A Comparison of On-Mote Lossy Compression Algorithms for Wireless Seismic Data Acquisition

Marc J. Rubin, Michael B. Wakin, and Tracy Camp
Dept. of Electrical Engineering and Computer Science
Colorado School of Mines, Golden, CO
mrubin@mines.edu, mwakin@mines.edu, tcamp@mines.edu

***Abstract*—In this article, we rigorously compare compressive sampling (CS) to four state of the art, on-mote, lossy compression algorithms (K-run-length encoding (KRLE), lightweight temporal compression (LTC), wavelet quantization thresholding and run-length encoding (WQTR), and a low-pass filtered fast Fourier transform (FFT)). Specifically, we first simulate lossy compression on two real-world seismic data sets, and we then evaluate algorithm performance using implementations on real hardware. In terms of compression rates, recovered signal error, power consumption, and classification accuracy of a seismic event detection task (on decompressed signals), results show that CS performs comparable to (and in many cases better than) the other algorithms evaluated. The main benefit to users is that CS, a lightweight and non-adaptive compression technique, can *guarantee* a desired level of compression performance (and thus, radio usage and power consumption) without subjugating recovered signal quality. Our contribution is a novel and rigorous comparison of five state of the art, on-mote, lossy compression algorithms in simulation on real-world data sets and implemented on hardware.**

I. Introduction

Creating a low-cost wireless sensor network (WSN) for continuous (e.g., 250 Hz sampling rate) geohazard monitoring necessitates a better approach than a simplistic "sense, send" modality. Since the radio on a wireless device consumes orders of magnitude more power than other components (e.g., ADC, CPU) [1], streaming all the data may consume too much power to be viable. As such, using compression to reduce radio transmissions will help increase system longevity, decrease overall system power requirements, and decrease system costs.

There have been several lossy[1] compression algorithms devised specifically for resource constrained wireless motes (e.g., 8-16 MHz CPU, 2-10 kB RAM). These algorithms include: K-run-length encoding (KRLE) [2], lightweight temporal compression (LTC) [3], wavelet quantization thresholding and RLE (WQTR) [4], low-pass filtered fast Fourier transform (FFT) [5], and compressive sampling (CS) [6], [7], [8]. In this article, we compare these five on-mote, lossy compression algorithms as potential data reduction techniques on real-world seismic data.

The main contribution of this paper is a rigorous evaluation and analysis comparing our lightweight and novel CS technique called Randomized Timing Vector (RTV) [6] to four other on-mote, lossy compression algorithms (KRLE, LTC, WQTR, and FFT) using identical real-world seismic data sets. Consequently, this work provides a novel comparative study of five state of the art on-mote, lossy compression techniques. Previous literature comparing on-mote lossy compression algorithms [9] did not simulate, implement, or evaluate the compression algorithms on the same set of data. Instead, the authors of [9] discussed the merits of each algorithm based on the mutually exclusive results presented in the literature surveyed; in other words, the algorithms were compared based on results from different data sets. We also note that the survey conducted by [9] did not include KRLE, WQTR, FFT, or our CS algorithm, RTV.

Results depicted in this paper demonstrate why CS, a lightweight and non-adaptive compression algorithm, is an attractive option for on-mote lossy compression. Specifically, CS offers guaranteed compression performance, low recovery error, and low power consumption without subjugating decompressed signal quality. We note that lossy compression is not appropriate for exploration geophysics, where little is known about the target signal being acquired. However, lossy compression is a feasible data reduction technique for seismic event detection, where there is a priori knowledge of the target signal and some information loss is acceptable. Moreover, we note that lossless compression is not covered in this article; comparing lossy to lossless compression is beyond the scope of this work.

II. Background

In this section we describe the five lossy compression algorithms used in this study. We provide implementation details where appropriate.

A. K-Run-Length Encoding (KRLE)

The authors of [2] propose a novel lossy adaptive compression algorithm, called K-run-length encoding (KRLE), which allows for some variability in the input data stream during data encoding. KRLE encodes the input signal by using a range of acceptable values specified by a parameter K. Specifically, the current value (y) of an input stream is considered redundant if it falls within some predefined range of the first novel value (x), that is, if $x - K \leq y \leq x + K$. For example, with $K = 3$, the sequence of integers $\{61, 62, 63, 64, 65\}$ are encoded simply as $\{61 : 4; 65 : 1\}$, which indicates that the decoder should reconstruct the value 61 four times and then the value 65 once. Experimental results from [2] show that

[1] A lossy compression algorithm means that some information is lost during compression and decompression.

KRLE can significantly increase compression rates of certain signals, where the compression rate is defined as:

$$CompressionRate = 100 \times \left(1 - \frac{CompressedSize}{OriginalSize}\right).$$

B. Lightweight Temporal Compression (LTC)

Much like KRLE, lightweight temporal compression (LTC) adaptively compresses data by encoding streams of redundant sequences [3]. LTC is different from KRLE in the way redundancy is defined. In LTC, a data point is considered redundant if it falls within some range of lines interpolated from previous data points. If the current data point falls within some user-specified range of interpolated lines (specified by a parameter K), then the data point is encoded as redundant. Otherwise, the current data point is used to start the next iteration of interpolation and compression. Results from [3] show that LTC performs comparably to Lempel-Ziv-Welch and wavelet based compression on micro-climate data.

