Summer 2014

Energy efficient and latency aware adaptive compression in wireless sensor networks

Thomas Mark Daniel Szalapski

Department: Computer Science

ENERGY EFFICIENT AND LATENCY AWARE ADAPTIVE COMPRESSION IN WIRELESS SENSOR NETWORKS

by

THOMAS MARK DANIEL SZALAPSKI

A DISSERTATION

Presented to the Faculty of the Graduate School of the

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

In Partial Fulfillment of the Requirements for the Degree

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

2014

Approved by:
Sanjay Madria, Advisor
Bruce M McMillin
Jagannathan Sarangapani
Wei Jiang
Sriram Chellappan

PUBLICATION DISSERTATION OPTION

This comprehensive report consists of five articles published or submitted for publication in peer reviewed journals or conference proceedings:

Pages 2 to 31, "Energy-Efficient Real-Time Data Compression in Wireless Sensor Networks" was published in the IEEE International Conference on Mobile and Data Management (MDM 2011), Luleå, Sweden, and was awarded the best paper.

Pages 32 to 50, "Tinypack Xml: Real Time Xml Compression for Wireless Sensor Networks" was published in the IEEE International Conference on Wireless Communications and Networking (WCNC 2012), Paris, France.

Pages 51 to 93, "On Compressing Data in Wireless Sensor Networks for Energy Efficiency and Real Time Delivery" was published in the 31st Volume of the Distributed and Parallel Databases Journal (DPDS 2013) 31(2): 151-182 (2013)., Springer.

Pages 94 to 121, "Energy Efficient Distributed Grouping and Scaling for Real-Time Data Compression in Sensor Networks" was submitted for publication to the IEEE International Conference on Big Data 2014 (IEEE BigData 2014). Washington DC.

Pages 122 to 151, "Toward Energy Efficient Multistream Collaborative Compression in Wireless Sensor Networks" was submitted for publication to the 10th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing, (CollaborateCOM 2014), Miami, Florida.

ABSTRACT

Wireless sensor networks are composed of a few to several thousand sensors deployed over an area or on specific objects to sense data and report that data back to a sink either directly or through a series of hops across other sensor nodes. There are many applications for wireless sensor networks including environment monitoring, wildlife tracking, security, structural heath monitoring, troop tracking, and many others. The sensors communicate wirelessly and are typically very small in size and powered by batteries. Wireless sensor networks are thus often constrained in bandwidth, processor speed, and power. Also, many wireless sensor network applications have a very low tolerance for latency and need to transmit the data in real time. Data compression is a useful tool for minimizing the bandwidth and power required to transmit data from the sensor nodes to the sink; however, compression algorithms often add a significant amount of latency or require a great deal of additional processing. The following papers define and analyze multiple approaches for achieving effective compression while reducing latency and power consumption far below what would be required to process and transmit the data uncompressed. The algorithms target many different types of sensor applications from lossless compression on a single sensor to error tolerant, collaborative compression across an entire network of sensors to compression of XML data on sensors. Extensive analysis over many different real-life data sets and comparison of several existing compression methods show significant contribution to efficient wireless sensor communication.

ACKNOWLEDGMENTS

One thousand thanks to Dr. Sanjay Madria for all his training and assistance with research, for securing grants and equipment, and especially for his gracious accommodation of my difficult schedule.

Thanks also to Drs. Bruce McMillin, Jagannathan Sarangapani, Wei Jiang, and Sriram Chellappan for serving on my advising committee.

Finally, thanks to my wife, Jennifer, for all her support and encouragement throughout my entire graduate education.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

SECTION

1. INTRODUCTION

Wireless sensors are used for a great host of different applications such as environment monitoring, health care, security, military, structural health, social behavior analysis, and vehicular networks. Wireless sensor networks are well known to be much more constrained than traditional computers. There can be thousands of wireless sensors in the same network all communicating with relatively low speed radios making bandwidth very limited. Most wireless sensors are powered by batteries. Changing the batteries in a sensor can be difficult, expensive, or even dangerous (especially in military uses) so the power consumption is a critical aspect of many wireless sensor deployments. Many wireless sensor networks also have a need for real time delivery of data; thus, minimizing latency is important.

Effective data compression is therefore imperative to an efficient deployment of a wireless sensor network. This document presents several compression algorithms targeting a wide variety of use cases for sensor networks. The algorithms are designed to be effective and simple to implement. Extensive analysis and experimentation show excellent results when compared to the state of the art research in the field.

PAPER

I. ENERGY-EFFICIENT REAL-TIME DATA COMPRESSION IN WIRELESS SENSOR NETWORKS

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.

1. 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.

This research is supported by DOE grant number P200A070359.

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

2. BACKGROUND

2.1. 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 12 generate the codes using the Huffman tree in Figure 19. Huffman codes were shown to be optimal for symbol by symbol compression in [1].

Table 1 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

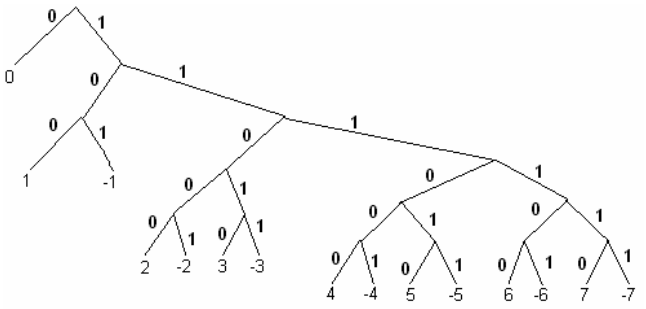

Figure 1 Huffman tree

2.2. 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.

2.3. 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.

3. RELATED WORK

3.1. 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 2 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 13 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.

3.2. 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 14 with anything past level 7 following the pattern of the last three levels.

Table 3 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

3.3. 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.

3.4. 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.

4. 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.

5. 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.

5.1. 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 4 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 2*B* + 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 5. 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 5 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.

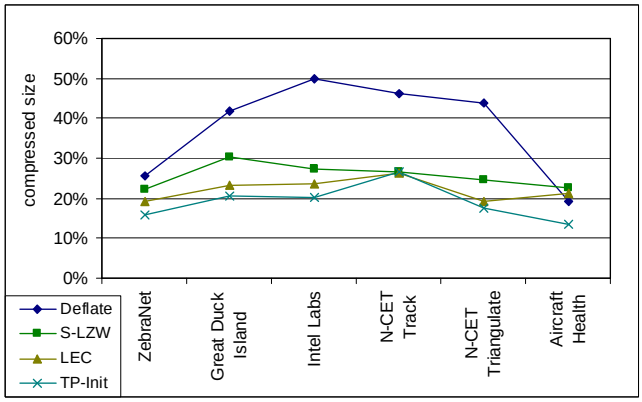

Figure 2 Initial codes compared to deflate, S-LZW, and LEC

5.2. 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:

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

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
```

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.

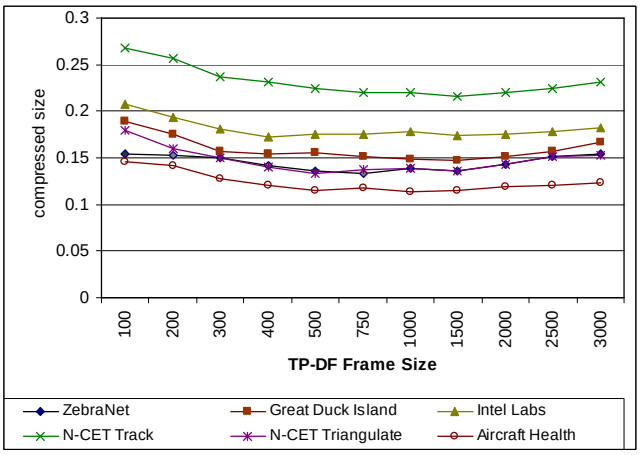

Figure 3 Frame size analysis for tinypack with dynamic frequencies

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.

5.3. 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 *n*th 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 19 can be compressed into Table 6 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 6 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 7.

Table 7 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 8.

Table 8 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.

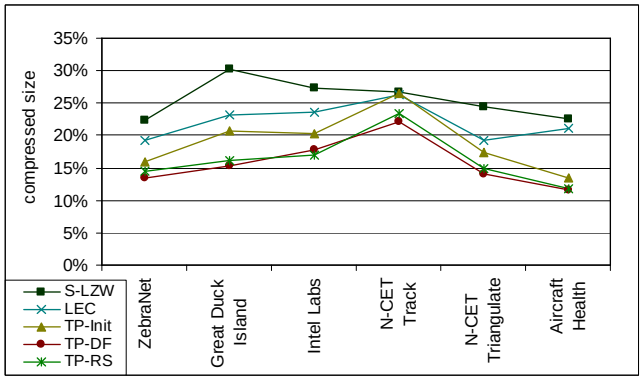

Figure 4 Tinypack with dynamic frequencies and running statistics

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.

5.4. 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 [19], 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.

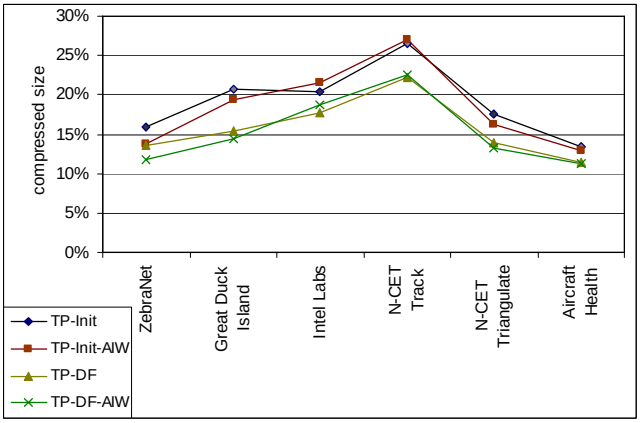

Figure 5 Effects of all-is-well bit

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.

5.5. 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 18 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:

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

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

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 9.

Table 9 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.

5.6. 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.

6. EXPERIMENTAL RESULTS

Experiments were performed using TOSSIM [17], 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 [18] simulator was used. PowerTOSSIM is built on top of TOSSIM and is capable of also measuring simulated energy consumption and latency.

6.1. 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).

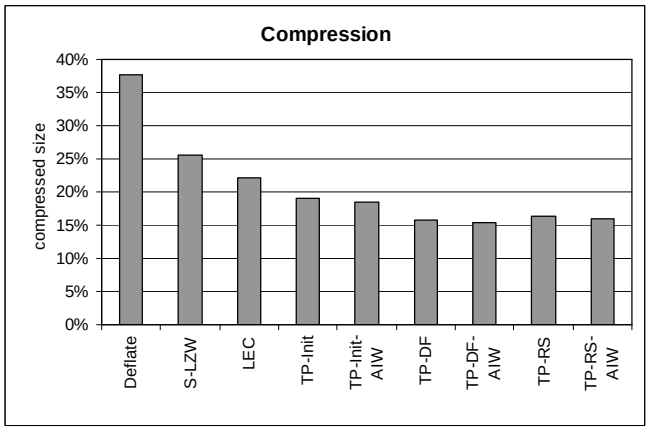

Figure 6 Compression summary

6.2. 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.

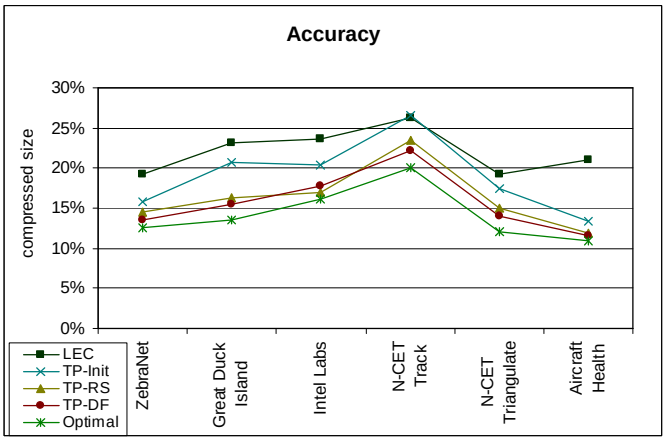

Figure 7 Accuracy

6.3. 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.

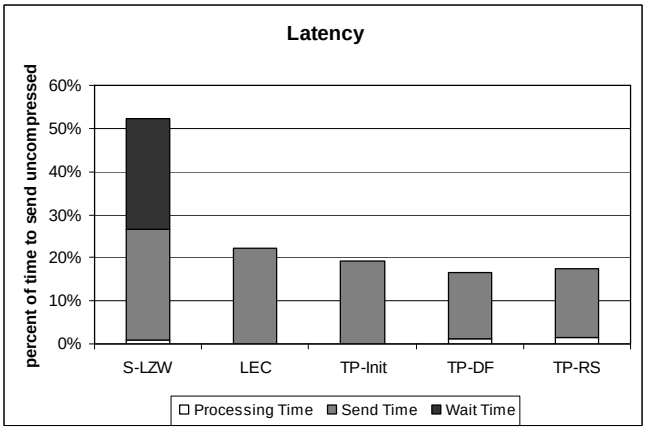

Figure 8 Latency

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.

6.4. 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.

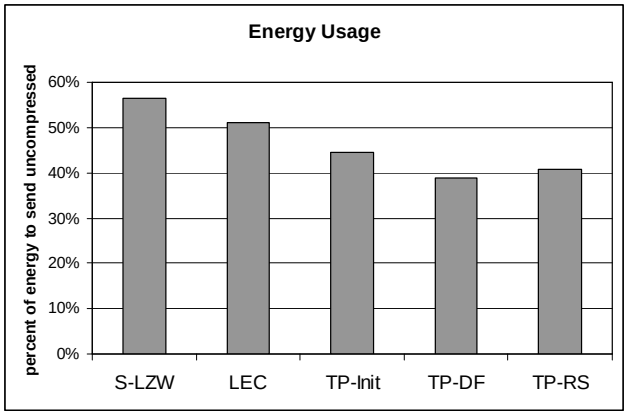

Figure 9 Energy usage

6.5. 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.

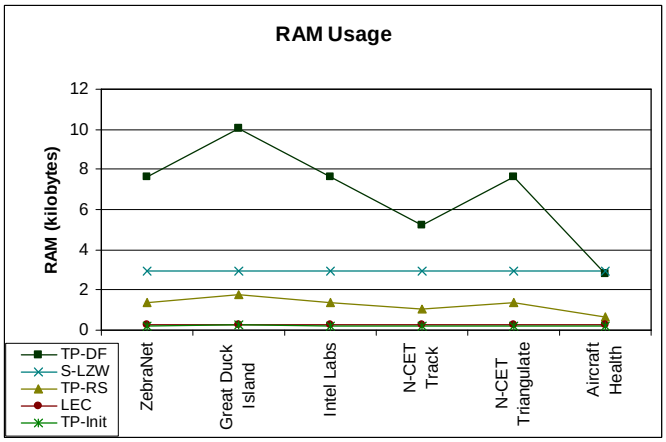

Figure 10 Ram usage

6.6. 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 35 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 11 includes energy spent processing.

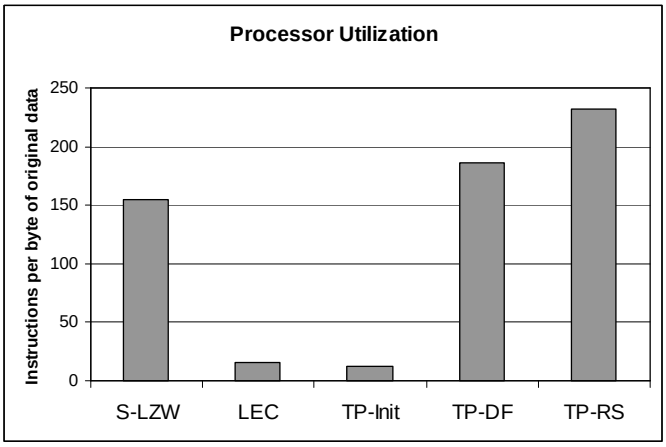

Figure 11 Processor utilization

7. 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.

II. TINYPACK XML: REAL TIME XML COMPRESSION FOR WIRELESS SENSOR NETWORKS

Wireless networks possess significant limitations in bandwidth. Additionally, real-time networks cannot tolerate high latency. While some good XML compression algorithms exist, there remains a need for methods that reduce latency and bandwidth usage further in real time wireless applications. This paper presents a new compression scheme which reduces bandwidth while minimizing latency of XML data while in transit. XML structural data is reduced to format strings and arguments are sent as they are generated using modifications of real-time compression techniques specific to each data type. Methods are introduced to gracefully handle lost data in environments where delivery of all packets is not guaranteed. Performance evaluations show increased compression ratios and a decrease in latency and energy for our method compared to existing XML data compression approaches.

1. INTRODUCTION

XML is designed to be a universal format for storing and transmitting data. XML it is inherently redundant and requires an inflated amount of memory to store and bandwidth to transmit. Also, many of these applications are used in wireless environments which generally have relatively low bandwidth capabilities. Although other more compact formats have been proposed, XML remains heavily used in both old and new applications. Efficient data compression should clearly be considered for these applications. Many compression algorithms have been designed which are specific to XML data [23][24]. Unfortunately, most only work well if all of the XML data is collected prior to compression which is not possible in many data streaming applications.

The U.S. Air Force uses XML for many real-time applications. These are characterized by an extremely low tolerance for latency. For example: if a collection of unmanned aerial vehicles (UAVs) are being used to track a ground object, each UAV must communicate the current location and movement vector of the object as soon as possible or it may be too far away before another UAV knows to look for it. So there exists a need for a fast, efficient, XML compression scheme which relies only on current and previous data. The N-CET project [22] incorporates several of these real-time, wireless, XML applications and was the primary motivation and source of data for this work. This project is explained further in Section 8 where the datasets are discussed.

We propose TinyPack XML, a novel compression method which capitalizes on the redundancy in XML structure and the similarity between XML packets sent by wireless devices. TinyPack XML compresses each packet as it is created without any need for delay. TinyPack XML compresses using format strings. The portions of the XML structure which are common to many packets are generated on the fly or *a priori* and the values which vary from packet to packet are compressed using techniques specific to the type of data being sent. Some pre-existing methods are used and others are modified to better fit the specific characteristics of the wireless networks. We consider correlated and uncorrelated numeric data and short and long text strings. In every experiment, the compressed data actually arrived faster than uncompressed since data transmission was more expensive than processing. We compare TinyPack XML to several existing XML compressors using metrics such as latency, RAM, and compression ratio. Experiments show that it achieves compression ratios comparable to and better than that of related methods which require all the data to compress.

2. EXISTING COMPRESSORS FOR XML DATA

2.1. DEFLATION

The deflation algorithm is a used in many common compression programs (including gzip and WinZip) and is often used as a comparison for compression algorithms since it performs fairly well on most types of data and is widely used.

2.2. XMILL

XMill [23] compresses XML data by separating it into three components: The element and attribute names, the text values, and the tree structure of the XML document. The text values are grouped by parent element name and the three components are then compressed using standard text compression techniques.

2.3. XMLPPM

XMLPPM [24] uses a similar restructuring as XMill but uses predictive arithmetic coding to compress the transformed data. Each symbol (character or string of characters) has a certain probability of appearing after every other symbol. These probabilities are calculated and arithmetic encoding is used to store each symbol.

2.4. WBXML

WBXML [25] is a binary XML format maintained by the Open Mobile Alliance used on many mobile phones. It converts all the pieces of XML into binary tokens and preserves the structure of the XML document.

2.5. XAUST

XAUST [26] generates a model for the compression and decompression of XML documents based on the schema. It then uses the automatically generated model along with arithmetic compression techniques to compress the document.

2.6. PAQ

PAQ [27] is a constantly evolving compression suite which generally produces the best compression ratios for most types of data. It achieves this by using enormous amounts of RAM and requiring much more time than other methods. PAQ can be configured to consume between 233 and 1712 MB of RAM. It is entirely impractical for real-time wireless systems and is included as an ideal lower bound for compressed size.

3. OUR APPROACH

While XML is defined as being only semi-structured, the data from most wireless applications including N-CET tend to be highly structured. Subsequent packets often had identical or nearly identical XML tree structures. We also examined several common benchmark XML datasets (which could be intuitively broken into packets) and found that most also exhibited this structural similarity between packets.

We generate format strings (similar to the well known printf function in the C programming language) for each type of packet. The format string expresses the structure of the XML data in the packet and the portions which differ from packet to packet (arguments) become all that must be transmitted for subsequent packets. For example, assume a target tracking application generated the following two data packets for a target’s location at separate times:

<target><lat>45</lat><lon>50</lon></target>

<target><lat>43</lat><lon>55</lon></target>

The format string could be expressed as <target><lat>[arg1]</lat><lon>[arg2]</lon></target> and the wireless device could just send the arguments [45, 50] and [43, 55] after the format string was established.

We use standard text compression to compress the format strings and various compression schemes for the arguments specific to the type of data they contain. These are detailed in the following section.

4. ARGUMENT COMPRESSION

4.1. CORRELATED NUMERIC DATA

For arguments containing numeric data where the numbers tended to be correlated between successive packets (such as location information, timestamp, size of tracked object in window, etc) the values were expressed as the change from the previous value and encoded using TinyPack compression with Running Statistics [28]. Smaller change values are assigned shorter bit strings based on the current mean and variance of the data. Change values are initially encoded based on Table 10 and then modified as the running average and standard deviation change.

Table 10 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

4.2. UNCORRELATED NUMERIC DATA

Uncorrelated numeric arguments (such as target ID) were converted to appropriately sized integer types and sent using the number of bits required to send the maximum possible value for that argument. So, for example, if a value could range from 0 to 1000, it would be sent with 10 bits per packet.

4.3. LONG TEXT STRINGS

Arguments which contained long or unstructured text strings (such as comments) were compressed using regular SLZW compression [29]. The dictionary begins with the common alphanumeric characters and punctuation. Then common subsequences of characters or uncommon characters are added to the dictionary as they are encountered. The system was designed to support pre-loading the dictionary with application specific symbols or by building the initial dictionary based on sample data.

4.4. SHORT AND SINGLE-WORD TEXT STRINGS

For arguments where the strings were comprised of a small subset of words (such as status and target name) each possible value was indexed. The dictionary could be preloaded or built on the fly using the last index position to indicate a new entry. New entries were compressed in the same manner as long strings and the index positions were sent with the minimum number of bits required. This is shown in Algorithm 21. So if the dictionary had seven entries, only three bits would be required. Note that if the dictionary had eight entries, four bits would be needed to allow for the new entry symbol to be encoded.

Algorithm 1 CompressShort(str, dict)

Objective: Compress short strings
Input: String *str*, current dictionary *dict*
Output: Encoded index value and updated dictionary

