 9 Energy Usage

E. RAM

The maximum amount of RAM utilized by each algorithm for each dataset is shown in Figure 10. S-LZW is designed to work on any generic dataset and uses the same compressor for every value in a sensed vector so the RAM usage was constant for S-LZW. As expected, TP-DF had the highest RAM usage because it stores the frequency tables; however, the RAM was still well within the limits of the Mica2, MicaZ, and most other sensors. LEC and TP-Init both use very little RAM since the codes are static and generated at runtime for each value.

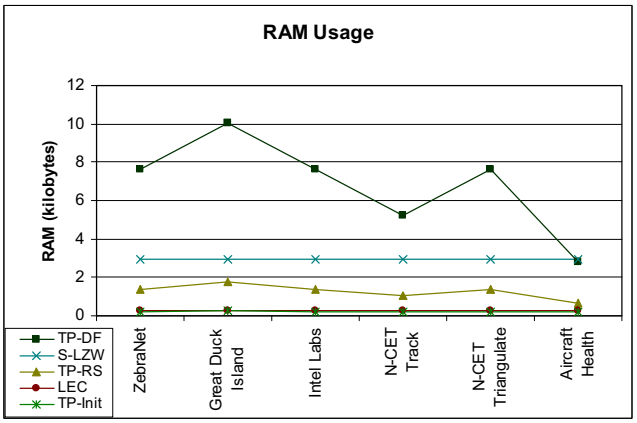

Figure 10 RAM Usage

F. Processor Utilization

In order to measure processor utilization, the program counters on each sensor were accessed at the start and end of each simulation. For these simulations, the data was compressed and not transmitted so that the processor utilization would not be affected by the compression ratio. Figure 11 shows the instruction count for each algorithm scaled to show the average instruction count per byte of uncompressed data. As with RAM, the static codes used in LEC and TP-Init cause the processor utilization to be very low. TP-DF and TP-RS required significantly higher processor time than the other algorithms; however, due to the nature of the sensor hardware, the savings in energy and latency from the reduced data size far outweigh the costs of higher processor utilization. The energy usage in Figure 9 includes energy spent processing.

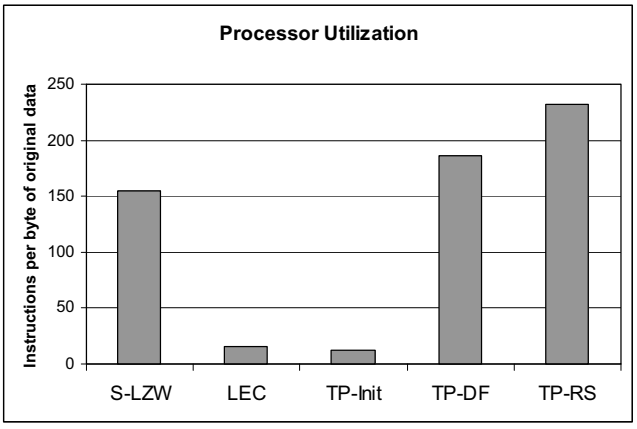

Figure 11 Processor Utilization

VII. Conclusions and Future Work

TinyPack effectively compresses data while not introducing delays and even reduces latency compared to sending uncompressed data. TinyPack is effective on all sensor networks which use time-based sampling and is especially effective on systems with high granularity or low local variance.

TP-Init required the least RAM and by far the least processing time of all the TinyPack algorithms but resulted in the poorest compression. TP-DF achieved the greatest compression ratios, but required more RAM than the other methods. TP-RS compressed almost as well and required much less RAM. So while TP-DF compressed most effectively, systems with low RAM would benefit from using TP-RS and systems with very low RAM or high cost for processor utilization could use TP-Init for best results.

While the focus of this paper has been lossless compression, TinyPack could be modified to continue sending change values of zero until the change exceeded some threshold. Additionally, packets could be dropped to indicate no change had occurred. In systems which could tolerate some rounding error or lossiness, this could dramatically increase the compression with a small degree of error.