C. Wavelet Quantization Thresholding and RLE (WQTR)

The authors of [4] describe a lossy adaptive compression algorithm that we refer to as wavelet quantization thresholding and RLE (WQTR). First, the WQTR algorithm works by calculating a discrete wavelet transform using Cohen-Daubechies-Feauveau (2,2) integer wavelets (CDF(2,2)) on subsets of 128 samples. Integer wavelets were selected because they can be implemented using only addition and bit shifting operations. Second, the wavelet coefficients are quantized to reduce signal resolution, which decreases the size of the signal and makes the signal more compressible. Third, the coefficients undergo thresholding, where coefficients with absolute values above some percentage threshold are kept while the other coefficients are zeroed out. The resulting signal, consisting of a few large quantized wavelet coefficients and many zeros, is then passed to a run-length-encoder (RLE[2]) and transmitted. Results from [4] show that WQTR's increased compression rates and low overhead make it a viable option for the authors' WSN deployment.

D. Low-pass Filtered Fast Fourier Transform (FFT)

The fast Fourier transform (FFT) is a well-known and efficient method to transform signals from the time to frequency domain [5], [10]. Briefly, an N-point FFT takes N complex numbers as input and produces N complex FFT coefficients; within a mote's memory, the FFT's input and output both consist of N real and N imaginary components. Assuming the imaginary component of the input is zero, as it is with a real-valued seismic signal, the real and imaginary components of an N-point FFT's output are symmetric and antisymmetric about the center frequency, respectively. Thus, instead of transmitting $2N$ numbers to represent the FFT's N complex coefficients, we make use of the FFT's symmetry to transmit $N/2$ real and $N/2$ imaginary components (i.e., the first "half" of the FFT's output). We recover the N complex FFT coefficients by mirroring the real and imaginary coefficients about the center frequency and multiplying the mirrored imaginary components by -1 (since it is antisymmetric).

To implement non-adaptive compression, we employ low-pass filtering on the Fourier coefficients before transmission; low-pass filtering allows low-frequency coefficients to "pass-through" the filter, zeroing out the frequency coefficients above a user-defined threshold [5]. In other words, we achieve non-adaptive compression by transmitting the lowest (in terms of frequency bins) L real and L imaginary components of the FFT's output, where $L < N/2$. During offline signal recovery, the $N/2-L$ real and $N/2-L$ imaginary components not transmitted are set to 0. The full time-domain signal is recovered using the inverse Fourier transform on the N recovered complex FFT coefficients. Our implementation of FFT compression is based on the source code provided by the Open Music Labs [10], which computes a fixed point FFT. We note that, although FFT-based adaptive compression (similar to WQTR) could have been implemented, we chose the non-adaptive approach for the sake of comparison against CS, which is a non-adaptive compression algorithm.

E. Compressive Sampling (CS)

Compressive sampling (CS) is the final lossy compression algorithm evaluated in this article. CS is motivated by the desire to simplify both the sensing *and* compression processes. We have proposed a novel lightweight CS algorithm called Randomized Timing Vector (RTV), which, after initialization, achieves lossy compression using only a 'for' loop and 'if' statement [6]. While other on-mote CS algorithms such as Additive Random Sampling [7] and Sparse Binary Sampling [8] have been proposed, herein we use our RTV algorithm for the CS implementation due to its superior performance [6].

Briefly, CS displaces the traditional (and often wasteful) mantra of "sample *then* compress" with "compress *while* sampling". At its core, CS uses numerical optimization methods to recover full-length signals from a small number of randomly collected samples. For CS, the computational burden occurs offline during signal recovery, and not on-mote during signal compression. In CS, compression occurs via a matrix multiplication of a randomized measurement matrix Φ (size $M \times N$, with $M \ll N$) with the original signal x (length N) to obtain a compressed vector y of length M (i.e., $y = \Phi x$). However, it is possible to implement on-mote CS without fully sampling x or computing a costly matrix multiplication [6]. In other words, our RTV algorithm greatly simplifies compression by *acquiring the data directly in compressed form*. For a thorough explanation of CS and RTV, we refer the reader to [11] and [6], respectively.

CS is advantageous because it greatly simplifies the compression process by acquiring compressed signals directly and shifts the computational burden away from the wireless mote to the system performing signal decompression. Thus, given the lightweight and non-adaptive nature of CS data compression, how does CS compare to the four other lossy algorithms (three adaptive and one non-adaptive) described previously?

[2]RLE is equivalent to KRLE with $K = 0$.

To answer this question, we analyzed the performance of the five lossy compression algorithms in two scenarios: 1) in simulation on two sets of real-world seismic data, and 2) on real hardware. We describe our experimentation, implementations, and results in the next two sections.

III. Simulation

We first simulated the five lossy compression algorithms on real-world seismic data collected in the mountains above Davos, Switzerland. The simulation experiments allowed us to evaluate the compression algorithms in terms of compression rates, recovery error rates, and event classification accuracies. Additionally, we further support our findings by simulating the compression algorithms on a second real-world seismic data set collected from a test levee in the Netherlands called IJkdijk (pronounced "Ike-dike").

The seismic data used in the first set of simulations was collected during the 2009-2010 winter season using a *wired* geophone array in the mountains above Davos, Switzerland [12]. The full seismic data set, sampled at 500 Hz with 24-bit precision, contains over 100 days of data with 33 slab avalanche events (e.g., Figure 1). A slab avalanche event is when a large, dangerous mass of snow and/or ice debris tumbles down a mountain, burying and/or damaging everything in its path. More information regarding the wired geophone deployment and subsequent geophysical data analysis can be found in [12]. Wirelessly transmitting all of this data would require a tremendous amount of power.