```
//The +1 is for the new entry symbol
bits = floor(log2(count of items in dict + 1))
If str is in dict
    code = index of dict padded with 0s to length bits
    Add code to output stream
Else
    code = index count of items in dict padded with 0s
    Update dict by adding str to the end
    Add code to output stream
End If
```

5. FORMAT STRINGS

5.1. STRUCTURE

Format strings are simply the element structure of the XML packet with the escape characters shown in Table 11.

In practice, the escape characters are actually single characters and are themselves compressed during the compression of the format strings discussed previously. The length and index parameters are expressed by a single character with the integer encoded as the dictionary index position of the character. For example, an integer with a fixed length of 4 would be encoded as the fixed length integer escape character followed by the fourth character in the dictionary.

Recall the sample XML packets from the previous example:

<target><lat>45</lat><lon>50</lon></target>

<target><lat>43</lat><lon>55</lon></target>

So the actual format string generated would be: <target><lat>\I\E<lon>\I\E\E.

Table 11 Escape characters

Character	Description
\I	Integer argument
\F[x]	Fixed length integer argument. Padded with 0s. x is the length.
\D	Decimal (floating point) argument.
\T	Text (long string) argument
\L	List (short and single-word string) argument
\?	Optional. Following portion may or may not appear (encode 0 or 1 in compressed stream).
*	Multi. Following portion can be repeated (encode number of repetitions).
\{ and \}	Open and close bracket. Enclose portions of string for optional and multi.
\P[x]	Previous. Argument is equal to previous argument at index x (need not encode).
\E	End tag. Serves to help compress format string.

5.2. GENERATION

We developed four different ways for the format strings to be generated. Each has its positive and negative sides and the decision for which to use is left up to the user.

First, the format string can be generated on the fly. The parser assumes that all non-structural data is arguments in the initial packet and adds optional and multi characters as the need arises. Also, arguments which never change (after a threshold) are moved from the argument list into the format string. This method requires no additional input from the user but has additional overhead since the format string must be transmitted and will often need to be modified.

The tags in the first packet are initially assumed to be part of the static structure of the format string and all the attributes and element values are assumed variable and are set up as arguments. The type of each attribute and element is inferred by the characters and length. As additional packets are sent, portions of the structure can be flagged as optional and other optional pieces can be added. If any attribute or element remains unchanged, it

is added to the structure of the format string and any changes in type are made as needed. The format string update messages are described in the next subsection.

Next, sample data could be used instead. This works similarly to the first method but removes the overhead for transmitting format strings during runtime and still doesn't require much of the user. Of course this is only useful if good representative sample data is available.

Third, the format strings can be automatically generated by the XML schema. This ensures that the string should never need to be updated and also requires little from the user. This works well if the XML schema is carefully defined; however, in the datasets we studied this frequently created unused arguments and unnecessarily long format strings since the schemas often allowed for much more than was actually used.

Finally, the user can simply write the format strings manually for each type of packet. If written well, this will be optimal and allow for the highest compressibility; however this would require more training than many users may want to do. We created a parser to check the validity of user-written format strings and to test them against sample data.

5.3. UPDATES

If the format string is built on the fly or if it is built *a priori* and the data changes in some significant way or if it was built incorrectly, then it needs to be able to be modified in real time.

Special format string modification packets can be sent through the network to alert the receiver of the necessary changes. These packets are marked as high priority and should never be dropped.

The modification could consist of any number of delete, insert, and replace messages. The replace messages contain an index and length for which portion of the format string is being replaced. These two numbers are followed by a format string fragment that is added into the format string. In our implementation, insert messages are simply replace messages with a zero length and delete messages are replace messages with an empty fragment.

6. LOSS AND ERROR

In the N-CET application, packets that are uninteresting can be dropped and errors can occur. Since the compression of the packets depends on the previous packet, any loss of a packet causes all the following packets to be meaningless. Instead of reporting the value at each packet as the change in value from the previous packet, we occasionally send baseline packets and all subsequent packets are expressed as the change in value from the last baseline. These baseline packets can then be flagged as high priority so that the application will not drop them. Also in lossy environments, these baseline packets can require acknowledgement to ensure delivery.

Figure 12 shows results of experiments comparing cost of acknowledging and resending lost packets with loss of compression due to packets being further from the baseline. If every packet is a baseline, then every packet must be sent and acknowledged, but if a packet uses a baseline from many packets ago, then correlation diminishes and compression is reduced. As the number of packets sent between baselines increases, the compression increases until it reaches a point where the benefit of correlation is lost. For our datasets (discussed in section 8) this point was reached between 90 and 120 packets.

The optimal number of packets between each baseline was found to be somewhere between 15 and 30. In our experiments, 20 packets were sent between each baseline.

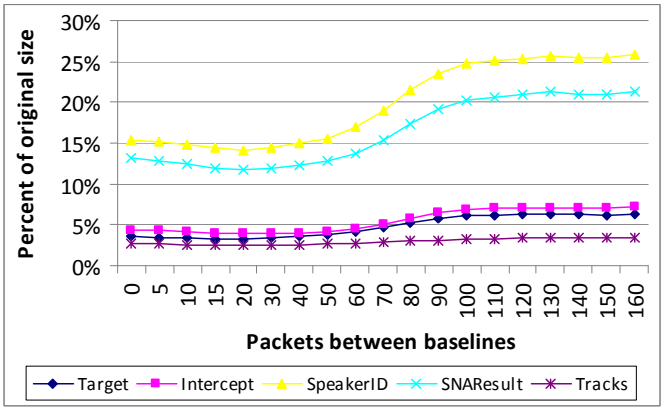

Figure 12 Baseline period

7. PACKET HEADER

In order to encode the extra information required to make the algorithm work, we append one byte of header information to each XML data packet sent over the network.

The first two bits indicate whether the packet is a new baseline, a format string update, a standard packet, or the beginning of a new transmission.

The next two bits represent the format string version so that if a format string update gets lost, the receiver will be able to detect that it is using an outdated version of the format string. It can then request a retransmission of the update from the sender or any neighboring nodes that may have heard the broadcast. If the number of versions exceeds eight then the version number simply wraps back to zero. In the case where four or more format string update packets are lost in a row, the receiver will use the wrong format string to attempt to decompress the data. All the packets will seem corrupted or will be erroneously decompressed. In a highly lossy environment, the number of bits can be increased to eliminate the errors.

The last four bits of the header byte are used for the baseline index and are handled similarly to the format string version bits. More bits are used for the baseline index since it is expected to change much more frequently. The main difference is that missing a set of baseline updates will not make the data appear corrupt but will only cause the data to be decompressed incorrectly. In high loss environments, the baseline packets can be sent both as baselines and as regular packets so that the regular packet can be decompressed and compared against the baseline packet to detect error in much the same way as the errors are detected in [28]

8. DATASETS

The N-CET project produced four different XML datasets with various types of data. We also used one dataset from a joint project between the U.S. Navy and Air Force which tracked aircraft and ships.

8.1. RFINTERCEPT

The UAVs were equipped with Electronic Intelligence sensors capable of intercepting RF signals (radio communications). These rfIntercept packets were sent at the beginning and end of each intercepted transmission and (depending on the duration) at several points in the middle of the transmission. The packets contain several pieces of information including ID, position, and heading of the UAV; radio frequency and transmission duration; and a line of bearing from the speaker to the UAV.

8.2. RFTARGET

If multiple UAVs intercepted the same transmission, the lines of bearing were used to triangulate the source of the communication and rfTarget packets were generated

containing data such as the estimated position of the speaker and the IDs of the rfIntercepts used in the triangulation.

8.3. SPEAKERID

The audio from the intercepted communications was compared to a database of previously captured voice samples to identify the speaker. The speakerID packets contained identifying data on the transmission and the ID and name (if known) of the speaker as well as the output of the voice matching algorithm such as the confidence.

8.4. SNARESULT

The N-CET project also utilized social network analysis techniques to identify the importance of the various speakers. The snaResult packets generated for each contain the list of related speakers who communicated on the same frequency during the same time period and the output of the Key Player Algorithm which assigns a rank to each speaker.

8.5. TRACKS

The joint tracking project produced XML data packets of a significantly higher complexity than the N-CET data. The packets contained unique IDs of the tracked vessel, the tracking entity, and the last entity that tracked the vessel; timestamps; position, direction, and speed of the tracked vessel; the type of sensor and platform used; and many identifying features of the vessels. The dataset only had a limited number of packets of real data so we generated 10,000 synthetic packets based on the real data to make the track dataset closer to the size of the others.

9. RESULTS

We compared the compression of TinyPack XML against Deflation, XMill, XMLPPM, and PAQ over the four datasets in both delay tolerant and real time experiments measuring compression, latency, processor usage, RAM requirements, and energy consumption.

The first result set in Figure 13 shows the results from the delay tolerant study. All the data was collected prior to compression and compression was done on the entire dataset at once. (XMill and XMLPPM require a single root tag so an arbitrary <r> </r> tag pair was added around the rest of the data for these algorithms). Results show Deflation and WBXML performing somewhat worse that the others with TinyPack XML slightly outperforming XMill and XMLPPM and slightly underperforming the expensive "ideal" PAQ algorithm. WBXML and TinyPack are designed for smaller XML documents and were not expected to perform ideally in a delay tolerant environment. The dataset schemas were very complex which negatively affected XAUST. To be fair, DTDs were rewritten in order to more closely match the actual data.

9.1. COMPRESSION RATIO

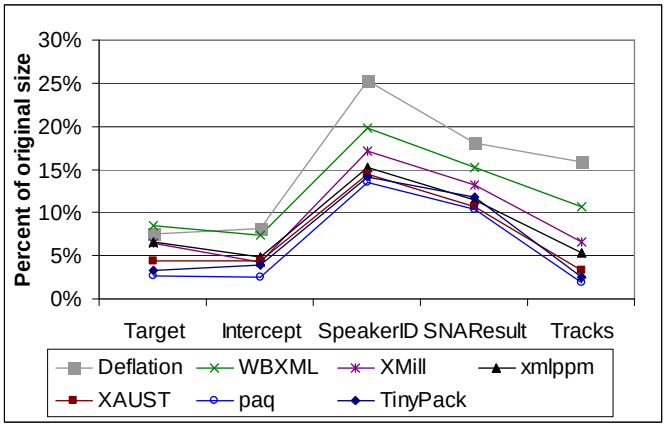

Figure 13 Delay tolerant compression results

The next experiments considered real-time environments where each data sample was compressed and transmitted as it was collected. Data was collected by compressing each sample individually. PAQ also has an incremental infrastructure for using data from previously compressed samples to assist in the compression of future samples. Results are shown in Figure 14 for real-time compression using all the algorithms and the PAQ incremental version.

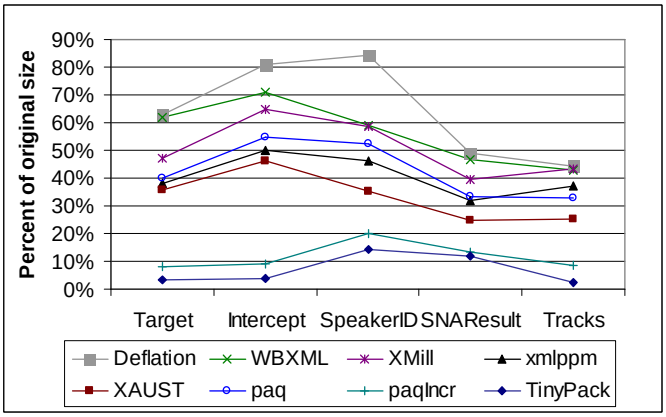

Figure 14 Real-time compression results

As expected, the incremental nature of TinyPack XML caused it to significantly outperform the other algorithms run on the individual samples; however, TinyPack XML also surpassed the incremental PAQ algorithm. The delay tolerant PAQ algorithm makes multiple passes through the data so restricting it from looking at past samples reduces its performance. TinyPack XML was designed specifically for real-time systems so it performs identically in both environments.

9.2. LATENCY AND PROCESSING TIME

The results for latency did not differ greatly between the datasets. In order to reduce clutter on the graph, the results are shown as the average across all four datasets.

In the delay tolerant experiments, all the data was collected before sending so latency was not considered.

Real time experiments for latency were performed using TOSSIM [31], which simulates the open source TinyOS operating system that runs on many sensors. TOSSIM simulated Crossbow Technology's MicaZ motes [30]. These motes are an example of a resource constrained system where bandwidth and energy are limited. PAQ required more RAM than the motes have available and in tests on a standard desktop computer took over twice as long to send due to the greatly increased processing time and is not included in the results. Latency results are shown in Figure 15 in terms of both processing and sending time. Since TinyPack requires more complex parsing of the XML data, the processing time is significantly higher, but the total time is lower since less time is needed to send.

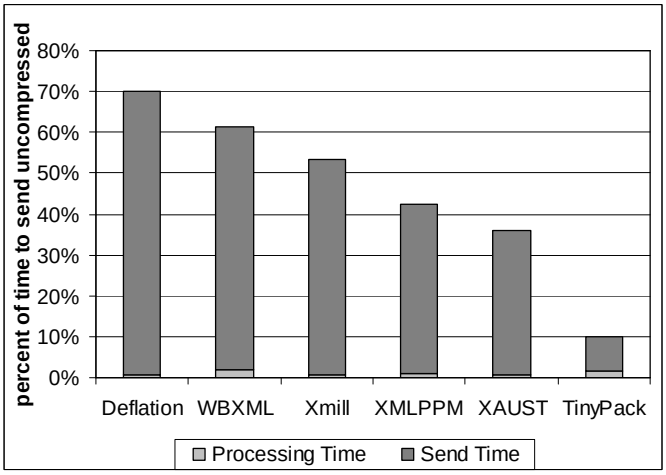

Figure 15 Latency

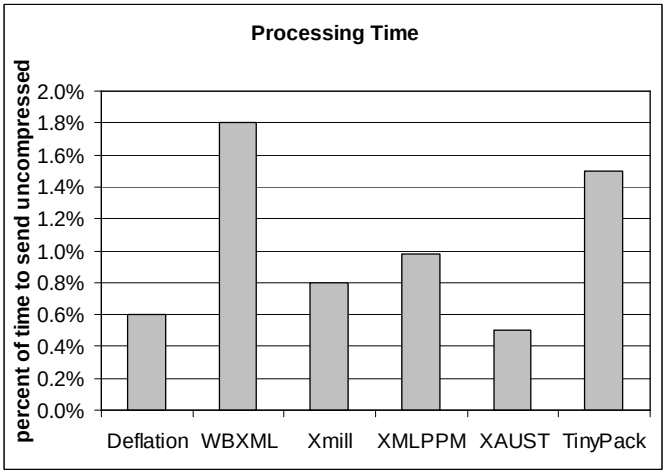

Figure 16 Processing time

Processing time is shown separately in Figure 16. On most systems (especially wireless networks), processing speed is exponentially higher than transfer speed so it is almost always beneficial to sacrifice some processor use to reduce the amount of data that would need to be sent.

9.3. ENERGY CONSUMPTION

The energy required to compress the data is basically a function of the processing and sending time. Energy is primarily important in wireless networks in which the nodes run on batteries. Results are similar to that of latency and are shown in Figure 17.

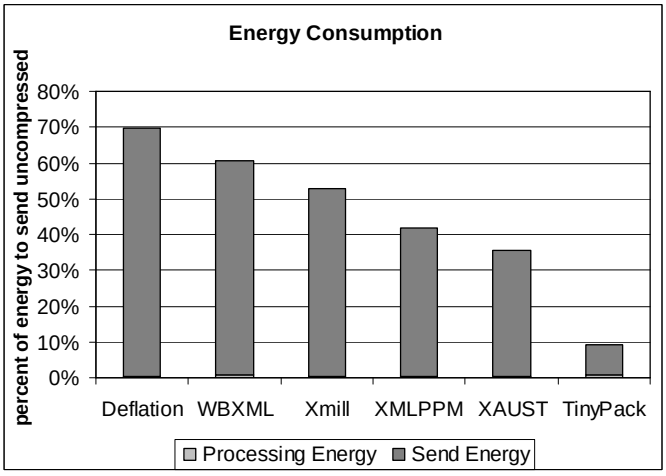

Figure 17 Energy consumption

9.4. RAM USAGE

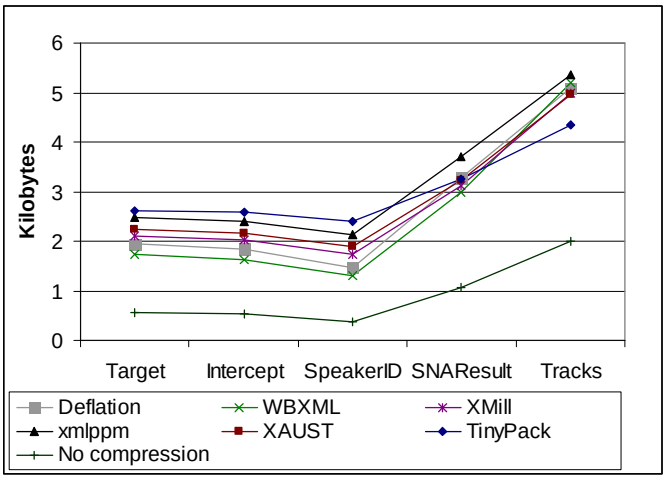

Figure 18 Ram usage

For all the methods except for PAQ, RAM required to compress each individual packet naturally was highly dependent on the original size of the packet. RAM requirements for the largest packet in each dataset are shown in Figure 18. With the exception of PAQ which requires at least 233 MB of RAM, TinyPack XML uses a little more RAM than the other methods for most of the datasets since it maintains lightweight compressors for each argument in the format string. The SNAResult and track data contained more static structure than the other datasets and required less RAM for TinyPack since the static portions of the structure are only stored in one place and are only compressed once.

10. CONCLUSIONS AND FUTURE WORK

TinyPack XML quickly and effectively compresses semi-structured, XML data. It is very useful for the N-CET project and other applications in reducing required bandwidth and storage in the network without introducing delay. It would be interesting to see how TinyPack XML performs on poorly structured data.

The other existing compression methods could be modified to only use current and previous data to compress. This would make the comparisons more accurate and would better show the benefits of TinyPack XML.

TinyPack XML successfully exploited the correlation of consecutive samples taken from a single sensor and the redundancy in single XML documents; however, samples taken from nearby sensors at the same time (or within some time range) also can be heavily correlated. Similarly, the XML data from the various types of data also contained some correlations. Cross referencing other packets from other sensors or other types of data could further increase the compression.

III. ON COMPRESSING DATA IN WIRELESS SENSOR NETWORKS FOR ENERGY EFFICIENCY AND REAL TIME DELIVERY

Wireless sensor networks possess significant limitations in storage, bandwidth, processing, and energy. Additionally, real-time sensor network applications such as monitoring poisonous gas leaks cannot tolerate high latency. While some good data compression algorithms exist specific to sensor networks, in this paper we present TinyPack, a suite of energy-efficient methods with high-compression ratios that reduce latency, storage, and bandwidth usage further in comparison with some other recently proposed algorithms. Our Huffman style compression schemes exploit temporal locality and delta compression to provide better bandwidth utilization important in the wireless sensor network, thus reducing latency for real time sensor-based monitoring applications. Our performance evaluations over many different real data sets using a simulation platform as well as a hardware implementation show comparable compression ratios and energy savings with a significant decrease in latency compared to some other existing approaches. We have also discussed robust error correction and recovery methods to address packet loss and corruption common in sensor network environments.

1. INTRODUCTION

Many real-time systems incorporate wireless sensor networks (WSNs) into their infrastructure. For example, some airplanes and automobiles use wireless sensors to monitor the health of different physical components in the system, security systems use sensors to monitor perimeters and secure areas, security forces use sensors to track troops and targets. It is well known that wireless sensor networks possess significant limitations in processing, storage, bandwidth, and energy. Therefore a need exists for efficient in-

network data compression algorithms that do not require delays in processing or communication while still reducing memory and energy requirements.

The idea of data compression has existed since the early days of computers [1][2][3], many new data compression schemes [5][6][7][8][9] for wireless sensor networks have been proposed recently to address various constraints and limitations in wireless sensor networks. 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, which is not suitable for many real-time sensing applications such as those mentioned above.

We propose TinyPack, a suite of data compression protocols for real-time sensor network applications. TinyPack reduces the amount of data flowing through the wireless network, optimizes bandwidth usage, and decreases en without introducing delays. First the data is transformed by expressing the sensed values as the change in value from the previous sensed data. 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 further 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 occurs. TinyPack is a lossless compression algorithm where the data can be decompressed at the sink or base station without any loss of granularity or accuracy.

Standard Huffman [1] 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,

thus making them ineffective in a WSN environment. 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 also use the notion of an all-is-well bit and perform some 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. Experiment and simulation results show a significant reduction in bandwidth, latency, and energy consumption compared to the other methods. One of the proposed algorithms also reduces RAM and processor usage while the others show a further reduction in bandwidth, energy, and latency at the cost of increasing the memory and processing requirements.

In summary, this paper makes the following contributions:

An improved set of static codes optimized for sensor data and computational efficiency in processing.

Algorithms for 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 use of an all-is-well bit that further increases compression performance and efficiency.

A novel and effective error detection and recovery method to handle missing and corrupted packets.

Extensive experiments comparing several performance metrics considering various approaches using many different real sensor data sets using simulation as well as a hardware platform.

2. BACKGROUND

2.1. 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. 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 12 generate the codes using the Huffman tree in Figure 19. Huffman codes were shown to be optimal for symbol by symbol compression in [1].

Table 12 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

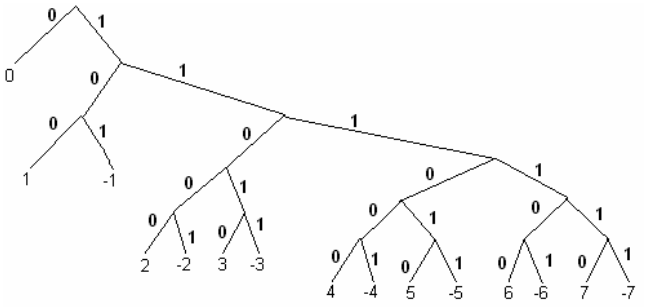

Figure 19 Huffman tree

2.2. 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 that 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. Thus, 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.

2.3. FRAMES

In delta compression (as with most compression schemes), a dropped packet can render following packets useless or at least complicated to decompress. Thus 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. Therefore, sending independently compressed frames of data also allows additional flexibility for the compression to be more specific to the current state of the system.

3. RELATED WORK

3.1. 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 13 S-LZW with mini-cache

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

Table 13 shows S-LZW and LZW compressing the string AAAABAAABCC using the mini-cache. Since every single character is pre-loaded into the dictionary, the algorithm begins by looking at the first string of two characters in the stream. If the string is in the dictionary, the next character is appended until the string no longer has a

dictionary entry. Then that new string is added to the dictionary and the known string (the new string minus the last character) is encoded into the output. The new output column shows a 1 and the mini-cache location if that symbol was in the cache or a 0 and the dictionary location otherwise. The other columns show the new entries in the dictionary and mini-cache and the total number of bits required for compression without or with the cache. Note that without the cache every symbol is exactly nine bits.

For example, for the first line of Table 13 the compressor begins by looking at the first character of the string "A." Since "A" is a single character it is already in the dictionary and the compressor looks at the string "AA." That string is not in the dictionary so it is added to the end (location 257) and the single character "A" is encoded (as the integer 65) and the algorithm continues with the second "A" as the next character in the stream. Since "A" was not in the mini-cache the output comes from the dictionary and both "A" and "AA" are added to the cache.

3.2. 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. For the values in a stream, the initial value is encoded as its difference from 0 and each successive value is encoded as its difference from the previous value. The codes are constructed by concatenating prefix and a suffix bits to represent the change value. Fewer bits are used for the smaller changes under the assumption that values typically change relatively slowly over time. The static codes are shown in Table 14 with anything past level 7 following the pattern of the last three levels.

Table 14 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

For example, a 0 value would be encoded as "00" ("00" prefix and no suffix) and -3 would be encoded as "01100" ("011 prefix and "00" suffix).

If it is known that the change values will not fall outside of a certain range, then the '0' bit in the prefix for the last level can be removed. For example in Table 14 the prefix for level 7 could be "1111" if -127 and 127 were the minimum and maximum possible change values.

3.3. GAMPS

Many lossy compression schemes have also been proposed such as [9]. GAMPS compresses the data from multiple sensors by grouping sensors with correlated values. The signals are approximated keeping within a parameterized maximum error. The Facility Location problem is then used to groups the sensors with the highest correlations and select baseline sensors which best represent the group. The values from the remaining sensors in each group are expressed as a ratio of the value of the baseline.

An example is shown in Figure 20. Graph (a) shows relative humidity signals from different sensors. In graph (b) the signals have been approximated. Graph (c) shows the fourth signal from graph (b) selected as the baseline for the group. The final graph (d) shows each of the other five signals as a ratio of the baseline signal. The data in graphs

(c) and (d) is then identical to the data in (a) within some error threshold but can be compressed much more than the original data.

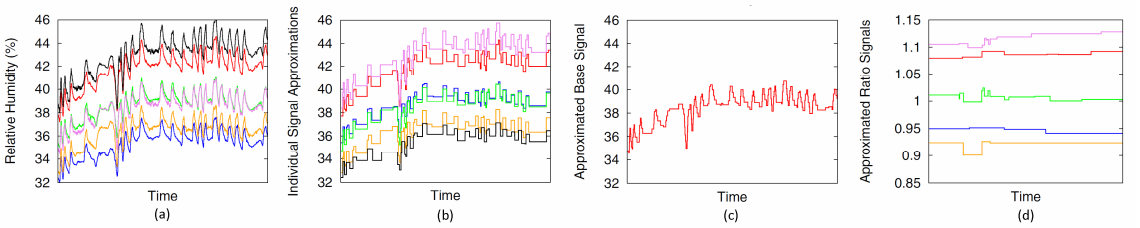

Figure 20 Gamps example

GAMPS achieves excellent compression ratios with low maximum error but requires that all the data be collected before compression and so is not suited for applications which require no loss or for the compression to be performed in real time.

3.4. PIPELINED IN-NETWORK PROCESSING

Other schemes have been introduced which depend on the network topology and routing. In [5] compression is achieved using pipelining. Data is gathered at each aggregation node in a buffer for some amount of time. During that time, several data packets with a matching prefix are combined into one. Following the prefix in the packet is a suffix list which gives the unique suffix to the common prefix from each of the original packets. This scheme is illustrated in Figure 21. Three packets each containing three items of data are compressed on the first item with a prefix of length three, the other two items remain uncompressed. This reduces the data size from 33 bits to 27 bits.

The size of the prefix is determined by the user of the application and remains static. The shared prefix system can also be used for timestamps and sensor IDs to maximize the reductions in size.

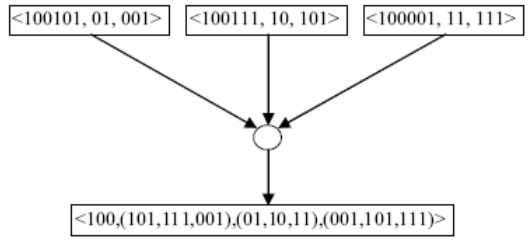

Figure 21 Pipelined compression

This scheme can be very effective if there is much redundancy inherent in the value prefixes; however, the compression is only done at aggregating nodes and depends on sample rates to be very effective.

3.5. CODING BY ORDERING

Another routing method is proposed in [6] where the order of packets collected at an aggregation node can indicate the value sensed at a different node. A packet containing the data tuples from n sensors can be arranged in a total of n! unique permutations. If the number of possible sensed values is relatively small, these permutations can be used to recreate dropped values from one or more sensors (see Table 15).

Table 15 Value indicated by order

Packet permutation	Integer Value
N1,N2,N3	0
N1,N3,N2	1
N2,N1,N3	2
N2,N3,N1	3
N3,N1,N2	4
N3,N2,N1	5

If there are n sensor nodes in a network and a packet at an aggregation is sent values from m different nodes, assume that out of those m nodes a total of l nodes' values are dropped and encoded. Given only the (*m-l*) values, there are (*n-m+l* choose *l*) possible combinations of IDs the dropped nodes can have. If there are k possible data values, there

are k^l possible combinations of values and IDs. Since there are (*m-l*)! possible permutations within the packet, l can be chosen as large as is possible without violating the following inequality

$$(m-l)! \geq (n-m+l \ \ choose \ \ l)k^l$$

For example, when n = 256, k = 16, and m = 100; l could be set as high as 44, so only 56% of the data would need to be sent. This scheme, however, performs well only when n is relatively large compared to k. If there is a wide range of possible data values, then some form of tolerated error would need to be introduced to accomplish any amount of reduction.

3.6. SUMMARY

We compare all the previously listed algorithms and the algorithm presented in this paper (TinyPack) across a number of compression algorithm characteristics in Table 16.

Table 16 Characteristics of sensor compression techniques

Characteristic	S-LZW	LEC	GAMPS	Pipelined	Coding by Ordering	TinyPack
Runs on a single sensor	Yes	Yes	No	No	No	Yes
Relies on temporal locality	Sometimes	Yes	Yes	No	No	Yes
Relies on spatial locality	No	No	Yes	Yes	No	No
Collect data prior to compressing	Some	None	All	Some	None	None
Algorithm adapts as data changes	Yes	No	Yes	No	No	Yes
Requires time synchronization	No	No	Yes	No	Yes	No
Requires related sampling intervals	None	None	None	Similar	Identical	None
Achieves lossless compression	Yes	Yes	No	Yes	Yes	Yes
Loss due to dropped packets or errors	Frame	Frame	Packet	Packet	Packet	Frame
Incorporates error detection	No	No	No	No	No	Yes

The algorithms presented in this paper and used for comparison concern lossless compression which can be achieved in real time at the sensing node.

4. 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.

5. OUR PROPOSED APPROACHES

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 compression 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.

5.1. 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 causing most changes reported to be 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 17 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 2*B* + 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. Then *C* is constructed as six 0 bits, followed by the binary representation of |57| (i.e. 111001), followed a 0 sign bit since 57 is positive. The entire code *C* is then 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 18. 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 18 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, for example 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 22 shows the results of the TinyPack initial codes (TP-Init) compared to the standard Deflate algorithm, S-LZW, and the LEC codes.

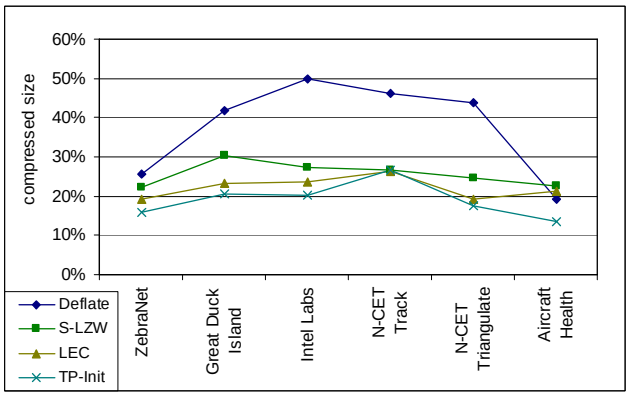

Figure 22 Initial codes compared to deflate, S-LZW, and LEC

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. This is due to the high degree of variance in that dataset. 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.

5.2. 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 to collect all the data and count the total frequencies of all the values before sending the currently collected data. To avoid the need to transmit them, 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 and 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 in order that 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. 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. 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
```

