Data Compression of Neural Signaling in an Implantable Microchip

Matthew Richardson
University of Washington Department of Computer Science
Tech Report #02-08-01

ABSTRACT

Intracellular neural recording is a technique that has been under development for many years, yet still remains fragile and difficult, and requires immobilization of the neurons being recorded. We are developing a system of implantable silicon chips that will intracellularly record, and locally store, the neural signaling. The principal reason we are doing this is to record from intact, mobile animals. We determined that data storage would be one of the major limiting factors in the implantable electronics. In this paper, I present an algorithm for data compression of intracellular neuron recordings that accomplishes a compression ratio of over 800:1 (0.01 bits/sample at a 2.5Khz sampling rate) with acceptable signal error (approximately 8% PRD). At this rate, we can record from ten intracellular electrodes for over 30 hours. The algorithm uses a combination of run length encoding and template matching.

INTRODUCTION

Recent advances in integrated circuit technology have opened new possibilities in the areas of neurobiology and electrophysiology. With modern MEMS (Micro-Electro-Mechanical Systems) technology, mechanical devices may be built on a micrometer scale. Modern microprocessors and microcontrollers can be small enough, and consume little enough power, that they may be surgically implanted and run independently for days. Advances in memory technology enable large amounts of memory to be implanted alongside the microchip, all in a miniature package that would have been unheard of a decade ago.

We believe that these advances in technology and miniaturization have opened new possibilities for connecting biological nervous tissue to implantable silicon devices that will be self-sufficient and self-contained, needing no external cabling or communication. Such devices will have the potential to greatly improve our understanding of how neurons compute and communicate. They will allow experiments that permit animal mobility, provide longer recordings than were previously possible, and may provide the ability to record from many more electrodes simultaneously than was previously possible. The ability to connect nervous tissue with silicon will give new tools to the fields of biology, cognitive science, electrophysiology, computer science, and medicine.

We propose to construct a system of intracellular recording chips and initially test the system on *Tritonia Diomedea*, a sea slug. *Tritonia* is an excellent candidate for initial tests because of its large, easily identifiable neurons. Furthermore, much work has been done on intracellular recording in *Tritonia*, providing a pool of experience and techniques in animal preparation, surgery, and experimentation. In examining the problem space for the implantable recording system, we identified several areas that needed further exploration and development. One of these areas was data storage.

For the implantable chip to operate independently from any host system, it must store data locally. *Tritonia* lives underwater, making wireless communication infeasible. Using estimates of chip size, we concluded that we could implant 32Mbits of memory.

Other limiting factors in the system include battery life, corrosion of connections, efficiency of the electrodes, cell and animal life expectancy, and electrode durability. We estimate that these factors should limit the recording time to no less than 24 hours.

With ten recording connections to nervous tissue, each recording at 2.5KHz with 8 bits of resolution per sample, the memory would be filled in 2.6 minutes. For this system to be useful, we desire the ability to record simultaneously from the ten connections for at least 24 hours. This equates to a compression ratio of over 500:1.

My search for a data compression algorithm was limited to those that can operate on-the-fly. Because only 2 minutes of raw data may be temporarily stored, the algorithm must perform compression in real time. Furthermore, the algorithm must be computationally simple in order to adhere to the limitation on the computational abilities of an implantable chip.

1. Background and Related Work

1.1 Compression Terminology

In data compression, an input sequence of symbols is converted to a new sequence that is shorter than the original. Throughout this paper, I use u(t) to represent the original sequence of data. When the compressed sequence is uncompressed, the resulting sequence is referred to as the *reconstructed* signal, and is assigned the symbol v(t). The ratio of the size of the uncompressed sequence to the compressed sequence is the *compression ratio*.

Data compression can be either *lossy* or *lossless*. In *lossless* data compression, the reconstructed signal is identical to the original signal: $\forall t$: u(t) = v(t). In *lossy* data compression, the reconstructed signal is an approximation to the original signal, the difference usually determined by an error metric and threshold: $\forall t$: u(t)=v(t)+Err(t). Lossy data compression can always achieve a compression ratio greater than or equal to that of lossless data compression.