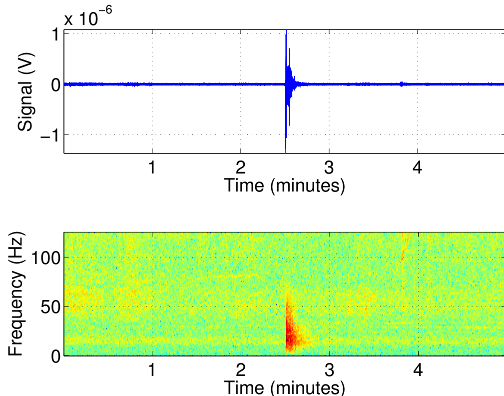

Fig. 1: Five minutes of seismic data containing a slab avalanche event, plotted in the time domain (top) and time-frequency domain (bottom). We simulated on-mote lossy compression on 2.75 hours of real-world seismic data, which contained 33 slab avalanches events that occurred in the 100+ days of data.

In our experiments, we simulated compression on the 33 five-minute chunks of seismic data containing slab avalanches from the *wired* deployment in 2009-2010. To simulate our real-world *wireless* mote deployment (recently installed and collecting geophone data above Davos, Switzerland), we subsampled the 33 slab avalanches from the original 500 Hz sample rate with 24-bit precision to a 250 Hz sample rate with 16-bit precision. Additionally, we performed compression on the raw ADC encodings, as these values are two byte signed integers, not floating point voltage values.

In terms of algorithm parameters, we selected powers of two for KRLE and LTC (i.e., $K = \{1, 2, 4, ..., 512\}$). For WQTR, we used thresholds between 10% and 90%, and added a 98% threshold for aggressive compression. For FFT, we selected low-pass filter thresholds between 10% and 90% of the center frequency. For CS, we employed compressed vector lengths of $M = \{0.1N, 0.2N, ..., 0.9N\}$. Note that the ratio of compressed vector length to full signal (i.e., M/N) is inversely proportional to the compression rate, e.g., a 30% ratio between M and N results in a 70% compression rate.

To increase the credibility of our simulations, we used the same C++ compression functions and memory usage as our on-mote implementations of CS, KRLE, LTC, and WQTR (see Section IV). In other words, we first implemented the compression functions in C++ for Arduino and then ported the methods to a desktop computer using a different driver to read in the seismic data. Additionally, we simulated the memory constraints of the Arduino Fio platform by limiting the size of each data buffer to be compressed. We note that since the FFT library obtained from the Open Music Labs [10] was written in assembly for Arduino, we used custom Matlab functions (with limited precision) for our simulations.

To simulate our hardware implementation, we compressed signals using a two buffer method; while one buffer of data was being acquired, the other buffer was being compressed and transmitted. Specifically, for KRLE, LTC, and CS, we compressed buffers of $N = 256$ short (two byte) integers at a time. Due to the increased memory required for computation, we compressed buffers of $N = 128$ short integers for FFT and WQTR.

In terms of radio usage, we simulated binary transmissions. For CS, this meant transmitting the M short integers selected during compression. For KRLE and LTC, we transmitted three bytes at a time: two bytes for the signed short integer and one byte for the number of occurrences. For FFT, we transmitted $2L$ short integers corresponding to the L real and L imaginary components of the low-pass filtered FFT coefficients. Lastly, for WQTR, we transmitted five bytes at a time: a four-byte floating point wavelet coefficient followed by one byte for the number of occurrences. We omitted the quantization step of WQTR due to extremely poor performance during simulation; though quantization equates to better compression rates, the loss of precision from normalizing, truncating, and/or interpolating the wavelet coefficients from four-byte floating points to two-byte shorts resulted in significantly higher recovered signal error rates. In other words, quantizing the wavelet coefficients to match the other algorithms (i.e., from floating point to short integer) resulted in recovered signals that were unusable.

A. Compression Rates

After simulating compression on the 33 five-minute chunks of data containing slab avalanches, we calculated the compression rates using the formula defined in Section II-A. Figure 2 shows histograms of the compression rates for each algorithm over the entire simulation.

The compression rates presented in Figure 2 show two clear advantages to using non-adaptive compression. First, nonneg-

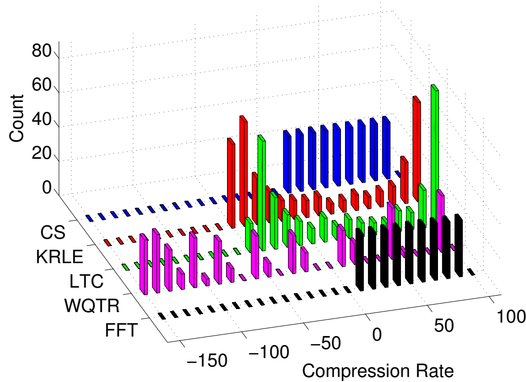

Fig. 2: Histograms of the compression rates from the simulation on the avalanche data. Note that CS and FFT (the two non-adaptive algorithms) are the only algorithms without negative compression rates.

ative compression rates[3] can be guaranteed. In other words, the rate of compression for non-adaptive algorithms does not depend on the compressibility of the input signal. Additionally, with CS and FFT, users can specify a compression rate for the lifetime of the mote by selecting the M parameter and the low-pass filter threshold, respectively.