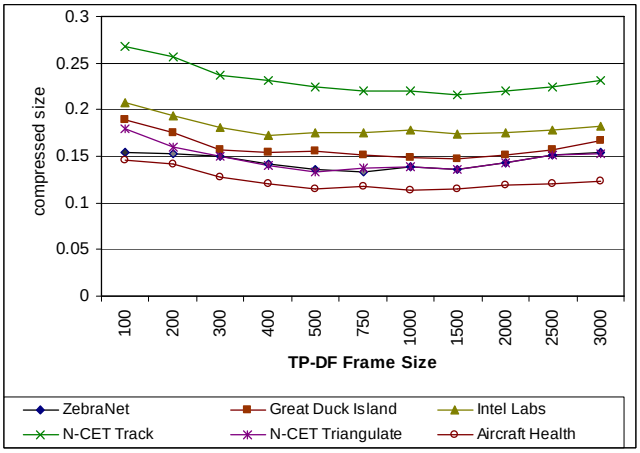

Figure 23 Frame size analysis for tinypack with dynamic frequencies

We ran TP-DF on all the datasets with a varying frame size. Results are shown in Figure 23. 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, thus again negatively impacting the compression size.

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

5.3. 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. 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 *n*th 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. 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. 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. Therefore, the Huffman tree in Figure 19 can be compressed into Table 19 where the table is stored on the sensor as an array 1-indexed on the tree level.

Table 19 Compressed tree

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

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

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 20.

Table 20 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 21.

Table 21 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

The probability of a level is computed as the sum of the probabilities of the nodes at that 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. 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. Thus, 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 allowing confidence intervals to 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 23). Again the experiments were run with a window size of 512.

Figure 24 shows the results of running both the dynamic frequencies (TP-DF) and running statistics (TP-RS) over the datasets compared to the other methods. 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.

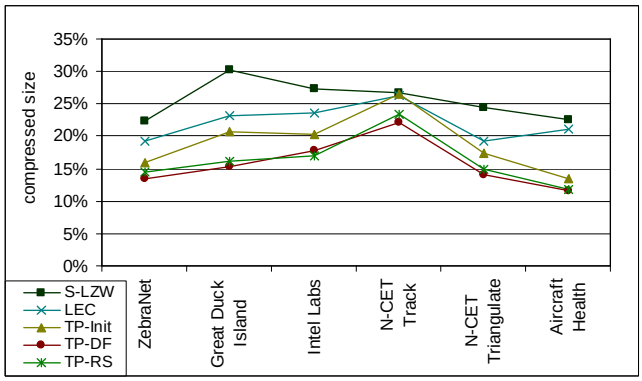

Figure 24 Tinypack with dynamic frequencies and running statistics

5.4. 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. 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. This idea has been used several times previously in sensor networks [19][20][21].

The datasets were affected differently by adding this. Figure 25 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.

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.

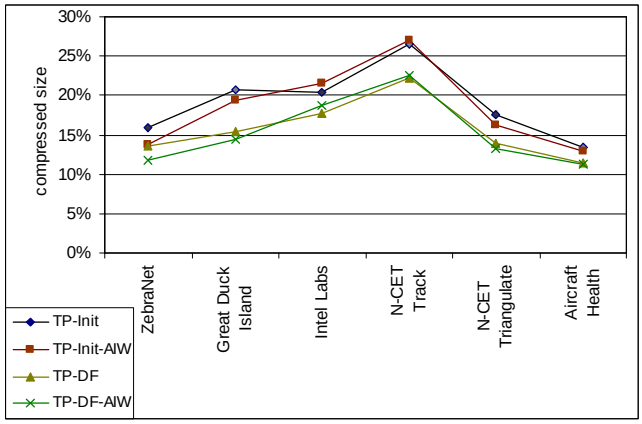

Figure 25 Effects of all-is-well bit

5.5. BASELINE FREQUENCY

In some applications, packets that are uninteresting can be dropped and drops can also occur accidentally. Since the compression of the packets depends on the previous packet, any loss of a packet causes errors that propagate to all the following packets. Instead of reporting the value at each packet as the change in value from the previous packet, we

examined the cost of only occasionally changing the baseline of which the change is reported. So instead of every packet being a baseline, baseline packets can be sent at different intervals and all subsequent packets are expressed as the change in value from the last baseline. These baseline packets can then be flagged as high priority so that the application will not drop them. Also in lossy environments, these baseline packets can require acknowledgement to ensure delivery. We experimented with static baseline intervals and using statistics of the data to determine when to send the new baseline. Figure 27 and Figure 27 show the effects on compression of changing the baseline frequency using static intervals and sending a new baseline when the packet size increased above a threshold compared to the average and standard deviation of the previous packet sizes.

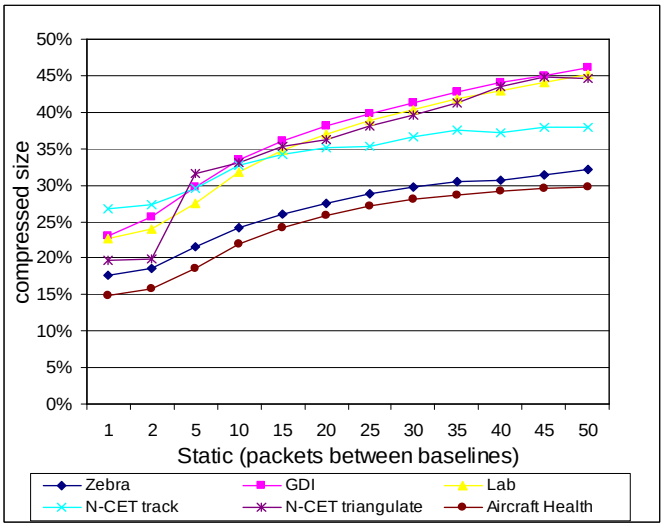

Figure 26 Baseline frequency (static)

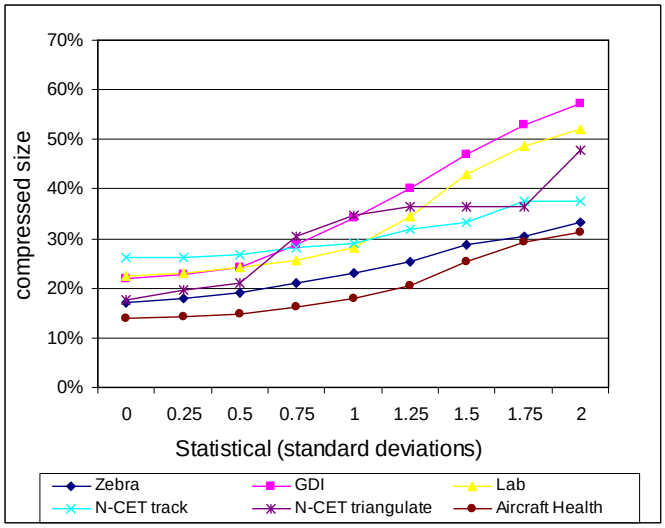

Figure 27 Baseline frequency (dynamic)

The results for the statistical approach were scaled using the total number of baseline packets sent to calculate the frequency and compared to the results for static frequencies for each of the datasets. The average results were almost identical making the static methods preferable since they require less processing, are more intuitive to implement and parameterize, and were more consistent in their effects.

As with most compression algorithms, the data is highly susceptible to dropped or corrupted packets. If one of the baseline packets is dropped or corrupted, then the data following that point would be unable to be decompressed. We experimented on and analyzed the cost of retransmitting baseline packets in scenarios with varying degrees of error. Error detection and correction are discussed in more detail section 7.

Figure 28 shows the cost of retransmission of the dropped baseline packets. As expected, the cost of retransmission drops quickly as the number of packets between baselines increases. The probability of a dropped packet being a baseline and thus requiring retransmission is inversely proportional to the number of packets between baselines resulting in the hyperbolic shape of the cost curve.

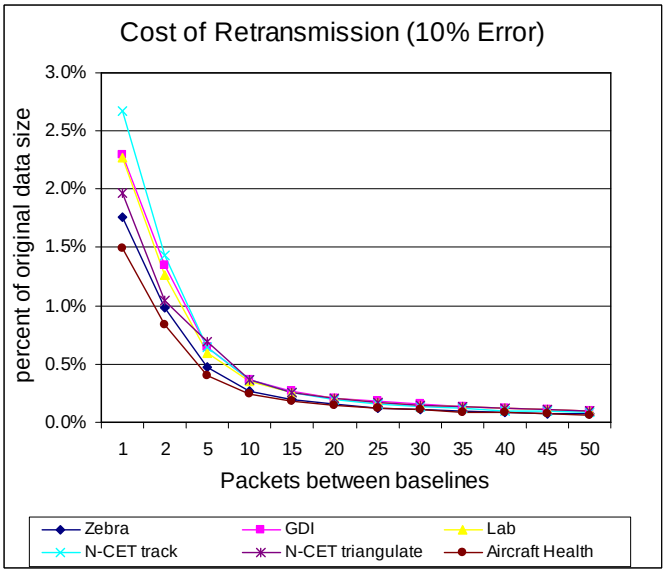

Figure 28 Retransmission

As expected, the cost of retransmission drops quickly as the number of packets between baselines increases. The probability of a dropped packet being a baseline and thus requiring retransmission is inversely proportional to the number of packets between baselines resulting in the hyperbolic shape of the cost curve.

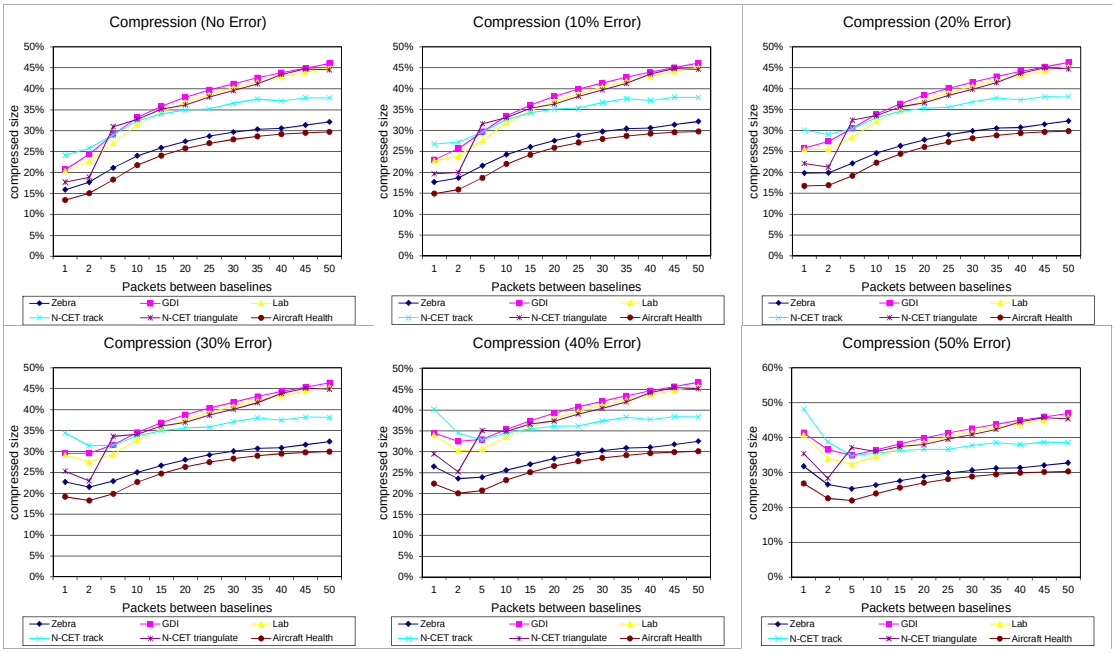

Figure 29 Compression with retransmission

Cost of retransmission was directly proportional to error percentage. The graphs for the other error amounts were omitted since the shape of the curves is identical. Figure 29 shows the total size of the transmitted compressed data including retransmissions of dropped baseline packets. This includes dropped retransmissions. For example, with 10% error, each baseline packet would be sent an average of 1.111 times and with 50% error, each baseline would be sent an average of twice. As the error rate increases, the cost of retransmission increases. As in Figure 28 the increased cost is greatest when the number of packets between baselines is low. As the number of packets between baselines increases, the added cost becomes negligible and the graphs become identical.