1.2 Entropy Encoding

Lossless compression at near-theoretical minimum entropy rates can be achieved by the use of standard techniques such as Huffman or Arithmetic encoding [1]. These techniques *entropy encode* a signal at or near the theoretical minimum *zero-order entropy* defined by the equation

$$H = -\sum_i p_i \log_2 p_i$$

where p_i is the probability of occurrence of symbol i. *Zero-order entropy* assumes that symbol probabilities are independent.

One technique for reducing the entropy is to *difference encode* the signal. In *difference encoding*, a new sequence is constructed: u'(t) = u(t)-u(t-1). In slow moving data difference encoding will often drastically reduce the entropy. The reduction comes from concentrating the probability distribution into a smaller area that is sharply peaked around 0.

Another technique for reducing the entropy is to consider *first-order* entropy. The *zero-order entropy* assumes that each sample is independent. In *first-order* entropy, probabilities are conditional on the previous value. If we let P^j be the probability distribution of u(t), given that u(t-1)=j, then the first-order entropy is defined as:

$$H = -\sum_j P^j H(P^j)$$

To implement first-order Huffman coding, we would have a separate Huffman code for each p_j. Because this algorithm is $O(n^2)$ in the number of symbols, ideal first-order Huffman coding is memory intensive, but it does provide a lower bound measure of entropy, and there are techniques that can be used to more efficiently encode at rates approaching the first-order entropy.

I performed entropy calculations on a difference encoding of the data (Section 2 provides information about the data used in these experiments) to estimate the compressibility by a standard encoding scheme such as Huffman coding. Table 1 summarizes the results. To accomplish our goal, the bit rate must be reduced to less than 0.015 bits/sample. From the table it is apparent that simple zero- or first-order entropy compression will not accomplish our goal.

	Zero-Order Entropy	First-Order Entropy
Spike Areas	3.4	2.07
Non-Spike Areas	0.591	0.557

Table 1: Measured data entropy (in bits/sample)

1.3 Standard Data Compression methods

Many standard methods for lossless compression of generic data have been developed. LZW compression [2], a standard compression algorithm, compresses a difference encoding of the original signal to 0.56 bits/sample. This is slightly better than the zero-order entropy of 0.59 bits/sample, but far from the 0.015 bits/sample required to accomplish our goal (Note that the LZW algorithm can compress at better than *zero-order entropy* rates because it does not make the independence assumption). Other standard methods gave similar results.

1.4 ECG Compression techniques

Much work has been done in compressing of electrocardiogram (ECG) signals, which have many characteristics similar to the neural signals that we are compressing. A review of many ECG compression algorithms was done by Jalaleddine et. al. [3].

One popular technique is the Scan Along Polygon Approximation (SAPA) [4] algorithm, which approximates u(t) with a polygon. Similar in style to SAPA is the Turning-Point technique, which stores only the peaks and valleys of u(t) [5]. In our neural data, the spike shape typically remains fairly constant throughout the recording. An algorithm that takes advantage of this similarity should provide higher compression rates and/or less error than a technique like those above in which each spike is treated independently. Consequently, I did not consider either SAPA or TP.

In 1993, Uchiyama et. al [6] examined ECG compression by template matching. Templates were SAPA approximations of the spikes in the original data. Because they desired lossless compression, the difference between the original data and templates was encoded using an entropy encoding scheme. The result was a compression ratio approximately equal to that of a direct entropy encoding.

Standard methods such as LZW or entropy encoding, as well as lossless ECG methods such as the one described above, do not achieve the necessary compression ratio of over 500:1. Consequently, I have developed a lossy compression algorithm, tailored to the neural data from *Tritonia*, that uses a combination of templating and run length encoding and achieves a compression ratio of over 800:1.

2. DATA AND OVERVIEW

I obtained four sample recordings from representative neurons in *Tritonia*. Throughout the paper, I will refer to channel 0 of the data as *dev-data*. This is the data being discussed when it is not otherwise specified. I will refer to channels 1,2, and 3 collectively as *eval-data*. I used *dev-data* for all algorithm development and experimentation, and reserved *eval-data* for final algorithm evaluation (see table 5). Each channel is recorded at 2500 samples/sec, 8 bits per sample, for 31679106 samples (3.5 hours).

Refer to figure 1 for a subset of the data. Over 99% of the data is essentially flat, punctuated occasionally by a neural impulse, or *spike*. I refer to the flat areas as *nonspike areas* and the neural signaling areas as *spike* areas.