The second advantage is that the non-adaptive compression algorithms have less variability in the resulting compression rates. We encourage the reader to note the high variability of compression rates for KRLE, LTC, and WQTR in Figure 2 compared to the low variability of compression rates for CS and FFT. In the case of KRLE and LTC, the combination of small K values and high signal variance led to negative compression rates. For example, with a K value of 1, KRLE would encode a signal {0,2,4} as {0:1,2:1,4:1}, which requires more bytes than the original signal.

B. Recovered Signal Error

For signal recovery (decompression), we utilized a combination of C++, Python, and Matlab. Decoding KRLE and LTC based encodings was straightforward, since these algorithms specify what number to print and how many times it occurred. WQTR decompression also included this decoding step, followed by an inverse wavelet transform. Decoding the FFT algorithm's output required recovering the N FFT coefficients and computing the inverse FFT. Lastly, for CS, we employed reweighted ℓ_1-norm minimization [13] assuming sparsity in the time-frequency domain (Gabor atoms); in previous work, we verified that this real-world data set is sparse in the time-frequency domain [14].

We computed signal recovery errors by computing the normalized root mean square error (NRMSE) for the decompressed versus original signals. NRMSE is calculated as:

$$NRMSE = \frac{\sqrt{mean((x-\widehat{x})^2)}}{max(x)-min(x)},$$

[3]Negative compression rates occur when the number of bytes required for the compressed signal encoding is greater than the original signal.

where $\widehat{x}$ is the recovered signal and x is the original signal containing a slab avalanche. Figure 3 plots the mean NRMSE and 95% confidence intervals of all five algorithms based on their respective mean compression rates. For KRLE, LTC, and WQTR in Figure 3, the K values and thresholds increase from left to right. For example, KRLE with $K = 1$ resulted in approximately -47% mean compression rate and almost zero NRMSE, while $K = 512$ resulted in 97% mean compression rate and 0.017 NRMSE.

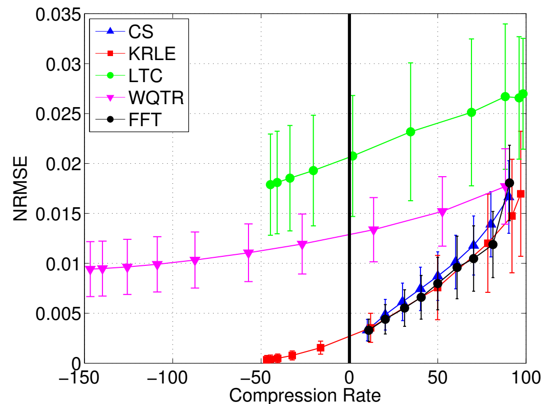

Fig. 3: Mean NRMSE results (with 95% confidence intervals) from five lossy wireless node compression algorithms simulated on a real-world seismic data set containing avalanches.

The NRMSE results in Figure 3 show that CS, a lightweight non-adaptive compression technique, performs as well as the best performing algorithms: KRLE and FFT. In other words, the recovery errors of signals compressed and decompressed with CS fell within all the 95% confidence intervals of KRLE and FFT. Moreover, as shown, KRLE sometimes provides negative compression rates.

The NRMSE results are quite striking, considering that CS compression does *not* necessitate acquiring and storing every sample in the original signal. In other words, instead of acquiring the full signal and *then* performing compression, CS allows us to acquire the compressed signal directly. In geophysical monitoring applications where data acquisition is expensive (e.g., due to power consumption from high sampling rates), CS is an attractive option because it is the only compression technique that does not require acquiring the full signal first [6].

Though NRMSE results provide a nice depiction of compression rates and recovery error, the error rates alone do not paint a complete picture. For example, what does it mean for CS to have 0.017 mean NRMSE at 90% compression rate? Is the recovered signal still useful?

C. Avalanche Event Classification

In hopes of answering such questions, we applied our automated avalanche detection workflow [15] to the recovered (decompressed) signals from the five compression algorithms. First, we divided each signal into consecutive five second frames. A class label was then assigned to each five second frame, depending on whether the frame contained part of an

avalanche event. Each frame was transformed from the time to the frequency domain using a 1024-sample, non-overlapping, normalized FFT. From the frequency domain we extracted 10 features common in acoustic signal processing: centroid, 85% rolloff, kurtosis, spread, skewness, regularity, flatness, and the maximum, mean, and standard deviation of the top 1% most powerful frequencies. See [15] for more details.

After extracting features from the original and recovered signals, we then moved to the machine learning task. Specifically, for each of the recovered and original signals, we trained and tested a decision tree classifier on subsets of the data using a 10-fold cross-validation procedure. We used stratified subsampling to create the training and testing subsets, which included all five-second avalanche frames and an equal number of randomly selected non-avalanche frames (to avoid overfitting).

We ran a 10-fold cross-validation procedure 100 times per recovered signal, randomly selecting a new set of non-avalanche frames each time. Figure 4 plots the mean classification accuracies and 95% confidence intervals of the five evaluated compression algorithms based on mean compression rates.