5.6. 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.

6. PHYSICAL IMPLEMENTATION USING SENSOR NETWORK TEST-BED

We implemented the algorithms on a network of seven Mica2 sensors running the TinyOS operating system. One sensor served as the base station for the network and the other sensors were loaded with data from the datasets. The sensors then compressed and sent that data to the base station using each of the different algorithms. All the sensors were time synchronized and sent data using time division multiplexing. For datasets with

more than six sensing nodes, experiments were done on the data from six at a time until the data from all sensing nodes had been passed through the network.

Each experiment was run separately in order that the measurement of one metric would not affect the others. For example, if the sensors tracked RAM usage while processor utilization was being measured, the results would be slightly inflated.

6.1. COMPRESSION

The results from all the previous compression experiments are combined in Figure 30 which shows the compressed size of each dataset. Shown are 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). As expected TP-DF performed the best in terms of compression compared to the other algorithms. The all-is-well bit increased the performance over some of the datasets.

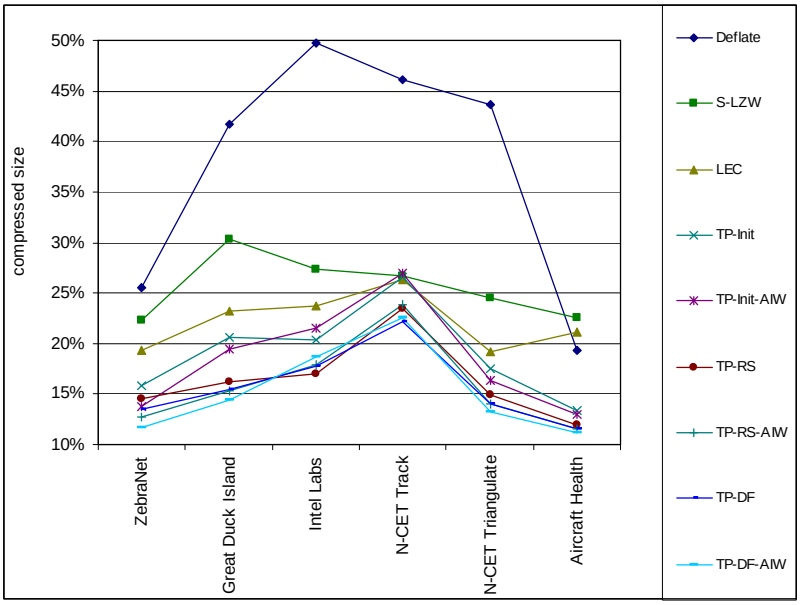

Figure 30 Full compression results

To summarize, we calculate the entire compression of all the data across every dataset and normalized the results to give equal weight to each dataset in Figure 31. The all-is-well bit added a slight benefit in the average case although its usefulness depends heavily on the characteristics of the data sensed. As it can be observed, the TinyPack algorithms provide compressed sizes of 11% to 27% outperforming the other methods which range from 19% to 50%.

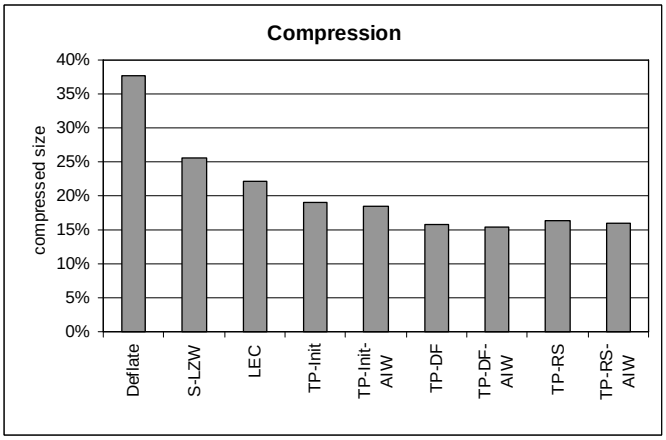

Figure 31 Compression summary

6.2. 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 32 shows the performance of the TinyPack and LEC algorithms compared to the performance of a theoretical optimal algorithm. Deflate and S-LZW both resulted in greater compressed sizes and are not shown here to allow for greater precision in the figure. 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.

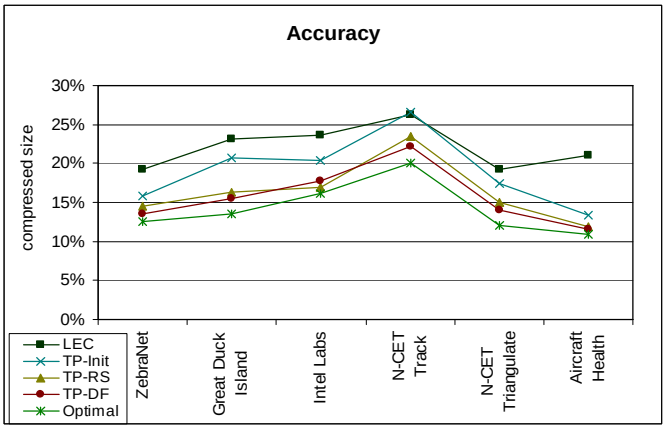

Figure 32 Accuracy

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.

6.3. 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 much faster than radio data rate for wireless sensors (for example, the Mica2 mote [15] has a 16 MHz processor and a 38.4 kbps high data rate radio). For the Mica2 motes, latency is decreased proportionally to the compressed size of the data. Thus, TinyPack has a decrease in latency of 80-85% compared to uncompressed data. Latency was measured at the base station by querying the system clock at the beginning and end of each transmission and at the beginning of each nodes time window to determine the processing time. For S-LZW the nodes logged and averaged their own wait times and sent that data at the end of the experiment.

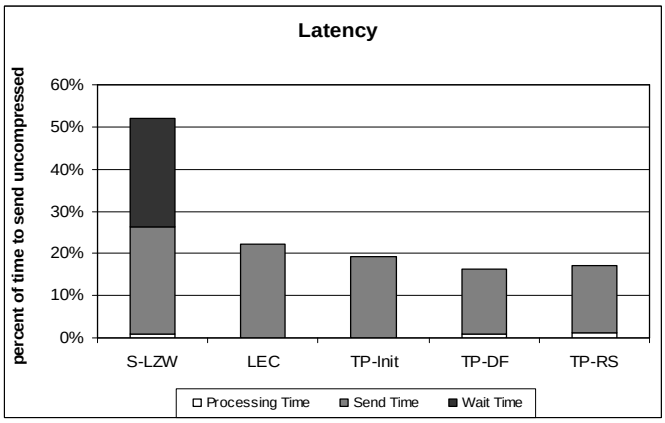

Figure 33 Latency

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 33 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 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. Send time is directly proportional to compression (shown in subsection 6.1) and processing time is directly proportional to the processor utilization (shown in subsection 6.5).

6.4. RAM

The maximum amount of RAM utilized by each algorithm for each dataset is shown in Figure 34. S-LZW is designed to work on any generic dataset and uses the same compressor for every value in a sensed vector making the RAM usage 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.

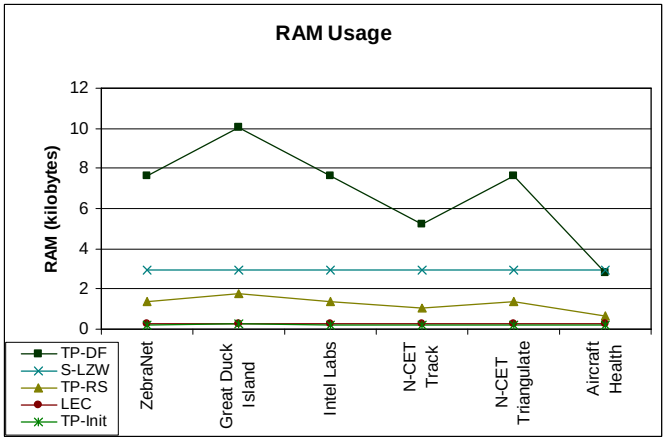

Figure 34 Ram usage

6.5. 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 to prevent the processor utilization from being affected by the compression ratio. Figure 35 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 from processing is included in the results of the energy simulation in Figure 36.

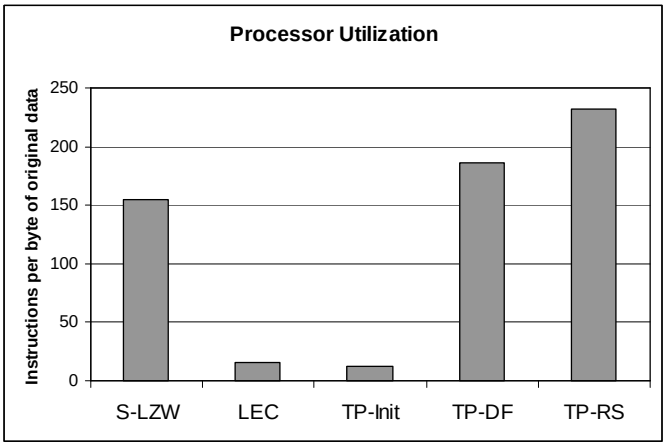

Figure 35 Processor utilization

7. EXPERIMENTAL RESULTS USING A SENSOR NETWORK SIMULATOR

Experiments were performed using TOSSIM [17], 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 verify the experimental results as well as measure energy consumption and to test the algorithms under larger networks and different architectures. In addition to TOSSIM the PowerTOSSIM [18] simulator was used. PowerTOSSIM is built on top of TOSSIM and provided the capabilities of measuring simulated energy consumption and latency.

7.1. ENERGY USAGE

Energy consumed for compressing, writing to memory, and transmitting was measured using PowerTOSSIM. Results shown in Figure 36 are again scaled to a percentage of the cost to send the data uncompressed and averaged over all the datasets. As with latency, the all-is-well bit in each case decreased the energy usage by less than half a percent. Energy usage data was not collected for the Deflate algorithm since it was included only as a compression benchmark and was not implemented in PowerTOSSIM.

As can be seen by comparing Figure 31 and Figure 36, energy results closely matched the compression results since most energy is consumed while transmitting the data.

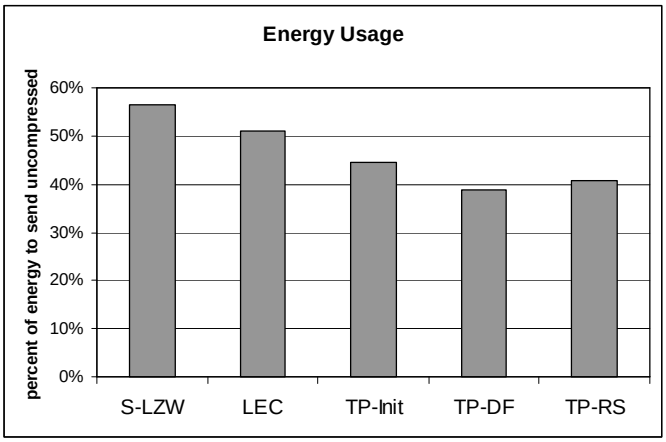

Figure 36 Energy usage

7.2. LATENCY IN A MULTIHOP ENVIRONMENT

Experiments were performed to show the effects of the algorithms in a multi-hop environment. Sensing nodes sent data to the base station through a varying length series of forwarding nodes. For sensors with a slower processor or faster radio, the processor utilization becomes a greater factor, but in a multi-hop environment, the algorithms with the best compression ratio still outperform the others. Modifying the simulation to use a data rate of 2.5 Mbps radio like the Manchester-coded sensors in [16] generated the latency results shown in Figure 37. The left graph shows the latency on a single sensor and the right graph shows how latency changes with the number of hops. As the average number of hops increases, latency approaches sending time since there is no additional processing needed when forwarding the compressed packets. After two or three hops the algorithms with the best compression ratio have the lowest end-to-end latency even for sensors with high speed radios.

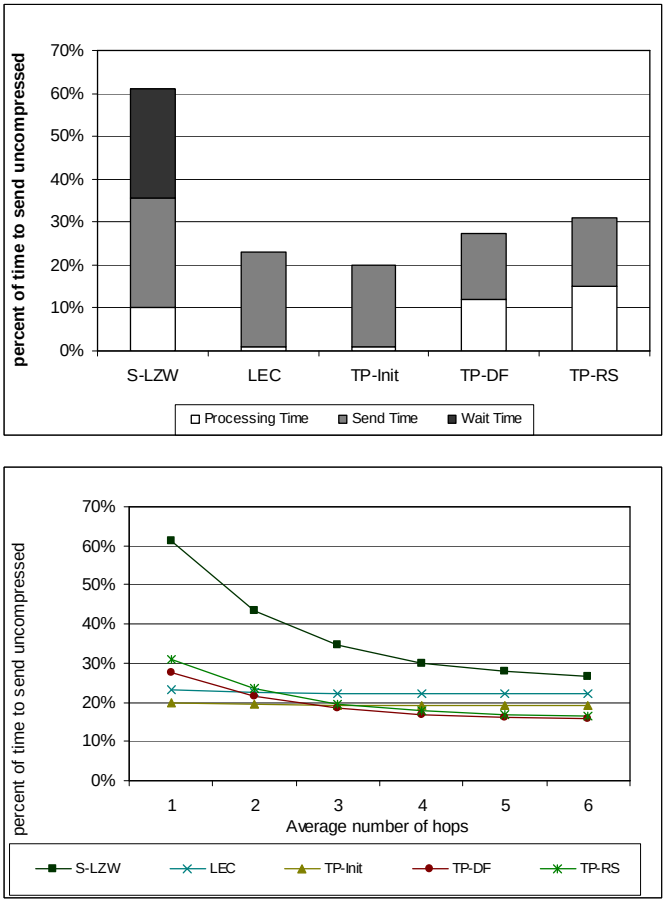

Figure 37 Latency for high speed radio single and multi-hop

8. ERROR DETECTION AND RECOVERY

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.

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

8.1. DROP DETECTION

For dropped packets, the probability of a subsequent error "correcting" the value and causing the errors to be undetected can be computed using a state diagram and transition matrix. The state number is defined as the difference between the value calculated at the receiver and the value transmitted by the sender. For example, state 3 represents that the receiver believes the value to be 3 greater than it really was and state 0 represents either no error or undetectable error. Since transitions can go from any state to any other state and the number of states is equal to twice the number of possible values, the diagram is far too complex to include. The probability of an error causing a transition from a state X to a state Y is

$$P(X,Y) = 2^{-2\lceil \log_2(|X-Y|+1)\rceil - 1}$$

Clearly $P(X,Y) = P(Y,X)$ so the probability of transitioning from X to Y and then from Y back to X is just $P(X,Y)^2$. The probability of a second error correcting the value and causing both errors to go undetected is represented by transitioning from the initial state 0 to any state X and back and is

$$\sum_{x=-\infty}^{\infty} P(0,X)^2 = \sum_{x=-\infty}^{\infty} 2^{-4\lceil \log_2(|X|+1)\rceil-2} \approx .0357$$

Therefore the probability of two drops going undetected in a frame is roughly 3.57%. 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. 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 22.

Table 22 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%

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.

8.2. SINGLE BIT ERROR DETECTION

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 18 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:

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

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

As the height of the tree approaches infinity, $E(P)$ approaches 2 and $E(S)$ approaches 1. The probability of a bit error 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 either be detected at the end of the packet transmission or would cause the data to vary so greatly that the

probability of a future error correcting the value is exponentially less than if the error was in the suffix.

Suffix bit errors cause the error in value to change in the same way as dropped packets. Thus, the probabilities of errors going undetected are one third those of the dropped packets.

8.3. CORRECTION

If the data is sent based on a sampling interval or if the packet headers contain sequence numbers, then the above error detection mechanisms can easily be used to reconstruct dropped or corrupted packets. In the case of a single dropped packet, the values dropped are equal to the difference between the calculated value at the receiver and the value of the error detection packet. For example, assume again that a temperature sensor sensed values at 23, 25, 28, and 29. The values encoded and transmitted would then be 23, +2, +3, and +1. Assume that the packet containing the +3 value was dropped and the calculated value at the receiver is 23+2+1=26. At the end of the frame, the sender transmits the non-encoded real value of 29 as the error detection packet. Since 29-26=3, the receiver can instantly calculate the missing value as +3. In the case of multiple dropped packets, the difference represents the total error over all drops. For consecutive drops, we simply divide the total error by the number of drops and assign that value to each missing packet. For non-consecutive drops, the values are scaled based on the ratio of the previous and next packet surrounding each missing packet.

We experimented using the same frame size of 512 and a 1% Poisson distributed drop rate. Table 23 shows the average error compared to actual value of the dropped packet as well and the percentage of errors greater than 1%

Table 23 Error correction

Dataset	errors	average	>1%
ZebraNet	57	0.18%	2.5%
Great Duck Island	7642	0.34%	4.2%
Intel Labs	23035	0.07%	1.3%
N-CET Track	4607	0.26%	3.4%
N-CET Triangulate	2231	0.19%	2.9%
Aircraft Health	4586	0.12%	1.7%

9. CONCLUSIONS AND FUTURE WORK

The TinyPack suite of protocols 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. 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.

IV. ENERGY EFFICIENT DISTRIBUTED GROUPING AND SCALING FOR REAL-TIME DATA COMPRESSION IN SENSOR NETWORKS

Wireless sensor networks possess significant limitations in storage, bandwidth, and power. This has led to the development of several compression algorithms designed for sensor networks. Many of these methods exploit the correlation often present between the data on different sensors in the network. Most of these algorithms require collecting a great deal of data before compressing which introduces an increase in latency that cannot be tolerated in real-time systems. We propose a distributed method for collaborative compression of correlated sensor data. The compression can be lossless or lossy with a parameter for maximum tolerable error. Error rate can be adjusted dynamically to increase compression under heavy load. Performance evaluations show comparable compression ratios to centralized methods and a decrease in latency and network bandwidth compared to some recent approaches.

1. 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, and 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. In addition, with the emergence of collaborative on-demand sensor applications [50], a need exists for efficient collaborative data 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 for wireless sensor networks have been proposed. Many emphasize low energy profile [42][43] to function in the constrained wireless environment. Others exploit the physical layout of the sensors [5][6], or the spatio-temporal correlation often present in the data to achieve better compression. GAMPS [9] effectively uses spatio-temporal correlation by grouping correlated sensors and using amplitude scaling to relate the streams of values from the correlated sensors, but is centralized and requires collecting all of the data before compression. The distributed ASTC approach [41] performs the compression in-network by building and merging clusters and cliques of related sensors. It gives good compression ratios, but generates additional peer-to-peer communication and heavier energy usage from the increased processing.

We propose a distributed collaborative method designed for real-time sensor networks such as those used in the sensor cloud [50]. Correlated sensors form groups and use amplitude scaling on their signals to express their sensed values in terms of other sensors in the group. The grouping and scaling is done in a distributed fashion in real time. This is similar to the method used in GAMPS[38] which employs a centralized algorithm on the data after it has all been collected; however, GAMPS provides no reduction in bandwidth or energy use on the sensors and is not designed for real-time systems.

If some loss in the accuracy of the data is tolerable, then the potential for compression increases greatly even for small loss. In our work here, we include a parameter for the maximum tolerable error for a single sensed value. For sensors with multiple inputs, the parameter can be set globally for all signals or individually for different error tolerance

for each type of sensed value. Setting any max error to 0% naturally achieves lossless compression. We provide in-depth analysis and discussion of different methods for measuring error and compare the compressibility and actual error for variations methods of utilizing the error tolerance.

We then compare the results of our approach to the existing spatio-temporal existing methods such as GAMPS [38] and ASTC [41]. We also compare our method to the single sensor TinyPack [28] and LEC [43] methods and compare our prediction methodology with PREMON [40] and a sensor network adaptation of Kalman Filters [39]. Experiment and simulation results show significant reduction in bandwidth, latency, and energy consumption compared to the other methods.

In summary, this paper makes the following contributions:

Novel algorithms for lossy collaborative compression in sensor networks with tunable maximum loss

Discussion and analysis of how to select and handle tolerable loss in the data

An ultra low-weight prediction mechanism

An analysis of several methods of grouping and clustering

Novel and effective error recovery techniques

2. RELATED WORK

2.1. GAMPS

A lossy multi-stream compressor is proposed in [38]. GAMPS compresses the data from multiple sensors which sense correlated data using mathematical techniques to groups 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 a threshold parameter to achieve compressed sizes under 1% of the original size.

For a single sensor, the series of values is scanned until the difference between the maximum and minimum exceeds twice the error threshold. The entire sequence (excluding the last one which caused the excess difference) is approximated as the average of the maximum and minimum. In this way the approximation never differs from the original by more than the error threshold. In order to keep the time windows consistent across all sensors in a group, the time slices are all reset when any sensor requires it.

A baseline sensor exists in each group. Linear regression models are used to find the closest linear function which maps each sensor to the baseline. Again, if the error exceeds the threshold a new function is found.

The actual grouping is dependent on the above processes. An initial time window is set and the groups are set for each time window using a heuristic solution to the Facility Location problem. Initially all the sensors are in one group. Then a base sensor is chosen at random and sensors are added to its group as long as the cost of adding them is less than the cost of starting a new group. After the groups are set for each time window, the time windows are tested to see if halving or doubling will increase the compressibility of the data.