In many applications sensors are not only temporally located but also spatially located (sensors sense data similar to that of a nearby sensor). It could prove effective to express the delta values as the change from the value of a nearby sensor instead of the change from previous value or some hybrid of the two.

References

[1] D. A. Huffman. “A Method for the Construction of Minimum-Redundancy Codes.” In Proceedings of the I. R. E., 1952.

[2] J. S. Vitter, “Design and Analysis of Dynamic Huffman Codes”, Journal of the ACM, 34(4), October 1987, pp 825–845

[3] J. Ziv and A. Lempel. “A Universal Algorithm for Sequential Data Compression. IEEE Transactions on Information Theory,” 23(3):337–343, 1977.

[4] C.E. Shannon, “A Mathematical Theory of Communication”, Bell System Technical Journal, vol. 27, pp. 379–423, 623-656, October, 1948.

[5] T. Arici, B. Gedik, Y. Altunbasak, and L. Liu, “PINCO: a Pipelined In-Network Compression Scheme for Data Collection in Wireless Sensor Networks,” In Proceedings of 12th International Conference on Computer Communications and Networks, October 2003.

[6] D. Petrovic, R. C. Shah, K. Ramchandran, J. Rabaey, “Data Funneling: Routing with Aggregation and Compression for Wireless Sensor Networks,” In Proceedings of First IEEE International Workshop on Sensor Network Protocols and Applications, May 2003.

[7] Sadler C. and Martonosi M. “Data Compression Algorithms for Energy-Constrained Devices in Delay Tolerant Networks,” In Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys), 2006.

[8] F. Marcelloni and M. Vecchio, “An Efficient Lossless Compression Algorithm for Tiny Nodes of Monitoring Wireless Sensor Networks,” Computer Journal, vol. 52, no. 8, pp. 969–987, 2009.

[9] S. Gandhi, S. Nath, S. Suri, and J. Liu. “GAMPS: Compressing Multi Sensor Data by Grouping and Amplitude Scaling,” In Proceedings of the 35th SIGMOD international Conference on Management of Data, New York, NY, 771-784. 2009.

[10] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson, “Wireless sensor networks for habitat monitoring,” In WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications. New York, NY, USA: ACM, 2002, pp. 88-97.

[11] P. Zhang, C. M. Sadler, S. A. Lyon, and M. Martonosi. “Hardware Design Experiences in ZebraNet.” In Proc. of the ACM Conf. on Embedded Networked Sensor Systems (SenSys), 2004.

[12] P. Bodik, W. Hong, C. Guestrin, S. Madden, M. Paskin, and R. Thibaux. Intel Berkley Labs. 2004

[13] J. M. Metzler, M. H. Linderman, and L. M. Seversky, “N-CET: Network-Centric Exploitation and Tracking,” in MILCOM 2009 - 2009 IEEE Military Communications Conference. IEEE, October 2009.

[14] Zhao, Xiaoliang, Qian, Tao, Mei, Gang, Kwan, Chiman, Zane, Regan, Walsh, Christi, Paing, Thurein, Popovic, and Zoya, “Active health monitoring of an aircraft wing with an embedded piezoelectric sensor/actuator network: II. wireless approaches,” Smart Materials and Structures, vol. 16, no. 4, pp. 1218-1225, August 2007.

[15] Crossbow Technology, Inc. MicaZ Datasheet. http://www.xbow.com/, 2010.

[16] P. Levis, N. Lee, M. Welsh, and D. Culler. “TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In Proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys) 2003.

[17] V. Shnayder, M. Hempstead, B. Chen, G. W. Allen, and M. Welsh, “Simulating the Power Consumption of Large-Scale Sensor Network Applications,” In Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys), 2004.

[18] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. "TAG: a Tiny Aggregation Service for Ad-Hoc Sensor Networks." In Proceedings of the Fifth Symposium on Operating Systems Design and implementation (OSDI ’02), 2002.