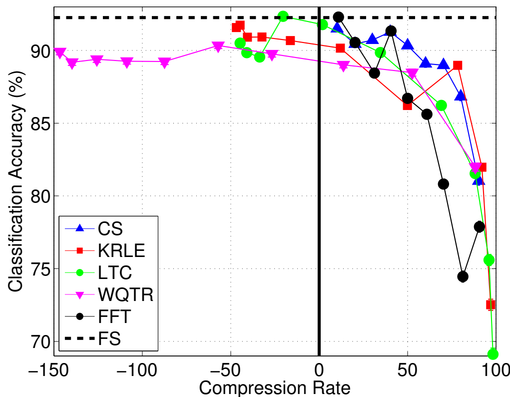

Fig. 4: Mean and 95% confidence intervals of classification accuracies from our machine learning workflow (to detect slab avalanches) performed on the recovered signals. FS shows the classification accuracy for the full (uncompressed) signal.

There are three interesting trends in Figure 4 worth noting. First, classification results show that CS performed quite well. For example, the mean accuracy of detecting slab avalanche events recovered from 60% compressive sampling (40% compression rate) was 91.3%. In comparison, with full sampling, we reached 92.3% mean classification accuracy. In other words, the 40% increase in compression rate for CS over full sampling resulted in only a 1% decrease in mean classification accuracy.

Second, observe the "bumps" of increased classification accuracies for KRLE with parameter $K = 128$ (78% compression rate) and FFT with low-pass filter thresholds of 60% and 10% (40% and 90% compression rates, respectively). We believe that these temporary improvements in classification accuracies occur because the algorithms were effectively denoising the data before feature extraction. For example, by encoding many values as a single number, KRLE removes frequency content and greatly simplifies the signal during periods of little variance. However, the classification accuracies of KRLE with $K = 256$ and $K = 512$ (92% and 97% compression rates, respectively) degrade rapidly to 82% and 73%, respectively. Likewise, the FFT's mean classification accuracies quickly degrade from above 90% accuracy (with 40% compression rate) to below 75% accuracy (with 80% compression rate). These rapid downward trends in mean classification accuracies given higher K values and lower filter thresholds suggests that both KRLE and FFT remove useful frequency information from the signal before feature extraction and machine learning.

Lastly, although CS and FFT both had exclusively nonnegative compression rates, FFT's accuracies were highly variable or significantly worse than the other algorithms evaluated. We hypothesize that this occurs because the low-pass filtered FFT explicitly removes the mid to high frequency components of the recovered signal, thus eliminating information that may be critical for our pattern recognition workflow to detect avalanches. Herein lies an advantage of CS over FFT; with high rates of compression (i.e., greater than 50%) it appears that CS recovers more useful information from the compressed signal than the low-pass filtered FFT. For example, with 50% compression rate, FFT had a mean classification accuracy of 86.7% while CS had a mean classification accuracy of 90.3%.

D. IJkdijk Seismic Data

Given that CS compares favorably to other lossy compression algorithms in terms of compression rates, NRMSE, and classification accuracies on one data set, how does CS perform on a different data set? To answer this question, we ran all five compression algorithms on a seismic data set collected from the IJkdijk test levee in the Netherlands. Briefly, IJkdijk is a test levee that is monitored and measured in various ways while it is brought to failure. Geoscientists collected several days of 16-bit passive seismic data from 24 *wired* geophones as the levee was brought close to failure. For our experiments, we simulated compression on a small segment of data (about 50 minutes) from a single geophone sensor deemed interesting by the team of geophysicists, geologists, and geotechnical engineers (e.g., Figure 5). The data was subsampled from 4000 Hz to 250 Hz to mimic the current bandwidth limits of our low cost *wireless* geophysical sensors. Results from our compression simulations are plotted in Figure 6, which shows the mean NRMSE with 95% confidence intervals of the recovered data.

Though the errors in Figure 6 are larger than in Figure 3, the relative ordering of the lossy algorithms remains approximately the same. In this case, with 50% or less compression rates, CS performs the best, with KRLE close behind; above 50% compression rates, CS was tied with FFT as the best performing algorithm. It is interesting to note the relatively flat performance of WQTR (in terms of NRMSE). We hypothesize that this is due to the simplicity of the CDF(2,2) integer wavelet and relatively small buffer size used in WQTR compression.

We did not evaluate the IJkdijk signals in terms of machine learning accuracy, simply because we have not yet designed a

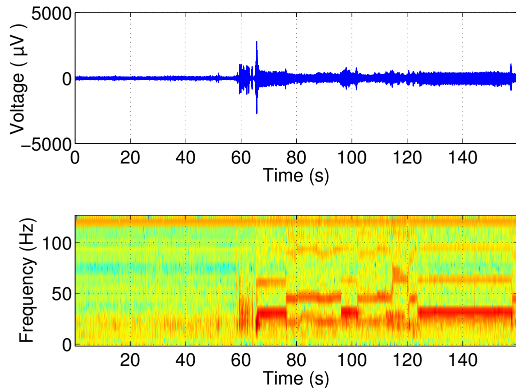

Fig. 5: We simulated the five lossy compression algorithms on a second real-world seismic data set collected from the IJkdijk test levee in the Netherlands. The top is example data plotted in the time domain and the bottom is in the time-frequency domain.