This method is very effective but requires full centralized knowledge of all the data before compression is possible at all.

2.2. ASTC

In [41], a distributed, lossy, spatio-temporal approach is introduced. One-hop clusters comprised of correlated sensors are formed based on previous sensed values. A select number of the sensors in a cluster are chosen to form a master cluster on which temporal correlation is used to form a model. This model is sent to neighboring clusters, which can merge with the original cluster forming larger clusters.

Individual nodes which do not remain correlated to their respective clusters are evicted. These evicted nodes then listen to their neighboring clusters and can either join an existing cluster or form a new cluster depending on whether or not any of the neighboring clusters accept them.

2.3. PREMON

PREMON [40] uses an algorithm similar to that of MPEG and JPEG compression. Sensor correlation is computed as vectors to macro blocks which are used to build a model for the data. The sensors then only report deviations from the model. All the computation of the models is done in a centralized fashion at the sink and the models are transmitted back to the sensors. The model is periodically reconstructed and retransmitted to the sensor nodes.

2.4. LEC AND TINYPACK

A number of very lightweight compression codes are introduced in [43] and [28]. LEC consists of a set of delta compression codes based on JPEG compression and applied to sensor nodes. A similar set of codes is derived in TinyPack which is more highly tuned to the temporal correlation observed in many real life datasets. These codes

are used as the basis for the delta compression used in reporting the deltas from the baseline values in this work.

3. BACKGROUND

3.1. COLLABORATIVE COMPRESSION

Compression on a single sensor can often be achieved by exploiting temporal correlation in the data. In the single sensor TinyPack algorithms [28], each sensed value is compressed using the most recent previously sensed value as a baseline and expressing the value as a function of that baseline. In multi-sensor environments, neighboring sensors can be used as the baseline allowing for greater compression under the assumption that the values from the two sensors are correlated.

3.2. SPATIAL LOCALITY

Wireless sensor networks where multiple sensors are deployed over an area generally exhibit spatial locality (data from readings taken by sensors geographically near each other are correlated). Any type of data which changes in a continuous fashion across space will be temporally located such as temperature, humidity, location of tracked objects, light intensity, distance to a sensed event, etc. In fact, it can be demonstrated that any network deployed over a certain area will either generate spatially located data or random noise.

Consider an arbitrary sensor network sensing a set of values $\{v_1, v_2, \ldots, v_{2N}\}$ sensed at locations $\{x_1, x_2, \ldots, x_{2N}\}$ where N is an integer. Assume that the values are not correlated. Then placing sensors at locations $\{x_1, x_3, \ldots, x_{2N-1}\}$ and $\{x_2, x_4, \ldots, x_{2N}\}$ would yield completely different values. Thus, offsetting the sensor locations would generate

entirely different data. Therefore, excluding applications which generate pure noise, we can assume that readings at nearby sensors will be correlated.

Note that this does not apply to situations where the sensors are deployed individually on specific locations such as those placed on animals for location tracking. These applications do not necessarily exhibit spatial locality (although they may) and were not included in this study.

4. TOLERABLE ERROR AND PREDICTION

We consider a parameterized maximum tolerable error percentage E_{max}. Instead of reporting every value exactly as sensed, if a value deviates from some baseline less than E_{max}, the baseline value can be used instead. This allows for much greater compression while keeping the error bound by the tunable maximum. This parameter can be adjusted based on the application need, i.e., in real-time, but can tolerate some error (lossy), or non-lossy, but can tolerate some latency.

4.1. MEASURING ERROR

A common method of measuring error, E, between a reported value, V_R, and the actual value V_A, is the following formula.

$$E = \frac{|V_A - V_R|}{V_A}$$

Unfortunately, that measure is dependent on the units used. For example, if temperature is measured in Kelvins, degrees Celsius, or degrees Fahrenheit, the calculated error can vary greatly for the exact same data.

Consider a sensor which reported a temperature of 2°C when the actual temperature was 1°C. Table 24 shows the calculated error for the exact same data expressed using the

three most common temperature scales. The calculated error ranges from 0.365% to 100% for the exact same data.

Table 24 Inconsistent error measure

	Celsius	**Fahrenheit**	**Kelvin**
Actual	1	33.8	274.15
Reported	2	35.6	275.15
Calculated Error	100%	5.32%	0.365%

Even just within one scale the error can be misleading. If a sensor is measuring temperature and reporting the value in degrees Celsius, when the temperature is very close to 0 a small change in the value could cause a drastic increase in the error percentage. Also, when the actual value is 0, the error percentage is undefined.

In practice, the best way to set an upper bound for error would be to explicitly set the bounds in terms of the scale. For example, when set by the end user, the tolerable error for a temperature reading could be +/- 1°C. For analysis, however, it is useful to have a method of normalizing the error to a percentage. One method to do this would be to divide the difference by the maximum range of the sensor; however, since this range can be very large compared to the actual sensed range, the error percentages would be artificially low. For our analysis we use the maximum range of actual sensed values as the denominator for the error normalization

Table 25 Consistent error measure

	Celsius	**Fahrenheit**	**Kelvin**
Actual	1	33.8	274.15
Reported	2	35.6	275.15
Observed minimum	0	32	273.15
Observed maximum	40	104	313.15
Range	40	72	40
Calculated Error	2.5%	2.5%	2.5%

. Table 25 shows the calculated error for the same data assuming the temperatures measured range from 0 to 40 degrees Celsius and demonstrates that it is consistent across scales.

4.2. BASELINE SELECTION

Let *D* be the maximum value by which a particular sensed value can differ from the baseline in order to maintain an error percentage within the upper bound E_{max}. Any time a value differs from the baseline by more than *D*, a new baseline must be selected. The easiest approach would be to simply use the current sensed value as the new baseline; however, different characteristics of the various signals could afford better results for other methods.

We consider six different methods for selecting a new baseline and analyze the compression and actual error that result for varying maximum error. The first method simply selects the current value, *V*, as the new baseline. Next, if the data is assumed to increase or decrease steadily over time, then the new baseline could be set as $V+D$ (where *D* is negative when the values are decreasing). However, if the data has a general trend of increase or decrease but has small local fluctuations, the new baseline could be $V+D/2$. We also consider $V-D/2$ which penalizes rapid increase and decrease and performs better when the data trends back to the average. The last two methods utilize a *jumping baseline*, i.e. the current baseline is increased or decreased based on the previous baseline, *B*, not the current value. The reported value is always evenly divisible by the baseline jump width which is determined by the max tolerable error. They are denoted $B+D$ and $B+D/2$ and are similar to the second and third methods but are more

compressible since the number of possible baselines is lower (all will all be in the form of *initial_baseline* + *kD*/2 where *k* is an integer based on the max error).

The analysis was performed using a publicly available dataset from a study at an Intel Berkley laboratory [12]. The data contains over 13 million readings for temperature, relative humidity, light intensity, and voltage from 54 sensors deployed in the lab. Figure 38 shows the results comparing the baseline update messages needed as a percentage of the messages needed to send the data uncompressed.

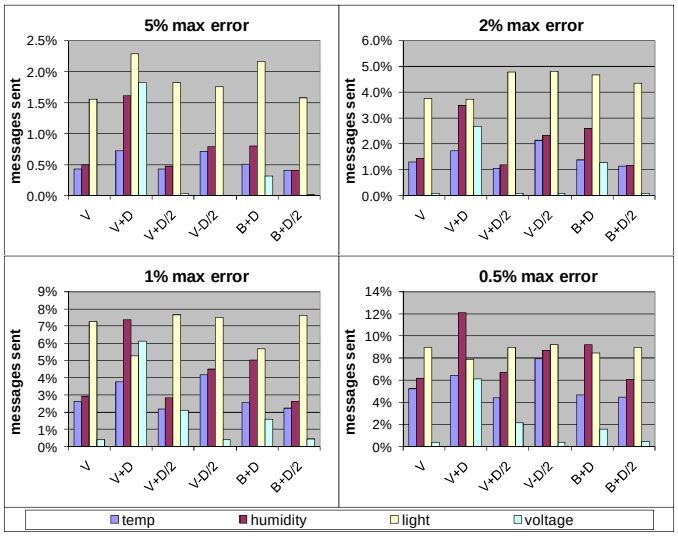

Figure 38 Messages sent on varying max error

Voltage generally exhibited minor fluctuations causing both of the *+D* methods to perform poorly. Both of the *B+* methods performed well compared to the others. Since they have additional compressibility, they are significantly more effective for compression.

We also computed the actual error generated by each method over the same datasets by comparing the compressed values with the original values. Results are shown in Figure 39.

Light intensity had the lowest actual error for the *V* method since in the dataset it regularly experienced large changes and then remained very consistent for long periods.

The jumping baselines were at or near the minimum for all the experiments. Additionally, the jumping baseline methods provide additional compressibility due to the increased frequency of the baseline values.

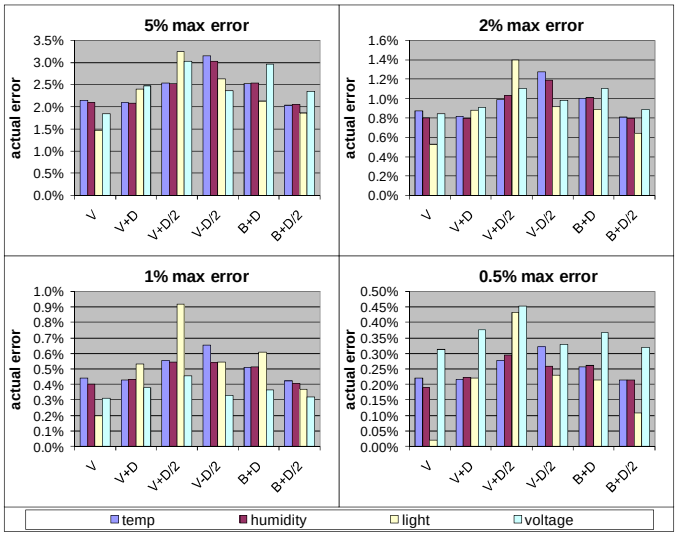

Figure 39 Actual error on varying max error

4.3. BASELINE COMPRESSION

We extend the benefit of jumping baselines for compression by implementing a simple prediction mechanism. A data stream can be in one of three states: trending up, trending down, or staying somewhat constant. If data is trending either up or down, then the next baseline should be selected as far in the direction the data is trending as it can be within the error bounds. If the data is remaining relatively constant, then the next baseline should be selected as close to the current value as possible. We determine the state by tracking whether the new baseline is above or below the previous baseline for two jumps. If both jumps were in the same direction, the data is trending either up or down depending on the direction of the jumps. The prediction only requires caching the

previous value and the previous jump direction. The additional computation is also trivial.

Table 26 Prediction example

Seq no	Sensed value	Last value	Last jump	This jump	Baseline
1	242	237	--	--	240
2	253	242	--	up	250
3	261	253	up	up	270
4	276	261	up	--	270
5	284	261	up	up	290

For example, Table 26 shows an example of a light sensor with a maximum error set at +/- 10 lux.

Algorithm 1 CheckReading(*v*, *p*, *S*, *d*)

```
Objective: Check the current reading and select a new
           baseline if needed
Input:  Sensed value v, previous value p, max variance S,
        previous jump direction d
Output: Reported value r
    If |p – v| > S
        r := NearestBaselineTo(v)
        If v > p And d == UP
            r := r + S/2
        Else if v < p And d == DOWN
            r := r – S/2
        End If
        If v > p
            d := UP
        Else
            d := DOWN
        End If
        p := r
    Else
        r := p
    End If
```

Initially, the baseline is selected as close as possible to the actual sensed value. When the upward trend is established at sequence number 3, the baseline is selected as high as possible while remaining within the error tolerance of +/- 10. Then as the data continues

to trend upward, the baseline does not require as many jumps while never exceeding the maximum tolerable error. This process is shown in detail in Algorithm 1.

4.4. ENTROPY RESULTS

The total amount of bytes needed to transmit a stream of data can be measured by the entropy of the dataset. Assuming no additional prediction methods are used for a data stream, the entropy of the data (as defined in [4]) provides a measure of the minimum number of bits that would be required to transmit the data if some theoretical optimal compression was used. Thus, entropy is an effective means of calculating the total "compressibility" of a stream of data. Assuming no predictions or other transformations are used, the theoretical minimum number of bits required to transmit a value can calculated with the following formula, where P is the probability of that value appearing in the data stream (count of that value divided by total messages in the stream):

$$bits = \log_2\left(\frac{1}{P}\right)$$

We used entropy to measure the effectiveness of the jumping baseline compression and prediction and compared the results to other prediction methods. PREMON [40] is an MPEG based prediction algorithm designed specifically for sensor networks. Kalman Filters are also commonly used to predict data streams. We compared against a Kalman filtering scheme which has been adapted for sensor networks [39]. PREMON and Kalman filters perform sophisticated prediction, reducing the number of messages that need to be sent while the jumping baseline method can afford higher compressibility. We also included the simplistic approach of merely rounding the data to the nearest baseline since that gives a similar reduction in entropy.

PREMON and rounding were configured to use the same maximum tolerable error and the Kalman Filters (which are not bounded on error) were configured to have the same total calculated error as the jumping baseline method.

Total number of messages sent as a percentage of the total number of messages in the original data for the Intel Labs dataset is shown in Figure 40. The entropy of the transformed data as a percentage of the original entropy for the same data is shown in Figure 41.

As expected, Kalman filters and PREMON required fewer messages to be sent due to more accurate prediction, but since the size of the messages would need to be higher, the jumping baselines performed best in terms of overall entropy. Thus compression will be more effective using the jumping baselines over the other methods.

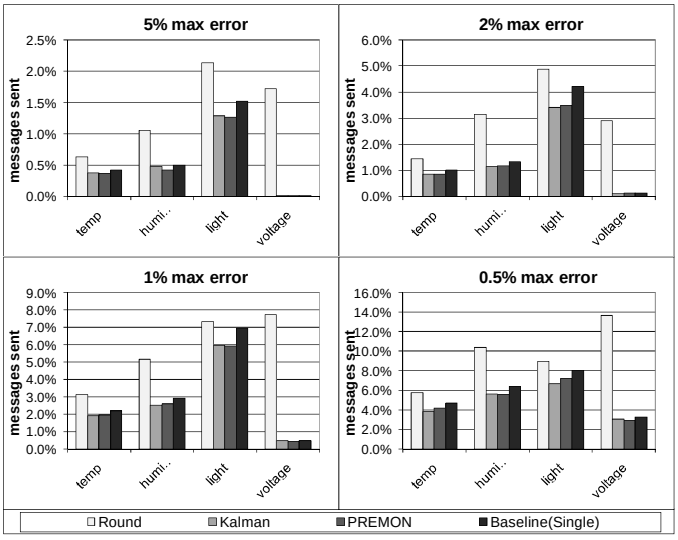

Figure 40 Messages sent on varying max error for different prediction algorithms

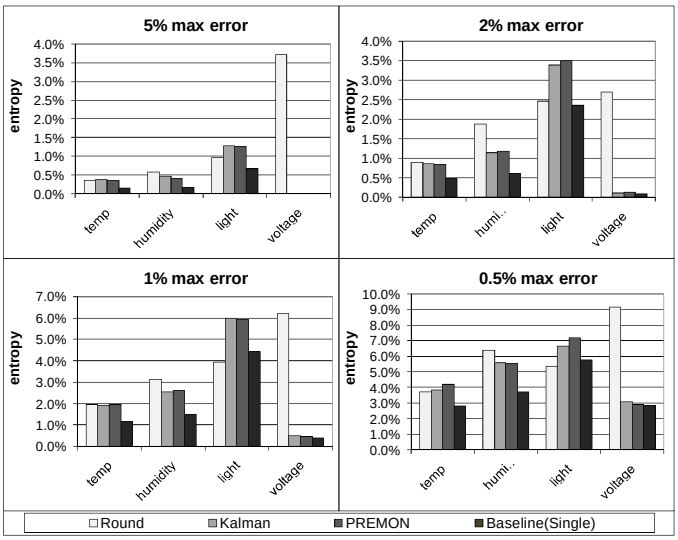

Figure 41 Entropy on varying max error for different prediction algorithms

5. COLLABORATION

5.1. CORRELATION

Collaboration between the sensors can then be used to further enhance the compression of the entire dataset. Correlated sensors can transmit the count of jumps in which their baselines differ. The sensor chosen as the base sensor serves as a parent node in the correlation tree. Then the child node can report its values using its offset from the parent sensor's baseline as its baseline. The algorithm used is identical to Algorithm 1 except the total count of baseline jumps is reported as an offset of the other sensor instead of an absolute.

For example, consider two light sensors where sensor S_2 is reporting its values based on sensor S_1. Assume again the maximum error is +/- 10 lux. Table 27 shows a sample data stream for the two sensors including the actual sensed values, the message sent, and the final reported value as interpreted at the sink.

Table 27 Collaboration example

Seq no	S_1 sensed	S_2 sensed	S_1 sent	S_2 sent	S_1 final	S_2 final
1	237	259	+24	+2	240	260
2	242	266			240	260
3	253	271	+1		250	270
4	261	278	+2	-1	270	280
5	275	282			270	280

At the first sensed values, the sensors have no baselines, so S_1 uses 0 as its baseline and S_2 uses S_1's initial value as its baseline. In the message at sequence number 3, S_2 would have needed to transmit a jump message if it were reporting its own values, but since S_1 reported a jump, S_2's interpreted value automatically jumped. Two noteworthy things happened at sequence number 4. The prediction detected the upward trend in S_1's data and selected the highest baseline within the tolerable error, and S_2 corrected its offset from S_1's baseline.

5.2. CODES

The codes used for transmitting the compressed baseline jumps for individual or correlated sensors are drawn from those used in [28]. An example set of codes for the delta values of -127 to +127 is shown in Table 28.

Table 28 Delta codes

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

For example, a change value of +3 would be transmitted as 00101 and -3 would be transmitted as 00111. The pattern can continue for values as high as are needed. If the maximum value is known, the last level need not have a 1 at the end of the prefix.

These codes can be used to both encode and decode very efficiently with minimal processor utilization. The value expressed by a code can be computed by the following equation where *B* is the number of 0 bits before a 1, *S* is the first bit of the suffix (sign bit) and *k* is the number represented by the remaining suffix bits interpreted as an integer:

$$(-1)^S \left(2^B + k\right)$$

For example, the value -14 would be represented by 0001 1 110 where prefix=0001 (thus $B = 3$ and $2^B = 8$), $S = 0$, and $k = 110 = 6$. So (-1)(8+6) = -14.

5.3. MESSAGES

There are only two message types sent by the sensors: baseline jumps, and parent sensor changes (rebellions). Since these rebel messages are expected to be infrequent compared to the baseline jumps, it would be inefficient to assign an entire bit to distinguish between the message types. Instead a value is selected from the table to use as the indicator and all the other values are shifted down one. For our experiments, we used -15. So if a value started with 00011111, it is interpreted as a rebel message and the rest of the bits contain the new parent node ID. Then an actual -15 message would be encoded like -16 and so on. Node IDs are compressed by using the minimum number of bits needed for the total number of nodes. For example, if there were 33 to 64 nodes deployed, the IDs would use 6 bits.

Another small gain can be obtained by shifting past known invalid values. For example, if a data stream is trending up (using the prediction method), +1 is an invalid

jump since it would jump by at least +2. So any positive change automatically has another +1 added to it. This often had only a slight benefit but for data streams that steadily increase or decrease over a long period saw an additional 20-30% drop in the compressed size.

5.4. GROUPING

Not all sensors in a network are necessarily correlated and the values from sensors that are correlated may not be equal. Distinct groups of sensors which exhibit higher correlation tend to emerge and the values at one sensor can often be more efficiently transmitted as a difference from another sensor's values.

We compare using two very simple and lightweight grouping mechanisms: sink side and node side.

The sink side approach assumes that the sink is not another sensor node and does not have the same energy and processing constraints. It also assumes that the sink can communicate back to the sensors. The node side method makes no assumptions.

In the sink side algorithm, the sink performs the facility location computations as done in [38] over a window of the recent data and reports back to the nodes the ideal parent node for that window.

In the node side algorithm, the nodes maintain an array indexed by other node IDs with two entries. The first entry contains the current baseline jump distance from that node and the second entry contains the number of times the first entry has changed. Every time a node would need to send a jump message from its current parent, it finds the minimum jumps in the array and selects that node as its new parent. If two nodes select

each other as the parent, the tie is broken by node ID and the node with the lower ID is selected as the parent.

If a node's parent node selects a parent, the node does not need to select a new parent. It merely calculates the value of its parent based on the reported value from the grandparent node. If the grandparent node is not in radio range however, the node will need to select a new parent.

6. RESULTS

6.1. BANDWIDTH

Results for total bandwidth requirements are shown in Figure 42. We compared results between our baseline compression on single sensor, the GAMPS algorithm, ASTC, and our collaborative compression approach. The sink side algorithm performed almost identically to the node side algorithm but slightly worse due to the increased amount of messages sent and is not included in the graphs.

Bandwidth is shown as a percentage of the bandwidth required to send the data uncompressed. We assumed uncompressed data would be transmitted with the minimum number of bytes required to cover the observed range of possible values. Voltage only required one byte to send uncompressed while temperature, humidity, and light intensity required two bytes for each sensed value.

Collaborative baseline compression performed best in terms of required bandwidth compared to the other approaches for all data types studied except for voltage. The single sensor baseline compression performed best for the voltage because voltage is included in the dataset as a data integrity check and is not expected to be correlated between neighboring nodes.

Voltage also had higher variance in a short time interval but did not change drastically over time which accounts for the greater variance in results for voltage across the different tolerable error rates.