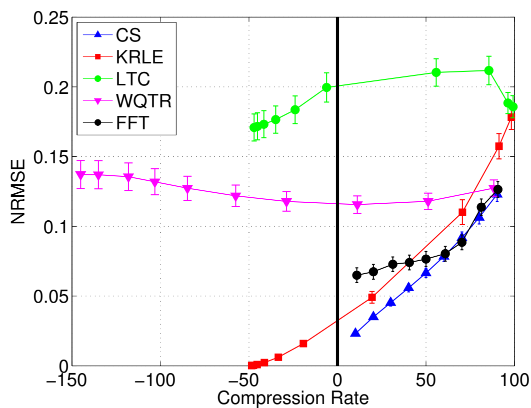

Fig. 6: Mean NRMSE results (with 95% confidence intervals) from five lossy compression algorithms simulated on the real-world IJkdijk seismic data set.

pattern recognition workflow to detect stages of levee failure (e.g., seepage, erosion, and collapse). Such a workflow would help us evaluate the relative performance of the five compression algorithms in regards to whether the recovered signals are really useful. We hypothesize that the recovered signal for CS is useful, however, as the NRMSE is quite low.

Another aspect worth noting between Figures 3 and 6 is that the mean compression rates of two adaptive compression algorithms (KRLE and LTC) differed between signals. Note how much the mean compression rate (x axis) decreases between the avalanche and IJkdijk data sets; for example, the mean compression rate of KRLE with $K = 64$ (fourth from the right) decreased from 49.9% in the avalanche data set to 19.5% in the IJkdijk data set. Additionally, the mean compression rates for LTC decreased as well. In comparison, compression rates for FFT and CS did not change between data sets. In other words, once we pick M for CS and the low-pass frequency threshold for FFT, we can calculate what the compression rates will be. These results demonstrate how compression rates for some adaptive compression algorithms depend ultimately on the signals being compressed.

Herein lies an advantage for the non-adaptive algorithms over the adaptive ones; with FFT and CS, users can specify compression rates that will be guaranteed for the lifetime of the wireless mote. For KRLE, the most comparable performing lossy algorithm evaluated, users must select a value for parameter K and hope that compression rates will be nonnegative. Additionally, users must guard against selecting a K value that is “too big” for the target signal, which would lead to lost signal information.

Of course, as with the other parameterized lossy compression algorithms evaluated, both FFT and CS suffer from the same challenge: i.e., how to best select the ideal parameter to provide maximal compression without subjugating recovered signal quality. Despite the challenge of parameter selection, we argue that the non-adaptive algorithms are preferable to the adaptive algorithms because of the ability to precisely estimate compression rates, radio usage, and thus, power consumption. For the adaptive algorithms, how would users estimate the mote’s power requirements based on selecting, for example, $K = 64$ versus $K = 128$? With CS and FFT, users can estimate power requirements and thus, system cost, with a very high degree of confidence. In other words, CS and FFT users do not have to pad power requirements to guard against extra power consumption that occurs from possible negative compression rates. We investigate this issue in detail in the next section.

IV. Hardware Implementation

To further evaluate the five lossy compression methods, we implemented the algorithms on a low-cost Arduino Fio wireless mote platform (2 kB RAM, 8 MHz CPU) with a long-range XBee Pro 802.15.4 radio module. Arduino Fio is our mote platform of choice because we are currently building high precision geophysical sensing “shields” that can plug and play with these low cost and easy to use platforms. Additionally, we used high power (1.5 km line-of-sight) radios to mimic a real-world wireless sensor deployment on a typical avalanche path or earth dam.

For repeatability, we tested the algorithms by compressing 36 seconds of real-world seismic data hard-coded in the mote’s FLASH memory. For our experiments, the 36 second test signal was synthesized from three short and low-noise slab avalanche events from the Swiss data set (Figure 7).

Similar to our simulation experiments in Section III, our hardware experimentation consisted of adjusting the compression algorithm parameters and evaluating the resulting compression rate, signal recovery error, and power consumption. As discussed in Section III, we implemented a two-buffer modality for all algorithms (including CS); while one buffer was being acquired from FLASH memory, the other buffer was getting compressed and transmitted. To minimize radio power consumption, we put the XBee Pro radio in a low-power sleep state and further buffered the transmissions into 100-byte payloads, the maximum size of the XBee Pro radio’s packet payload. When the 100-byte payload buffer became full, we woke up the radio, transmitted the payload, and put the XBee Pro back to sleep.

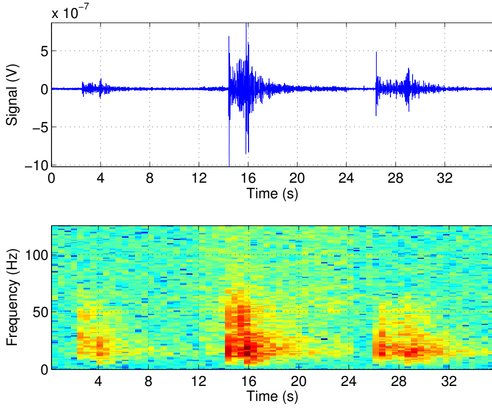

Fig. 7: For repeatability, we stored 36 seconds of synthesized seismic data in FLASH memory. The 36 seconds of data contain three slab avalanches. In the above figures of the time domain (top) and time-frequency domain (bottom), the avalanches begin at approximately three seconds, 14 seconds, and 26 seconds, respectively.

A. Recovered Signal Errors

We compressed the 36 second test signal stored in FLASH using each of the five algorithms with the same parameters as our simulations; see Section III for algorithm parameter values. Compression rates were calculated based on the size, in bytes, of the compressed versus original signals received. Signal recovery was performed offline with a combination of Python, C++, and Matlab using techniques summarized in Section III-B. The solid shapes in Figure 8 plot the NRMSE versus mean compression rate of the five parameterized algorithms executed in hardware on the real-world test signal. The white-filled shapes in Figure 8 represent the NRMSE from simulated compression on the exact same 36 seconds of data.

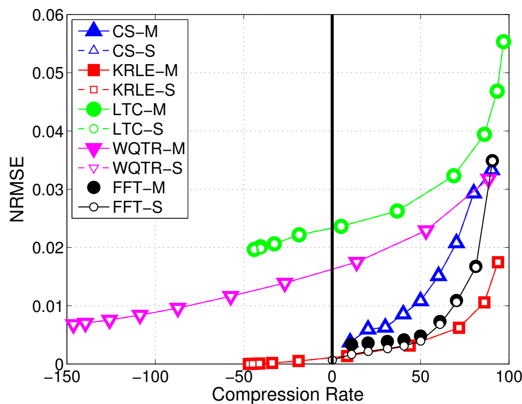

Fig. 8: The NRMSE of the recovered signal compressed on mote (solid shapes) and in simulation (white-filled shapes) on 36 seconds of seismic data containing three slab avalanches.

The most notable trend in Figure 8 is that, for all but FFT, the NRMSE rates for signals recovered from our hardware implementations (solid shapes) and simulated compression (white-filled shapes) were identical. These results help validate the credibility of our simulation experiments in Section III by showing that a signal compressed in hardware is equivalent to the same signal being compressed in simulation. Despite our efforts to simulate the mote's limited precision for computing the FFT, there were small differences in the NRMSE results for the simulated versus on-mote FFT compression. We hypothesize that the simulated FFT performs better because it was implemented in Matlab on a 64-bit computer (with 64-bit computation); the on-mote version, on the other hand, was implemented in assembly on an 8-bit microcontroller.

Furthermore, because we simulated compression on a sizable 2.75 hours of seismic data in Section III, the NRMSE results presented in Figure 8 should be observed with caution. In other words, it would be naive to conclude that KRLE is the best performing lossy compression algorithm for seismic data (in terms of NRMSE) due to the small, low variability 36-second data set used for the simulation of Figure 8. Instead, we refer the reader to Figures 3 and 6, which show results from 2.75 hours and 50 minutes of seismic data, respectively; as shown in these two figures, CS was either the best or tied for the best performing algorithm (in terms of lowest NRMSE). We hypothesize that KRLE performed best on the small 36-second data set because the signal contained very little variability and noise events. This lack of high signal variability is unlike the large 2.75 hour data set used in simulation, which contains background noise events caused by helicopters, airplanes, wind, ski lifts, etc.

B. Power Analysis

Lastly, we analyzed the power consumption of the wireless mote as it executed the five compression algorithms and compared the results to full sampling. Specifically, we measured the voltage difference across a 10.1Ω resistor in series from the mote to ground, then derived the current draw using Ohm's law. All voltages were measured at 50,000 Hz sampling rate using a 16-bit precision National Instruments USB-6218 DAQ and LabView SignalExpress. To reiterate, we used an Arduino Fio wireless mote with a long-range XBee Pro 802.15.4 radio module (1.5 km line-of-sight range). From our power analysis, we then estimated the longevity of a reasonably sized battery used to power an Arduino Fio wireless mote running these algorithms. Figure 9 depicts the estimated longevity of a 6600 mAh battery in ideal conditions. The dashed line shows the "benchmark" battery life of full sampling (i.e., the original signal with 0% compression rate).

Since our power analysis was based on compressing and transmitting a short, 36-second signal stored in FLASH, the battery longevity results should also be observed with some caution. Put another way, it would be naive to conclude that LTC will *always* consume the least power because it had the highest estimated battery longevity (i.e., 1208 hours of battery life at 96.9% compression rate). Additional power analyses on much larger data sets would be needed to verify such findings. Despite the limited size and duration of our power analysis, however, we are confident that we can draw veritable conclusions regarding the relative power consumption of the adaptive versus non-adaptive algorithms.

The most notable and conclusive trend in Figure 9 is how costly negative compression can be. Adaptive algorithms KRLE, LTC, and WQTR all had instances where negative compression rates resulted in power consumption greater than

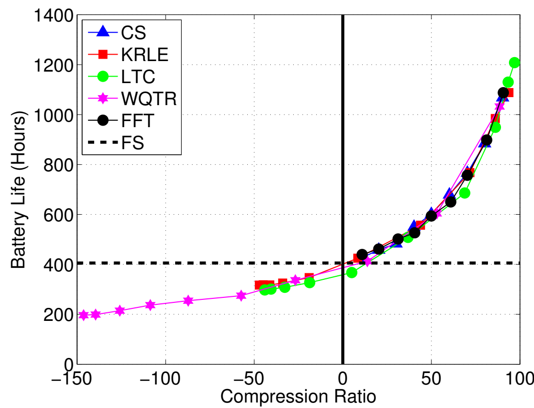

Fig. 9: The estimated longevity of a 6600 mAh battery used to power a wireless mote running each algorithm.

that of full sampling. In other words, without careful parameter selection in relation to the target signal, KRLE, LTC, and WQTR may drain the battery faster than full sampling. On a different note, the non-adaptive compression algorithms (i.e., CS and FFT) were the only methods with battery longevities exclusively above full sampling. Herein lies the advantage of CS and FFT: nonnegative compression rates and resultant power savings can be guaranteed.