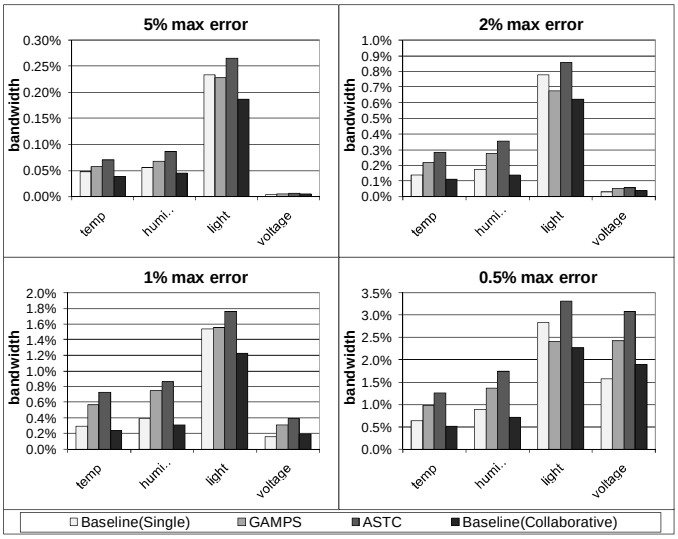

Figure 42 Bandwidth utilization on varying max error for different compression algorithms

6.2. LATENCY

Latency was measured in a network of TelosB motes 0 loaded with the data from the Intel Labs experiment and configured to send data to the sink based on the timestamps in the dataset.

Figure 43 shows the latency results for the collaborative baseline compression and comparative methods. Results show time required to process the data, transmit the data, and any time required to wait to send the data.

For comparison, GAMPS was modified to send data as soon as enough had been collected to perform the compression. ASTC incurred some wait time as the nodes communicated to build the prediction model. The nodes were not synchronized for the dataset, so for the jumping baseline, a correlated sensor reporting its value from a base

sensor would occasionally need to delay sending its offset until the base sensor had sent its value.

Again the results shown are totals over the entire dataset for temperature, humidity, light, and voltage values.

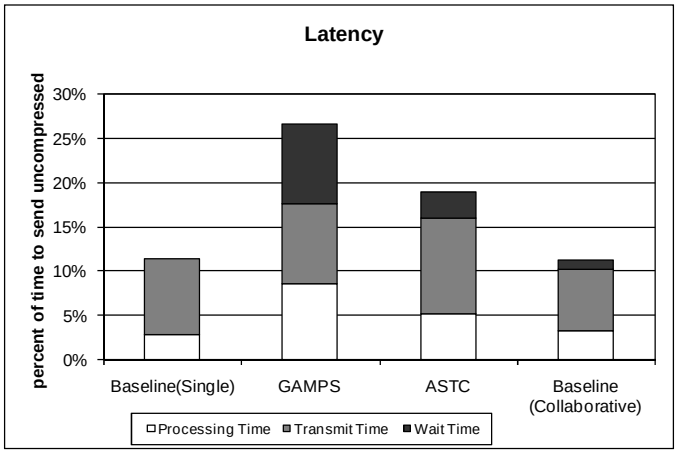

Figure 43 Total latency for single hop network for different compression algorithms

Tolerable error only affected the transmit time. Results are shown for 5% max error for better clarity since at lower errors, the latency for processing would be difficult to see. The transmit time is a simple function of the compressed size of the data. At 5% max error, our collaborative baseline approach performed the best in terms of latency. As the tolerable error decreased, our single sensor baseline method had the least latency.

Latency results shown are for a single hop network. As the number of hops increases, the total latency at each hop approaches the latency of the transmit time since no additional processing or wait time would be required. Since the collaborative baseline algorithm provided the best compression ratio, it performs better compared to the other algorithms as the number of hops between the sensing node and the final sink increases.

6.3. ENERGY USAGE

A network of MicaZ motes [15] running TinyOS was simulated in TOSSIM [17]. Energy consumption was modeled using PowerTOSSIM [18] which provides a layer of energy usage tools on top of the sensor simulation tools provided in TOSSIM. Figure 44 shows the average energy per sensor required to compress the data for each of the algorithms. The energy required to transmit the data is a directly proportional to the compressed size of the data. Energy usage results for transmitting the data are not shown since they would be proportionally identical to the bandwidth results.

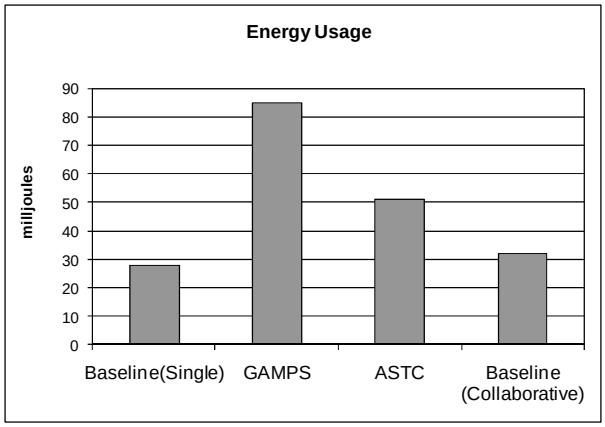

Figure 44 Energy usage due to processing for different compression algorithms

The MicaZ mote has three different radio power settings that require 11, 14, and 17.4 mA respectively while transmitting. The MicaZ processor uses 8 mA in active mode [15]. The total energy required is dependent on the radio power setting. Since total energy consumption is based on current and time, the total energy results are proportional to the latency results for processing and transmission in Figure 44 except the transmission energy scales to 11/8, 14/8, or 17/8 of the transmission time based on the radio power used.

There was no appreciable difference for processing between the different types of data in dataset thus energy results are shown as totals over the entire dataset. Maximum error also did not have a significant impact on processing requirements. Results shown are the average of the four simulations.

The simplicity of the jumping baseline approach gives it a much lower processing profile than the other methods. GAMPS was not designed to be energy efficient and as a result did not perform well. Baseline compression on a single sensor naturally performed better than the collaborative approach since the collaboration uses the single sensor method as its initial baseline.

7. ERROR RECOVERY

7.1. OUTLIERS

If a signal contains outliers, the compression can suffer since the baseline will change to report the outlier and change back on the following packet. If some latency is tolerable in the system, the sensor can wait to report the change in the baseline until it has sampled a few more values to confirm if the change in the baseline is due to an outlier in the data.

We defined an outlier detection window of size *W*. The readings in a window are considered outliers if they satisfy the following two conditions:

The readings immediately preceding and following the window are the same value

The readings in the window differ from those immediately preceding and following the window by more than one baseline jump

It other words, if a sensed stream briefly reports a drastic change in value and then returns to the previous value, that change is likely to be an error and those readings are considered outliers. We performed simulations for window sizes of 1, 2, and 3. For

window sizes greater than 1, any value that would be considered an outlier using a smaller window size is still considered an outlier. Results are shown in Figure 45.

Manual inspection of the data revealed some clear outliers where a temperature reading or other value type would drop to 0 for a single sensed value and otherwise remain fairly constant.

Naturally, false positives could occur if a sensed stream rose above or fell below the current baseline beyond the error threshold for a brief moment and then returned; however, the reported value would still be very close to within the tolerable error band and the total error of the compressed stream would not be significantly impacted.

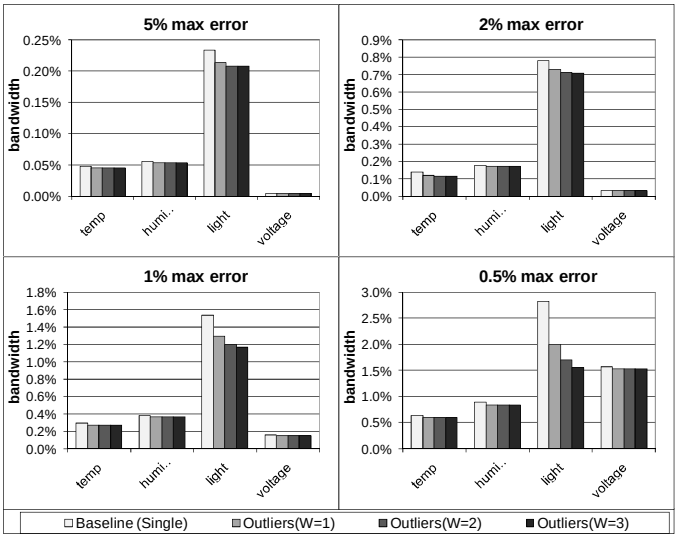

Figure 45 Bandwith savings with outlier detection

There were not many outliers detected in the dataset; however, on average, for a window size of 1, outliers comprised 0.11% of the data stream but required 7.4% of the bandwidth. Thus, detecting outliers in this way can significantly reduce the bandwidth required to send the data especially if the number of outliers is high.

Most of the outliers in the dataset were single values so increasing the window size above 1 did not cause more outliers to be found in all cases except for light intensity. The

lights used in the experiment were fluorescent lights which produce a flickering affect. This flickering caused brief significant changes in the datastream that were calculated to be outliers. The question of whether or not such flickering should really be treated as outliers should be determined based on the goals of the individual experiment. The datasets studied contained few outliers but the outliers consumed a significant amount of bandwidth compared to their frequency.

7.2. SIGNAL RECONSTRUCTION

The actual error present in the compressed stream can be reduced by using the compressed data to approximate the original data through curve smoothing techniques. Since the actual error is bounded by a maximum tolerable error *E*, the range of possible true values that produces the compressed stream is known. This can be used to aid the curve smoothing process and generate a more accurate reconstruction of the original data stream.

If the real data changes slowly and smoothly, this can provide a dramatic decrease in the actual error of the reported stream; however, if the data is highly varied within the bands, then attempts to reconstruct the original stream can actually add more error. The maximum added error is known, however, since it can be no more than twice the configured maximum tolerable error (assuming the reconstruction is designed to remain within *E* of the reported value from the compressed stream).

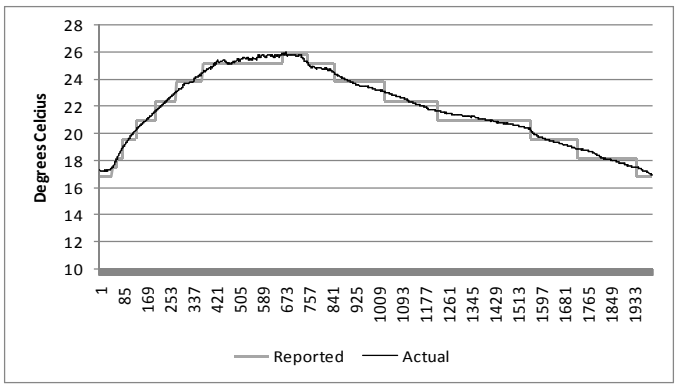

Figure 46 Reported vs. Actual temperature for 2% max error

Figure 46 shows 2000 readings from a temperature sensor compressed using jumping baseline algorithm. The compressed and actual values are shown.

Due to the unique nature of the jumping baseline algorithm, when the baseline changes the true value at that point can be accurately reconstructed. When data is trending up or down and the baseline jumps, the true value at the point of the jump will be nearly equal to the average of the two baselines. (If the sample interval was infinitely small, it would be exactly equal). When the data stream is peaking or oscillating (neither trending up nor down) the true value at a baseline jump can be accurately approximated by the value of the new baseline. Since the data trend is known, this can be used to design a very simple signal reconstruction algorithm that can greatly reduce the total error in the stream.

The reconstructed stream is build by first approximating the values at the points where the baseline jumped. Then any curve fitting algorithm can be used to fit a curve to those points to create the fully reconstructed stream. For our testing, we simply approximate the curve by assuming the data between the points is linear. Figure 47 shows the same data as Figure 46 but with the reconstructed stream added.

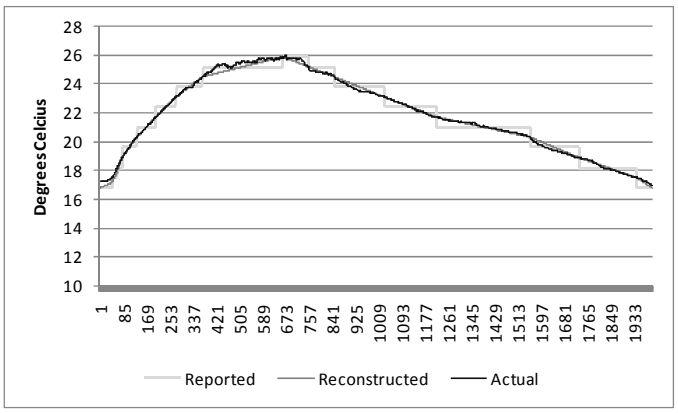

Figure 47 Reconstructed stream

We computed the actual error both with and without signal reconstruction for different configured max tolerable errors over the entire dataset. Temperature, humidity, and light intensity all were very similar. Signal reconstruction reduced the measured average error to approximately 1/6 of the max tolerable error. Aggregated results are shown in Table 29. Voltage streams were not as continuous as the other three and signal reconstruction was not as effective. The actual error of the voltage streams after reconstruction was approximately 1/3 of the max tolerable error for each configured maximum used in the experiments. Voltage results are shown in Table 30.

Table 29 Error (temperature, humidity, light)

Max tolerable error	**Baseline error**	**Reconstructed error**
5%	2.47%	0.832%
2%	0.964%	0.323%
1%	0.483%	0.167%
0.5%	0.239%	0.0815%

Table 30 Error (voltage)

Max tolerable error	Baseline error	Reconstructed error
5%	2.56%	1.38%
2%	1.06%	0.692%
1%	0.519%	0.387%
0.5%	0.252%	0.193%

8. CONCLUSIONS AND FUTURE WORK

The jumping baseline method provides a very light weight collaborative compression scheme for wireless sensor networks. Energy and processing usage were well below those of existing algorithms while maintaining lower latency and requiring less bandwidth.

Compression could be improved even further in the future by taking advantage of correlations, not only between neighboring sensors, but also between different streams on the same sensor. For example, temperature and light were somewhat proportional in the dataset and were inversely proportional to humidity.

Since signal reconstruction could be done on the sink side, much more sophisticated algorithms could be used to fit a curve to the values approximated at the jump points.

V. TOWARD ENERGY EFFICIENT MULTISTREAM COLLABORATIVE COMPRESSION IN WIRELESS SENSOR NETWORKS

Wireless sensor networks possess significant limitations in storage, bandwidth, and power. This has led to the development of several compression algorithms designed for sensor networks. Many of these methods exploit the correlation often present between the data on different sensor nodes in the network; however, correlation can also exist between different sensing modules on the same sensor node. Exploiting this correlation can improve compression ratios and reduce energy consumption without the cost of increased traffic in the network. We investigate and analyze approaches for compression utilizing collaboration between separate sensing modules on the same sensor node. The compression can be lossless or lossy with a parameter for maximum tolerable error. Performance evaluations over real world sensor data show increased energy efficiency and bandwidth utilization with a decrease in latency compared to some recent approaches for both lossless and loss tolerant compression.

1. INTRODUCTION

Wireless sensors are used to collect and transmit data in a wide variety of applications. Many such applications utilize sensor nodes that collect several different streams of data on different sensing modules on the same sensor node. For example, sensor nodes in the Great Duck Island project [51] and an Intel Berkley Labs experiment [52] were used to collect temperature, humidity, light intensity, and more. Even applications that primary just sense one thing often send multiple streams of data from the same sensor. For example, ZebraNet [53] tracked locations of zebras sending two

streams of data for the GPS readings (easting and northing) and some metadata such as voltage and count of satellites in range of the GPS sensor.

It is well known that wireless sensor networks possess significant limitations in processing, storage, bandwidth, and power. This has, naturally, led to the development of many compression algorithms specific to sensor networks. Many of these algorithms rely on the data readings from a single sensor being correlated to previous readings on that same sensor (temporal locality) [42][43][28]. Others rely on correlations between similar data streams on other sensor nodes (spatial locality) [38][58][59][41]. Little work has yet been done, however, which directly exploits the correlation that is often present between different streams of data collected on the same sensor node.

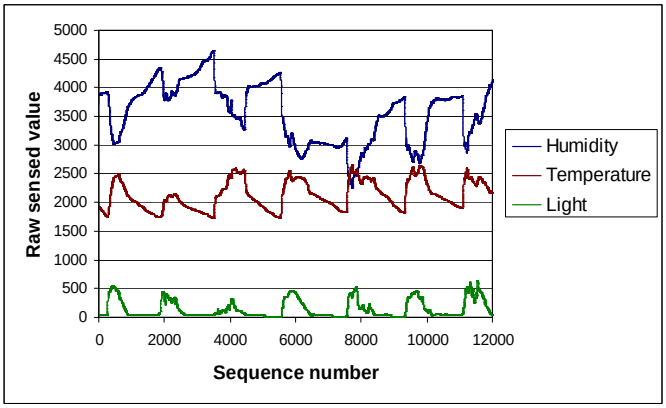

Figure 48 Multistream sensor readings

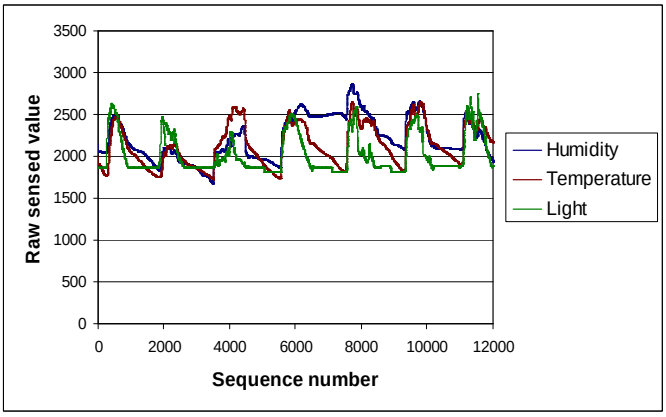

Figure 49 Scaled multistream sensor readings

To illustrate this correlation, Figure 48 shows values from 12,000 readings of temperature, humidity, and light intensity sensors on a single sensing node taken from the Intel Lab dataset. Figure 49 shows those same values scaled with the simple linear transformations shown in equation 1 where h_n is the nth humidity reading and h_n' is the scaled value. Similarly, t_n and l_n are for the temperature and light intensity, respectively along with their scaled notation. Clearly some benefits could be gained by leveraging the correlation between the different data streams.

$$\begin{aligned} h_n' &= 4000 - 0.5h_n \\ t_n' &= t_n \\ l_n' &= 1800 + 1.5l_n \end{aligned} \qquad (1)$$

In this paper, we present TinyPack-Collaborative (TinyPack-C), a lightweight compression algorithm leveraging the temporal correlation within each stream and the correlation between multiple streams of data on an individual sensing node. TinyPack-C is based on the initial code set presented in [28] and extended to include collaboration between the multiple streams from the various sensors on the same sensing node. Collaboration is computed based on a rolling linear regression scheme requiring constant time memory use and processing for each correlated pair of sensed values.

If some loss is tolerable in the data, compression is enhanced by first performing a modified version of the jumping baseline transformation introduced in [61] which converts the stream into a step function. The rolling linear regression is then applied to the flattened streams. The maximum tolerable error can be configured low for simply removing noise from the data or high if the application is not concerned with low variation in the data.

We present and analyze compression schemes for both lossless compression and loss tolerant compression with a configurable maximum error. We compare both varieties against state of the art compression methods. For the lossless case, we compare against the original TinyPack algorithm, LEC [43] and S-LEC[62]. We compare our lossy compressor with LTC [63] and the single sensor jumping baseline approach [61]. Simulations using TOSSIM [17] were done over several real life datasets covering a wide variety of sensor applications.

In summary, this paper makes the following contributions:

Novel algorithms for lossless compression leveraging collaboration across multiple streams on a single sensor node

Additional algorithms for lossy compression with a configurable upper bound for error

Lightweight mechanisms for computing correlation

Detailed analysis over several real world datasets

Methods for performing mathematical operations and aggregation on the compressed data without first decompressing the data

2. RELATED WORK

2.1. S-LEC

S-LEC, a lossless data compression scheme, is proposed in [62]. S-LEC begins with the static set of codes used in LEC [43] to represent delta values in a data stream. In LEC, each reading, the previous value is subtracted from the current value and the resulting delta value is coded based on a static table of codes derived from those used in JPEG compression. Smaller delta values have shorter codes. For S-LEC, codes that are the

same length are said to be in the same group and two bits are prepended to each value noting whether the current delta value is in the same, one higher, one lower, or any other group as the previous delta value. This enables reducing the size of the prefix come and improves the compression ratio when data is changing in a consistent fashion.

2.2. TINYPACK

Another lossless method is presented in [28], TinyPack initially uses a similar set of static codes for its compression, but the codes were optimized for wireless sensor data instead of JPEGs. Those codes are then dynamically modified either by counting the frequency of each value or by approximating those frequencies using a rolling average and standard deviation. The initial set of codes used in TinyPack-Init is shown in Table 31 and forms the basis on which the compression in this work is built.

Table 31 Static codes

prefix	suffix range	values
1	n/a	0
01	0...1	-1.1
001	00...11	-3,-2,2,3
0001	000...111	-7,...,-4,4,...,7
00001	0000...1111	-15,...,-8,8,...,15
000001	00000...11111	-31,...,-16,16,...,31
0000001	000000...111111	-62,...,-32,32,...,63
00000001	0000000...1111111	-127,...,-64,64,...,127

Except in the case of 0, the last bit of the suffix is the sign bit. For example, if the current reading was 3 higher than the previous reading, a delta value of +3 would be transmitted as 00110. A delta value of -4 would be encoded as 0001001.

2.3. LTC

In [63] a lossy compression scheme is introduced that approximates the data stream by a sequence of linear segments. As the data is collected by the sensor, the algorithm fits a line to the data as long as the line can be defined such that no point in the transformed data exceeds a maximum error bound. When a data point is sensed that cannot be fit to the line without exceeding the allowed error, that line is transmitted and a new line starts. The algorithm is effective but does introduce additional latency since the data is not transmitted until the sensed reading that necessitates a new line.