Although CS and FFT both showed exclusively nonnegative compression rates, CS improves upon FFT by performing better in terms of avalanche event classification accuracies (Figure 4), implying that CS recovers more useful information in the decompressed signal. Moreover, CS is advantageous because it can be implemented without sampling the entire (full) signal before compression; such an approach is particularly useful when data acquisition is expensive (e.g., powered sensor or power hungry ADC).

V. Conclusion

In this article we rigorously compared CS to four lossy on-mote compression algorithms found in the literature. Through simulation on two real-world seismic data sets, we show that CS performs comparably to other on-mote, lossy compression algorithms. Additionally, we evaluated our implementation of the five lossy compression algorithms on real hardware in terms of compression rates, recovery error, and power consumption. Our results show that CS, a lightweight and non-adaptive compression algorithm, performs favorably compared to KRLE, LTC, WQTR, and FFT. Specifically, CS can guarantee positive compression rates and reduced power consumption without sacrificing signal recovery error. Such results are promising, suggesting that CS, a non-adaptive compression algorithm with very little compression overhead, can compete with other, state of the art wireless sensor node compression algorithms. Furthermore, our results suggest that CS improves upon FFT in terms of the information recovered; although FFT had slightly better NRMSE results when simulated on the avalanche data, CS had less variable classification accuracies (in general).

For future work, we plan to conduct more fine-grained power analyses and create a detailed consumption model for the Arduino Fio wireless mote with XBee Pro radio. Additionally, we plan to publish all source code used in our experiments, providing easy to use lossy compression algorithms for the wireless sensor network community.

Acknowledgment

This work is supported in part by NSF Grants DGE-0801692 and OISE-1243539. We thank Dr. Alec van Herwijnen for collecting and sharing the avalanche data. We also thank Justin Rittgers, Minal Parekh, Ben Lowry, Jacob Grasmick, and Dr. Mike Mooney for collecting and sharing the IJkdijk data.

References

[1] N. Kimura and S. Latifi, "A survey of data compression in wireless sensor networks," *Proceedings of the International Conference on Information Technology: Coding and Computing*, pp. 8–13, 2005.

[2] E. Capo-Chichi, H. Guyennet, and J. Friedt, "K-RLE: A new data compression algorithm for wireless sensor network," *IEEE International Conference on Sensor Technologies and Applications*, pp. 502–507, 2009.

[3] T. Schoellhammer, E. Osterweil, B. Greenstein, M. Wimbrow, and D. Estrin, "Lightweight temporal compression of microclimate datasets," *IEEE International Conference on Local Computer Networks*, pp. 516–524, 2004.

[4] N. Xu, S. Rangwala, K. Chintalapudi, D. Ganesan, A. Broad, R. Govindan, and D. Estrin, "A wireless sensor network for structural monitoring," *ACM Conference on Embedded Networked Sensor Systems*, pp. 13–24, 2004.

[5] S. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, 1997.

[6] M. Rubin and T. Camp, "On-mote compressive sampling to reduce power consumption for wireless sensors," *IEEE International Conference on Sensing, Communication, and Networking (SECON)*, 2013.

[7] Z. Charbiwala, Y. Kim, S. Zahedi, J. Friedman, and M. B. Srivastava, "Energy efficient sampling for event detection in wireless sensor networks," *ACM/IEEE International Symposium on Low Power Electronics and Design*, pp. 419–424, 2009.

[8] H. Mamaghanian, N. Khaled, D. Atienza, and P. Vandergheynst, "Compressed sensing for real-time energy-efficient ECG compression on wireless body sensor nodes," *IEEE Transactions on Biomedical Engineering*, vol. 58, no. 9, pp. 2456–2466, 2011.

[9] T. Srisooksai, K. Keamarungsi, P. Lamsrichan, and K. Araki, "Practical data compression in wireless sensor networks: A survey," *Journal of Network and Computer Applications*, vol. 35, pp. 37–59, 2012.

[10] Open Music Labs, "Fast Fourier transform library." http://wiki.openmusiclabs.com/wiki/ArduinoFFT, 2014. Retrieved 04-10-2014.

[11] E. Candes and M. Wakin, "An introduction to compressive sampling," *IEEE Signal Processing Magazine*, vol. 25, pp. 21–30, March 2008.

[12] A. Herwijnen and J. Schweizer, "Monitoring avalanche activity using a seismic sensor," *Cold Regions Science and Technology*, vol. 69, no. 2-3, pp. 165–176, 2011.

[13] E. Candes, M. Wakin, and S. Boyd, "Enhancing sparsity by reweighted ℓ_1 minimization," *Journal of Fourier Analysis and Applications*, vol. 14, no. 5, pp. 877–905, 2008.

[14] M. Rubin, M. Wakin, and T. Camp, "Sensor node compressive sampling in wireless seismic sensor networks," *1st IEEE/ACM Workshop on Signal Processing Advances in Sensor Networks (SPASN)*, 2013.

[15] M. Rubin, T. Camp, A. Herwijnen, and J. Schweizer, "Automatically detecting avalanche events in passive seismic data," *IEEE International Conference on Machine Learning and Applications*, pp. 13–20, 2012.