2.4. JUMPING BASELINES

The jumping baseline approach in [61] approximates the data stream as a discrete step function which can be reconstructed to a linear function similar to the one generated by LTC at the sink. Any time a sensed value is outside the maximum tolerable error away from the current baseline, a new baseline is selected. The possible candidate baselines are selected from multiples of the maximum error such that the new value can be expressed as the number of baseline jumps above or below the previous baseline. The new baseline is also selected as far in the direction the data has been trending as possible without violating the maximum tolerable error. This process is described in more detail in section 0 and forms the basis on which our lossy compression is built.

3. BACKGROUND

3.1. TEMPORAL LOCALITY

Data from wireless sensor networks generally exhibits temporal locality (data values from the same stream are correlated to values that are close together in time). Any type of

data stream which changes in a continuous fashion will be temporally located such as humidity, position, light intensity, water level, etc. In fact, it can be demonstrated that any sensor stream sampled 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 that 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. Thus, offsetting the sample period would generate entirely different data. Therefore, application with time-based sampling which did not exhibit temporal locality must be sampling random noise. Excluding such applications 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.

Naturally 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.

The previously sensed value in each sensed stream can then be used as a baseline for compressing the value of the next sample in the stream. For lossless compression, the value can be transmitted as the difference between the current sensed value and the previous value (the *baseline value*). For lossy compression, the data can be approximated using the baseline value until the current value differs from the baseline value by more than the upper limit for tolerated error.

3.2. COLLABORATIVE COMPRESSION

In the case of collaborative compression, one sensed stream serves as the baseline for one or more of the other sensed streams on the same sensor. The data from this *baseline stream* is compressed leveraging temporal locality as discussed in the previous section and the data from the correlated streams are encoded based on the difference from some linear function of the baseline stream referred to as the *baseline function*. As with the single stream compression of the baseline stream, the lossless case would require that a delta value be sent every time the sensor samples data while the lossy case can use the baseline function as the approximated values for the compressed stream until the value is above or below the baseline function by more than the maximum tolerable error. The algorithm is shown in more detail section 0.

3.3. MEASURING ERROR

For the lossy compression, we consider a parameterized maximum tolerable error percentage E_{max}. Instead of reporting every value exactly as sensed, if a value deviates from its baseline less than E_{max}, the baseline value can be used instead. This allows for much greater compression while keeping the error bound by the tunable maximum. This parameter can be adjusted based on the application need, i.e., in real-time, but can tolerate some error (lossy), or non-lossy, but can tolerate some latency.

A common method of measuring error, E, between a reported value, V_R, and the actual value V_A, is shown in Equation 2.

$$E = \frac{|V_A - V_R|}{|V_A|} \qquad (2)$$

Unfortunately, that measure is dependent on the units used. For example, if temperature is measured in Kelvins, degrees Celsius, or degrees Fahrenheit, the calculated error can vary greatly for the exact same data.

Consider a sensor which reported a temperature of 2°C when the actual temperature was 1°C. Table 32 shows the calculated error for the exact same data expressed using the three most common temperature scales. The calculated error ranges from 0.365% to 100% for the exact same data.

Table 32 Inconsistent error measure

	Celsius	**Fahrenheit**	**Kelvin**
Actual	1	33.8	274.15
Reported	2	35.6	275.15
Calculated Error	100%	5.32%	0.365%

Even just within one scale the error can be misleading. If a sensor is measuring temperature and reporting the value in degrees Celsius, when the temperature is very close to 0 a small change in the value could cause a drastic increase in the error percentage. Also, when the actual value is 0, the error percentage is undefined.

In practice, the best way to set an upper bound for error would be to explicitly set the bounds in terms of the scale. For example, when set by the end user, the tolerable error for a temperature reading could be +/- 1°C. For analysis, however, it is useful to have a method of normalizing the error to a percentage. One method to do this would be to divide the difference by the maximum range of the sensor; however, since this range can be very large compared to the actual sensed range, the error percentages would be artificially low. For our analysis we use the maximum range of actual sensed values as the denominator for the error normalization (see Equation 3).

$$E = \frac{|V_A - V_R|}{V_{MAX} - V_{MIN}} \qquad (3)$$

Table 33 shows the calculated error for the same data assuming the temperatures measured range from 0 to 40 degrees Celsius and demonstrates that it is consistent across scales.

Table 33 Consistent error measure

	Celsius	**Fahrenheit**	**Kelvin**
Actual	1	33.8	274.15
Reported	2	35.6	275.15
Observed minimum	0	32	273.15
Observed maximum	40	104	313.15
Range	40	72	40
Calculated Error	2.5%	2.5%	2.5%

3.4. JUMPING BASELINE COMPRESSION

For our lossy compression algorithm, we begin with the jumping baseline compression introduced in [61]. The values in the stream are compressed to a step function by choosing a baseline value for a sensed value and only changing the baseline when the current sensed value differs from the baseline by more than the maximum tolerable error. The values selected as baselines are in the form *kE* where *k* is any integer and *E* is the maximum integer error that can be tolerated in a stream while remaining within the maximum error percentage E_{max}.

The initial baseline is selected by choosing the candidate baseline closest to the first value sensed in a stream. So for a sensed value *v* the baseline *B* would be selected as shown in equation 3. Adding 0.5 and truncating with the floor function is done as an efficient method of rounding.

$$k = \left\lfloor \frac{v}{E} + 0.5 \right\rfloor \qquad (3)$$
$$b = kE$$

When a sensed value differs from the current baseline by more than *E*, a new baseline must be selected. Note that there will be two candidate baselines that would be within *E* of the new value. The algorithm chooses the baseline based on which direction the data is trending. A data stream can be in one of three states: trending up, trending down, or staying somewhat constant. If data is trending either up or down, then the next baseline should be selected as far in the direction the data is trending as it can be within the error bounds. If the data is remaining relatively constant, then the next baseline should be selected as close to the current value as possible. The state is determined by tracking whether the new baseline is above or below the previous baseline for two jumps. If both jumps were in the same direction, the data is trending either up or down depending on the direction of the jumps. All that needs to be cached is the previous value and the previous jump direction. The additional computation is also trivial. For example, Table 34 shows an example of a light sensor with a maximum error set at +/- 10 lux.

Table 34 Baseline compression example

Seq no	Sensed value	Last value	Last jump	This jump	Baseline
1	242	--	--	--	240
2	253	242	--	up	250
3	261	253	up	up	270
4	276	261	up	--	270
5	284	261	up	up	290

Initially, the baseline is selected as close as possible to the actual sensed value. When the upward trend is established at sequence number 3, the baseline is selected as high as possible while remaining within the error tolerance of +/- 10. Then as the data continues

to trend upward, the baseline does not require as many jumps while remaining within the maximum tolerable error. This process is shown in detail in Algorithm 1.

Algorithm 2 CheckReading(*v*, *p*, *S*, *d*)

```
Objective: Check current reading, select next baseline
Input:  Sensed value v, previous baseline B, max difference E,
        previous jump direction d
Output: New baseline (reported value) B
    If |p – v| > E
        B := floor(v/E + 0.5)
        If v > B And d == UP
            B := B + E
        Else if v < r And d == DOWN
            B := B – E
        End If
        If v > p
            d := UP
        Else
            d := DOWN
        End If
        p := B
    Else
        B := p
    End If
```

4. OUR MULTISTREAM COMPRESSION APPROACH

4.1. ROLLING CORRELATION

A common simple method of approximating one data stream with another is to use a linear least squares approximation. The first stream is translated using a linear function in the form $Y = aX + b$ into an approximation of the second stream in such a way as to minimize the amount of error between the approximated stream and the actual stream. Computing full least squares regression is far too computationally complex to run on a sensor every time a new value is sensed; however, the correlation can be computed incrementally such that only a few calculations need to be made after each sample while still maintaining accurate correlation values.

Also, the correlation is not necessarily the same for the entire run of the sensor network so some decay should be introduced in the correlation equation such that the most recent data contributes a higher weight to the correlation and older data contributes less. Such decaying rolling statistics have been used many times for other applications [28][64][65]. Here we refine the rolling least squares to optimize for simplicity of calculation for the sensor networks.

A common method for calculating the slope and intercept of the regression line (correlation function) $Y = aX+b$ is shown in equation 4 where σ_X is the standard deviation of X, $E(X)$ is the expected value (mean) of X, and r is the Pearson Correlation of X and Y.

$$b = r\frac{\sigma_Y}{\sigma_X} \qquad a = E(Y) - bE(X) \qquad (4)$$

The standard deviation of a variable can be expressed in terms of the expected values of the variable and the square of the variable as shown in equation 5.

$$\sigma_X = \sqrt{E(X^2) - (E(X)^2)} \quad (5)$$

The Pearson Correlation coefficient is also commonly expressed in those terms as shown in equation 6.

$$r = \frac{E(XY) - E(X)E(Y)}{\sigma_X \sigma_Y} \quad (6)$$

Combining, equations 4, 5, and 6 we can derive equation 7.

$$
\begin{aligned}
b &= \frac{E(XY) - E(X)E(Y)}{\sigma_X \sigma_Y} \frac{\sigma_Y}{\sigma_X} \\
&= \frac{E(XY) - E(X)E(Y)}{(\sigma_X)^2} \\
&= \frac{E(XY) - E(X)E(Y)}{E(X^2) - (E(X)^2)}
\end{aligned} \tag{7}
$$

Since *E(X)* is simply the sum of *X* divided by the count of samples, if a running total is kept for *X*, *Y*, *XY*, and X^2, then the correlation function can be updated incrementally at each sensed value with a computational complexity of $O(1)$.

To allow more recent samples to have a greater impact on the correlation function we introduce a window size *W* over which to compute the statistics. We use the notation X_W to indicate the average of *X* over the window *W*. At each sensed value of X_i, X_{Wi} is recomputed using equation 8 so that the effect of older samples on the value of X_W slowly decays toward zero. We use $[XY]_W$ and $[X^2]_W$ for the averages of *XY* and X^2 respectively.

$$
X_{W_i} = \frac{W-1}{W} X_{W_{i-1}} + \frac{1}{W} X_i \tag{8}
$$

In practice, if the current number of samples *N* was less than *W*, then *N* was substituted for *W* in the equations. In that case X_W is the actual mean of the current samples of X_1 through X_N.

This leads us to the final equations for rolling least squares calculations for the correlation function used in this work shown in equation 9.

$$
\begin{aligned}
b &= \frac{[XY]_W - X_W Y_W}{[X^2]_W - (X_W)^2} \\
a &= Y_W - bX_W
\end{aligned} \tag{9}
$$

The mean square error (*MSE*), a measure of the average deviation from the correlation function, can also be computed on the fly in a similar fashion. The general

equation for calculating mean square error over variables *X* and *Y* given the correlation function defined by some *a* and *b* is shown in equation 10.

$$MSE = \frac{\sum_{i}^{N}\left(\left(Y_i - \left(aX_i + b\right)\right)^2\right)}{N} \tag{10}$$

This can be expanded and shown in the same form as the other equations used here as shown in equation 11.

$$\begin{aligned} MSE &= \frac{\sum_{i}^{N}\left(\left(Y_i - \left(aX_i + b\right)\right)^2\right)}{N} \\ &= \frac{1}{N}\sum_{i}^{N}\left(\left(Y_i - aX_i - b\right)^2\right) \\ &= \frac{1}{N}\sum_{i}^{N}\left(Y_i{}^2 - aY_iX_i - bY_i - a^2X_i{}^2 + abX_i + b^2\right) \\ &= \left[Y^2\right]_W - a\left[XY\right]_W - bY_W - a^2\left[X^2\right]_W + abX_W + b^2 \end{aligned} \tag{11}$$

The coefficient of determination, usually written as R^2 and used to measure the strength of the correlation, can also be computed incrementally. R^2 is simply the square of the *r* value from equation 6 and is shown in equation 12.

$$R^2 = \frac{\left(\left[XY\right]_W - X_WY_W\right)^2}{\left(\left[X^2\right]_W - X_W{}^2\right)\left(\left[Y^2\right]_W - Y_W{}^2\right)} \tag{12}$$

4.2. COLLABORATIVE CORRELATION

The above formulas can be used to dynamically track the correlation function between two streams as well as to periodically reevaluate which streams are correlated with which other streams.

Since the correlation function is computed in real time as the data stream is sensed, the correlation is built on the previous values and is not affected by the current sensed value until that value has been transmitted. This enables the calculations to be done on

the sink side as well the data is being decoded so that the correlation function is known without the need to transmit the correlation function across the sensor nodes wireless channel. This helps to reduce the total amount of bandwidth required by the application.

For the lossy case, the correlations must be computed after the values have been truncated to the baselines otherwise the sink side would not have the same data on which the correlations were built and would thus be unable to decode the stream unless the correlation functions were transmitted periodically along with the data.

A correlated stream can then encode its values as offsets from its correlation function of its baseline stream. A higher R^2 value indicates a higher correlation and therefore serves as a good metric for which stream to choose as a base for which other streams.

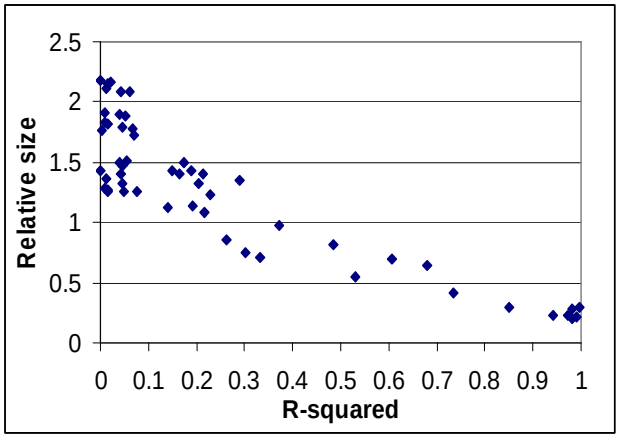

Figure 50 Compressed size for correlated pairs by r^2 value

The computational complexity for computing the correlation for every pair of streams is on the order of $O(S)$ where S is the number of streams. The number of streams on a single sensor node tends to be relatively low (the Great Duck Island weather dataset [51] had 12 which is the highest count of any of the datasets studied here). Even though the number of streams is low, the computation is still too heavy to be ideal. However, while the correlation function can be very dynamic, the sets of correlated streams tend to be

rather static, i.e., if some set of streams is found to be correlated, they are typically correlated for the entire run of the dataset. The R^2 values then need not be recomputed every time but only on occasion. Also in many applications, the computations can be done on the sink (which typically has much more processing power) and the correlated sets communicated back through the network. In our experiments, we recomputed the correlation sets every $10W$ samples (where W is the window size of the correlation functions).

To determine when to apply a correlation function, we analyzed each pair of streams on the sensor nodes from the Great Duck Island weather dataset. Figure 50 shows the R^2 value of each pair along with the compressed size using the correlation function divided by the compressed size using just the TinyPack-Init codes. If two streams were not correlated, then adding the correlation function as the baseline for a stream naturally required more bits to transmit the data. Most of the pairs of streams with an R^2 value greater than 0.25 had compression gains when using the correlation function. In our algorithm, any pair of streams with a measured R^2 value greater than 0.25 is defined as a correlated set.

If two streams are correlated to only each other, the one with the lower index is chosen as the baseline stream. If three or more are correlated to each other, then the R^2 values are summed for each pair a stream is in and the stream with the highest R^2 sum is selected as the baseline stream. For example, consider a sensor node sensing temperature (T), humidity (H), and light intensity (L) with the R^2 values for the stream pairs measured as shown in equation 13. The humidity stream would be selected as the base stream since it has the highest sum of R^2 values as shown in equation 14.

$$R^2_{T,H} = 0.68 \quad R^2_{H,L} = 0.62 \quad R^2_{T,L} = 0.53 \qquad (13)$$

$$\begin{aligned} sum_T &= R^2_{T,H} + R^2_{T,L} = 1.21 \\ sum_H &= R^2_{T,H} + R^2_{H,L} = 1.32 \\ sum_L &= R^2_{T,L} + R^2_{H,L} = 1.15 \end{aligned} \qquad (14)$$

5. EXPERIMENTAL SET UP

5.1. DATASETS

The datasets used for simulation were pulled from a wide variety of domains, which utilize wireless sensor networks including environment monitoring, animal tracking, vehicle-to-vehicle communication, and smart phone accelerometers. All are from publicly available real deployments of wireless sensor networks.

The Great Duck Island (GDI) [51] experiment deployed sensor nodes in and around the burrows of Leach's Storm Petrels. 32 sensors were deployed monitoring sensor voltage and various types of temperature, humidity, barometric pressure, and solar radiation. Data was analyzed to provide knowledge about the nesting conditions and behaviors of the birds. Strong correlations were observed between temperature, humidity, and solar radiation. Barometric pressure was also somewhat correlated.

For the Intel Berkley Labs (Lab) [52] deployment, 54 sensor nodes were configured inside a laboratory and used to transmit readings of temperature, humidity, light intensity, and voltage. Temperature, humidity, and light were all correlated, but voltage was not correlated to any other stream.

The ZebraNet project (ZNet) [53] tracked Kenyan zebras generating sensor readings of GPS position and some contextual data about the sensor nodes themselves such as the voltage, count of connected satellites, and horizontal delusion of precision. The sensors

were attached to the Zebras and data was used to analyze the social patterns of the animals.

The GATech Vehicular dataset (GATech) [66] was obtained testing a vehicle-to-vehicle network while the vehicles were in motion. Data streams included location, altitude, and speed of the vehicles along with bytes sent and received, signal strength, and noise.

The CenceMe project [67] examined the performance of a system combining off-the-shelf sensor-enabled mobile phones and the automatic sharing and aggregation of the data using social networking applications. Data was gathered by 22 different users and contained readings from the various sensors on the mobile phones including the Bluetooth, GPS, and accelerometer sensors.

5.2. IMPLEMENTATION

The algorithms were implemented in TOSSIM [17] on simulated MicaZ [15] motes. Experiments were done to show the impact of collaborative compression between the streams on bandwidth usage, energy consumption, and latency. PowerTOSSIM [70] was used to simulate the energy usage for each of the algorithms.

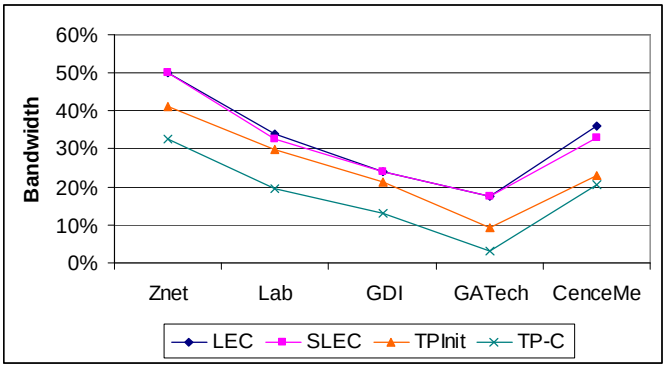

Figure 51 Bandwidth for lossless algorithms

Latency was measured by implementing the algorithms on TelosB motes [71] sending to a base station connected to a notebook computer. The data was stored on the sensor nodes before the experiments and was compressed and transmitted as if the sensors had sensed it. Thus, the time required for actually sensing the data was not included in the experiments; however, since those times are not related to the compression method used, the data would be uninteresting and would approximately be constant for each dataset.

Lossy compression was done four times for each algorithm and dataset. Maximum error was set to 5%, 2%, 1%, and 0.5% respectively for the four runs. Results are shown in the following sections.

6. RESULTS

6.1. BANDWIDTH-LOSSLESS

Bandwidth results are shown in Figure 51. Bandwidth is shown as a percentage of the bandwidth required to send the data uncompressed and is equivalent to the compressed size of the data as a percentage of the uncompressed size. Collaboration between the streams made significant improvements in bandwidth usage for most of the algorithms. The CenceMe data was not highly correlated causing TinyPack-Collaborative to only improve upon the TinyPack-Init codes by a small fraction. In contrast, compression of the GATech Vehicular dataset benefited greatly from the TinyPack-C algorithm since the data contained a high degree of correlation between the streams at a single sensor.

If no correlation is detected at all in the data, then TinyPack-Collaborative and TinyPack-Init should function identically in terms of bandwidth although TinyPack-Collaborative would consume more energy.

6.2. BANDWIDTH-LOSSY

Figure 52 shows the results of the error tolerant version of our algorithm. As with the lossless case, the introduction of correlation between the sensed streams on the individual sensor node significantly reduced the amount of bandwidth usage needed to transmit the data. As expected, all the algorithms performed better as more error was allowed in the system. The effect of leveraging correlation between the streams was roughly equivalent to the lossless case. The datasets that had high degrees of correlation saw the most benefit.

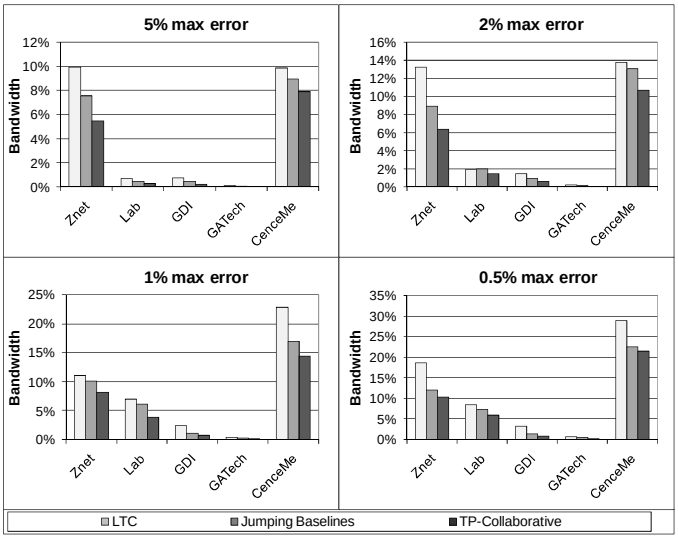

Figure 52 Bandwidth for lossy algorithms, all datasets

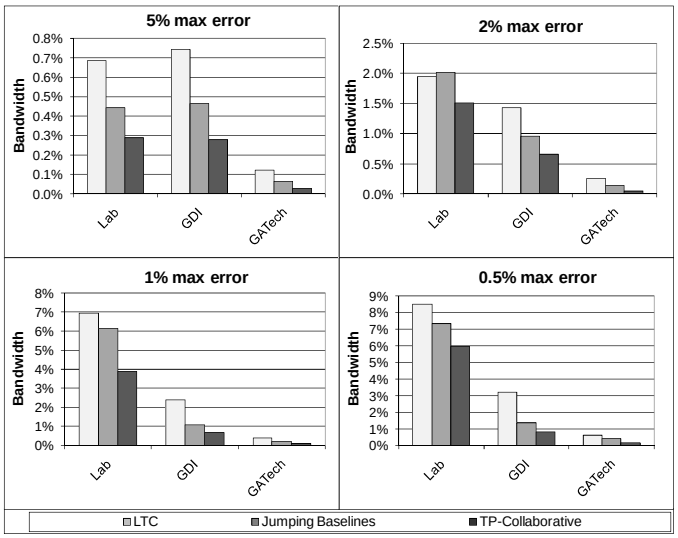

Figure 53 Bandwidth for lossy algorithms, selected datasets

The results vary greatly from one dataset to the next. This is due to the individual characteristics of the dataset. ZebraNet and CenceMe sensed data at a lower frequency than the others which decreases the benefits that can be gained by relying on temporal locality. The Lab, GDI, and GATech results are also shown in Figure 52 along with ZNet and CenceMe for comparison and are also shown in Figure 53 for greater clarity and readability.

As with the lossless case, the low degree of correlation in the CenceMe and ZNet dataset caused TinyPack-Collaborative to only perform slightly better than the other algorithms, while the GDI and GATech datasets were able to be consistently compressed to near or below half the size achieved by the Jumping Baseline algorithm.

While more tolerated error allowed for better compression in all cases, the relative compressed sizes for the different algorithms was roughly similar for all configured levels of tolerable error.

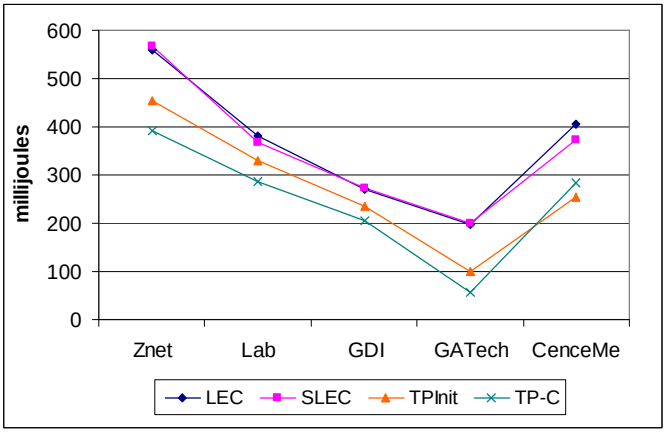

Figure 54 Energy consumption for lossless algorithms

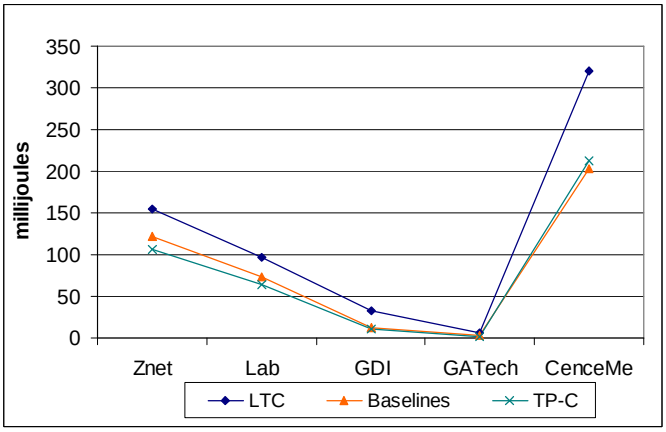

Figure 55 Energy consumption for lossy algorithms

6.3. ENERGY

The MicaZ motes simulated in PowerTOSSIM for measuring energy consumption have three different radio power settings that can be used requiring 11, 14, and 17.4 mA respectively. We selected the 11 mA radio for our experiments. Choosing a higher powered radio would make the results for energy consumption look almost identical to bandwidth since all the energy would be spent transmitting the data.

The results for the lossless case are shown in Figure 54. Since the bandwidth savings on CenceMe were not much greater for the TinyPack-C, the extra processor utilization was enough to cause it to require more energy than the jumping baseline method. The

high number of streams in the GDI dataset caused a higher increase in the energy requirements for TinyPack-C relative to the other datasets. Even using the low powered radios, the bandwidth savings are still enough to cause a lower energy profile for sensors running TinyPack-C over the other algorithms for most datasets.

The results for the lossy case are shown in Figure 55 based on the 1% maximum error configuration. The lower bandwidth requirements of the error tolerant algorithms cause the increased processor utilization to have a more significant impact on overall energy consumption; however, energy consumption for TinyPack-C was still close to or better than the other algorithms for all the datasets studied.

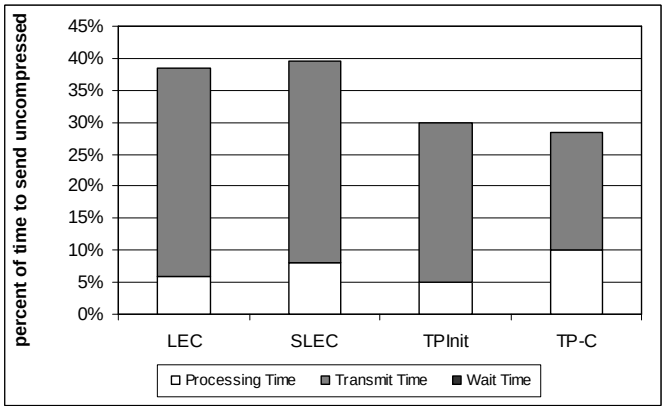

Figure 56 Latency for lossless algorithms

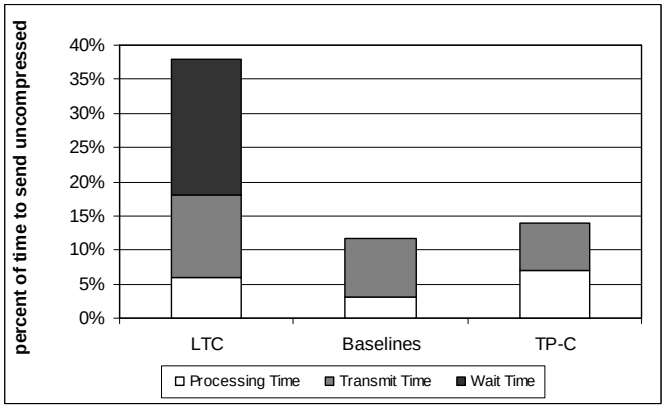

Figure 57 Latency for lossy algorithms

6.4. LATENCY

Latency results are shown for the lossless methods in Figure 56 and for lossy in Figure 57. Latency is shown as a percentage of the time that would be required to transmit the data uncompressed. Results are shown as the average across all the datasets including the processing, transmission, and wait time used by the algorithms.

As with energy, the higher processor utilization for TinyPack-Collaborative caused an increase in latency compared to the lighter weight TinyPack-Init and jumping baseline methods; however, in a multi-hop environment, the average latency per hop decreases with each hop and approaches the sum of the transmit time and the wait time as shown in Figure 58.

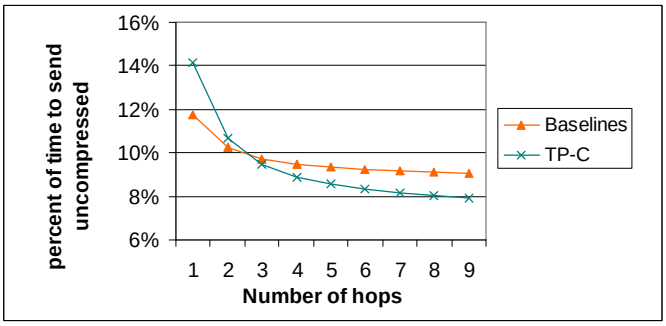

Figure 58 Latency for multi-hop environment

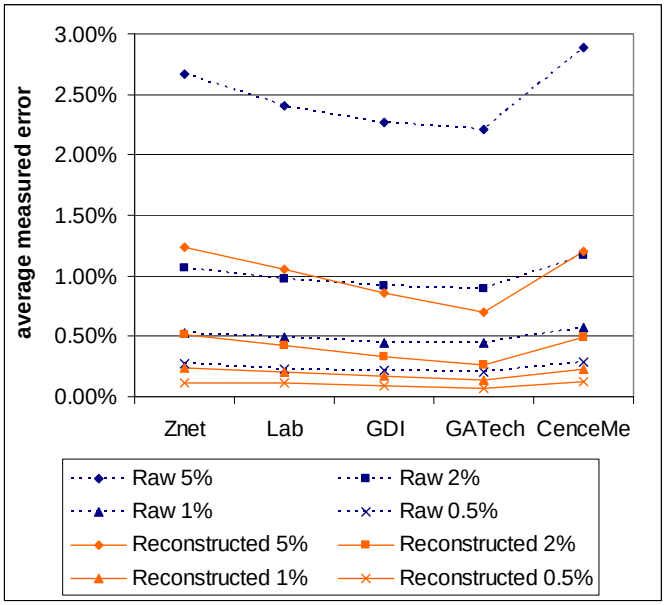

Figure 59 Average total error for raw baseline and reconstructed

7. ERROR ANALYSIS

The step function used to approximate the stream in the lossy case can be reconstructed into a series of line segments as done for the jumping baselines in [61]. This can reduce the total measured error in the data. The points at which new baselines were selected are used as the endpoints of the line segments.

Since the algorithm tracks whether the data was trending up, trending down, or peaking, this information can be used to better approximate the end points. If the data was trending up or down, then the line segment endpoint is selected as the average of the previous and current baselines. If the data is peaking (last jump was up, current jump was down or vice versa), then the previous baseline value serves as the endpoint.

Figure 59 shows the total error for both the raw baseline step function and for the reconstructed streams for each of the four configured maximum error percentages. Total error for the step functions is shown as dotted lines. The total error after reconstructing the streams as sequences of line segments are shown as solid lines. Data points for both

raw and reconstructed for the same maximum error are shown with the same shape in the figure.

Raw baseline step function total error was typically around one half of the maximum tolerable error. This is expected since the candidate baselines are integer multiples of the maximum tolerated error. The total error for the reconstructed streams ranged from around one quarter to one sixth of the maximum tolerable error. The more the data in a stream approximates a straight line over a short interval, the more accurate the reconstruction.

Experiments were also conducted using b-spline interpolation as a curve fitting technique, but the results were almost identical to the linear approximation and were much more computationally intense.

8. AGGREGATION OF COMPRESSED VALUES

As detailed previously, TinyPack-Collaborative, for both lossless and lossy compression, transmits values as the delta over some previous value or baseline function encoded using the TinyPack-Init codes. Some mathematical operations and aggregation can be performed on these encoded deltas without the need to first decode the data.

For instance, in an ad-hoc network, if an intermediate node between the sensor publishing the data and the base station begins forwarding data without seeing the initial baseline value, it can still perform aggregations on the data which the base station can apply to the baseline.

8.1. ADDING ENCODED VALUES

Adding two encoded deltas can be done without converting the value to a standard encoded integer. The codes contain a prefix, a suffix and a sign bit. In the case of two

positive or two negative numbers, the two suffixes with their prefix bits prepended can be added in simple binary, if the high prefix bit overflows (is set to 0), then the prefix length is incremented by one and the sign bit remains unchanged. In the case of a positive and negative number, the negative number is expressed in 2's complement. The two numbers are added as before and the prefix length is reduced by the number of leading zeros in the sum.

8.2. DROPPING PACKETS

If a sensor network is being overloaded such that a sensor needs to conserve additional bandwidth, one common method for quick bandwidth savings is to drop a packet. In a compressed stream, simply dropping a packet causes the decoding process to produce incorrect results; however, delta compressors such as TinyPack-Collaborative can drop packets without invalidating the data as long as the delta values of all the dropped packets are summed into the next transmitted packet. For example, if a sensor received the values 5, 7, 12 9 10 and transmitted them as +5, +2, +5, -3, +1 and needed to drop every other packet, it could send +5, +7, -2 and the sink would decode them as 5, 12, 10. Any intermediate nodes need not know the baseline on which the first packet is based.

8.3. MINIMUM AND MAXIMUM

Maintaining the maximum of a portion of a stream can be done without knowing the baseline by maintaining the current max delta and offset from the max delta by summing the delta values. For example, consider a sensor in an ad hoc network that samples the following values: 15, 13, 10, 12, 17, 13. The 15 is transmitted to the base station through one intermediate node and the remaining values through another node. The new

intermediate node first sees the -2 and maintains the max as shown in Table 35. Minimum can be maintained equivalently.

Table 35 Max delta example

sensed value	sent delta	current max delta	offset from max	actual max (delta+15)
15	--	--	--	15
13	-2	0	2	15
10	-3	0	5	15
12	+2	0	3	15
17	+5	+2	0	17
13	-4	+2	4	15

8.4. AVERAGE

Maintaining an average of a portion of a stream can be done without knowing the baseline as long as the count of samples included in the average is transmitted. The intermediate sensor maintains the current offset by keeping a running sum of the delta values. The sensor then maintains a sum of those offsets. Dividing that sum of offsets by the count gives the average delta value which can be added by the base station to the known baseline value to obtain the overall average. For example, consider a sensor that samples the following values: 10, 13, 17, 14, 8, 7, 15. Again, the intermediate node starts receiving and forwarding the data in the middle of the stream starting with the 13. This process is shown in Table 36.

Table 36 Average delta example

sensed value	sent delta	sum of deltas	sum of sums	count	avg delta	actual avg (delta+10)
10	--	--	--	0		--
13	+3	+3	+3	1	3	13
17	+4	+7	+10	2	5	15
14	-3	+4	+14	3	4.67	14.67
8	-6	-2	+12	4	3	13
7	-1	-3	+9	5	1.8	11.8
13	+6	+3	+12	6	2	12

9. CONCLUSIONS AND FUTURE WORK

TinyPack-Collaborative compression performed well compared to related methods in terms of bandwidth usage, energy requirements, and end-to-end latency. Collaboration between the data streams improved the compression performance in all experiments compared to compression without inter-stream collaboration. While collaboration between the same streams on different sensor nodes has been shown to be effective in increasing compression gains in other published works, collaboration between streams on the same sensor node can also be used to achieve greater compression leading to longer deployments, more data collection, fewer collisions, and faster response times for a wide variety of wireless sensor applications.

While the rolling least squares regression used here was shown to be effective, other more sophisticated methods such as Kalman Filters [39] or Principal Component Analysis [73] could be potentially improve the accuracy of the baseline correlation functions.

SECTION

2. CONCLUSIONS

The compression algorithms presented in this document have been demonstrated to be effective at reducing bandwidth requirements, energy consumption, and latency for many different types of wireless sensor networks. Using these algorithms in a wireless sensor network thus allows for cost savings, longer deployments, more data collection, fewer collisions during transmission, and reduced latency in data delivery.

BIBLIOGRAPHY

[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. Mica2 and MicaZ Datasheets http://www.xbow.com/, 2010.

[16] N. Chaimanonart, M. Suster, W. Ko, and D. Young. "Two-Channel Data Telemetry with Remote RF Powering for High-Performance Wireless MEMS Strain Sensing Applications" in 4th IEEE Conference on Sensors, 2005.

[17] 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.

[18] 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.

[19] 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.

[20] A. Sharaf, J. Beaver, A. Labrinidis, and K. Chrysanthis. "Balancing energy efficiency and quality of aggregate data in sensor networks." In The VLDB Journal, 13(4):384–403, 2004.

[21] A. Deligiannakis, Y. Kotidis, and N. Roussopoulos. "Hierarchical In-Network Data Aggregation with Quality Guarantees." In Proceedings of EDBT Conference, 2004.

[22] 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.

[23] H. Liefke and D. Suciu, "Xmill: an efficient compressor for xml data," In Proceedings of the 2000 ACM SIGMOD international conference on Management of data, vol. 29, no. 2, pp. 153-164, June.

[24] J. Cheney, "Compressing XML with multiplexed hierarchical PPM models," In Proceedings of the Data Compression Conference, IEEE, 2001.

[25] Wireless Application Protocol Forum, Ltd. Binary XML Content Format Specification. WAP Forum, 2001.

[26] H. Subramanian and P. Shankar. “Compressing XML Documents Using Recursive Finite State Automata,” In Implementation and Application of Automata, volume 3845 of LNCS, pages 282-293. Springer, 2006.

[27] PAQ. http://cs.fit.edu/~mmahoney/compression.

[28] T. Szalapski and S. Madria, “Real-Time Data Compression in Wireless Sensor Networks,” In the 12th International Conference on Mobile Data Management, 2011.

[29] 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.

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

[31] 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.

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

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

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

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

[36] 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.

[37] 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.

[38] 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.

[39] Olfati-Saber, R., "Distributed Kalman filtering for sensor networks," In Decision and Control, 2007 46th IEEE Conference on. Dec. 2007.

[40] S. Goel and T. Imielinski. "Prediction-based monitoring in sensor networks: taking lessons from MPEG." In SIGCOMM Computer Communications Rev. 31, 5 (October 2001), 82-98.

[41] A. Ali, A. Khelil, P. Szczytowski, and N. Suri. "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 (MSWiM '11). ACM, New York, NY, USA, 67-76.

[42] 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.

[43] 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.

[44] T. Szalapski and S. Madria, “Real-Time Data Compression in Wireless Sensor Networks,” In the 12th International Conference on Mobile Data Management, 2011.

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

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

[47] 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.

[48] 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.

[49] Willow Technologies. http://www.willow.co.uk/TelosB_Datasheet.pdf, 2013.

[50] Sanjay Madria, Vimal Kumar and Rashmi Dalvi, Sensor Cloud: A Cloud of Virtual Sensors, IEEE Software, Nov 2013.

[51] 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.

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

[53] 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.

[54] 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.

[55] 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.

[56] T. Szalapski and S. Madria, "On Compressing Data in Wireless Sensor Networks For Energy Efficiency and Real Time Delivery," In Distributed and Parallel Databases. June 2013, Volume 31, Issue 2, pp 151-182.

[57] A. Rooshenas, H.R Rabiee, A. Movaghar, M.Y. Naderi. "Reducing the data transmission in Wireless Sensor Networks using the Principal Component Analysis." Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP) 133-138, 7-10 Dec. 2010

[58] R. Masiero, G. Quer, M. Rossi. M. Zorzi.. "A Bayesian analysis of compressive sensing data recovery in wireless sensor networks." In Ultra Modern Telecommunications & Workshops, 2009. (ICUMT'09). 1-6. 2009.

[59] 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.

[60] A. Ali, A. Khelil, P. Szczytowski, and N. Suri. "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 (MSWiM '11). ACM, New York, NY, USA, 67-76.

[61] T. Szalapski and S. Madria, " Energy Efficient Distributed Grouping and Scaling for Real-Time Data Compression in Sensor Networks." In communication.

[62] Y. Liang, Y. Li. "An Efficient and Robust Data Compression Algorithm in Wireless Sensor Networks." Communications Letters, IEEE, vol.18. 439-442. March 2014

[63] T. Schoellhammer, B. Greenstein, E. Osterweil, M. Wimbrow, D. Estrin. "Lightweight temporal compression of microclimate datasets [wireless sensor networks]." 29th Annual IEEE International Conference on Local Computer Networks. 16-18 Nov. 2004.

[64] A. Vahidi, A. Stefanopoulou, and H. Peng. "Recursive least squares with forgetting for online estimation of vehicle mass and road grade: theory and experiments." Vehicle System Dynamics 43. pp. 31-55. 2005.

[65] M. Salgado, G. C. Goodwin, and R. H. Middleton. "Modified least squares algorithm incorporating exponential resetting and forgetting."International Journal of Control 47, no. 2 pp. 477-491. 1988.

[66] R. M. Fujimoto, R. Guensler, M. P. Hunter, H. Wu, M. Palekar, J. Lee, and J. Ko. "CRAWDAD dataset gatech/vehicular. v. 2006-03-15. Downloaded from http://crawdad.org/gatech/vehicular. Mar 2006.

[67] E. Miluzzo, N. D. Lane, K. Fodor, R. Peterson, H. Lu, M. Musolesi, S. B. Eisenman, X. Zheng, and A. T. Campbell. "Sensing meets mobile social networks: the design, implementation and evaluation of the cenceme application." In Proceedings of the 6th ACM conference on Embedded network sensor systems, pp. 337-350. ACM, 2008.

[68] 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.

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

[70] 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.

[71] Willow Technologies. http://www.willow.co.uk/TelosB_Datasheet.pdf, 2013.

[72] Olfati-Saber, R., "Distributed Kalman filtering for sensor networks," In Decision and Control, 2007 46th IEEE Conference on. Dec. 2007.

[73] A. Rooshenas, H. R. Rabiee, A. Movaghar, and M. Y. Naderi. "Reducing the data transmission in wireless sensor networks using the principal component analysis." In Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2010 Sixth International Conference on, pp. 133-138. IEEE, 2010.

VITA

Thomas Mark Daniel Szalapski obtained Bachelor of Science degrees in Computer Science and in Applied Mathematics at the Missouri University of Science & Technology in May 2008 and was awarded a Doctor of Philosophy degree in August 2014.