The time between spikes, and the shape of the spikes, are the most important features of the data [12]. The data between spikes is also useful (to a lesser extent), especially the low frequency components, which contain baseline information and possibly information about the inhibition or excitation of the neuron.

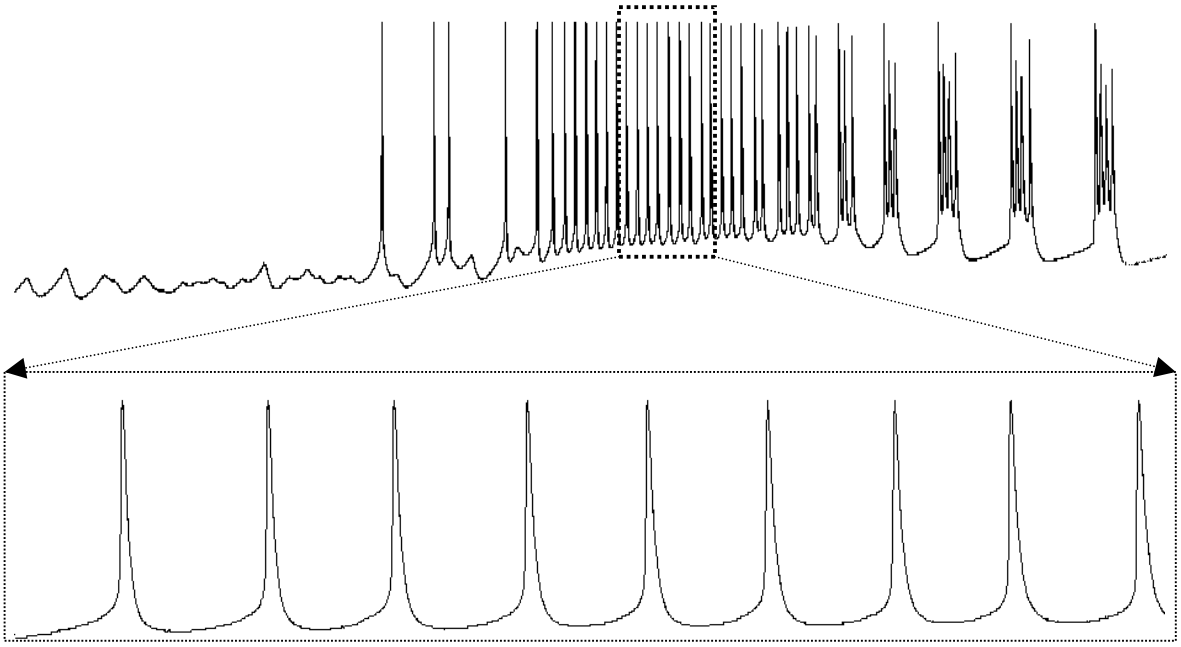


Figure 1: A subset of the data shown at two different magnifications. The upper subset is approximately 10 seconds (25000 samples). The lower subset is approximately 0.84 seconds (2100 samples)

Knowledge of the important aspects of the data, and a need for high compression ratios, prompted me to explore domain-specific techniques that would perform lossy compression in a manner that preserves the most important features of the data. To fully take advantage of spike similarities, I use a templating scheme for compression of *spike* areas. The algorithm uses previously encountered spikes as templates for future spikes. For *nonspike* areas, I use run length encoding, augmented with lossy techniques that extend the run lengths. In section 3 I introduce spike templating. Section 4 discusses run length encoding. In section 5, I discuss the results and give examples of the compressed data. Section 6 estimates the computational requirements of the algorithm. Section 7 comprises future work and conclusions.

3. SPIKE TEMPLATING

Observation of the data reveals that different spikes often have very similar shapes. To take advantage of this, I explored a templating compression scheme whereby each spike is represented not by a sequence of points but rather by a reference to a previous spike which looks similar. I define the *instantiation* of a template to be the insertion of a template index, along with other required parameters into the compressed data stream, in order to represent the occurrence of a spike in the original data. Below is the algorithm for spike templating:

Detect Spike S S_1=rising segment of S, S_2=falling segment of S T_1=FindNearestTemplate(S_1, Dictionary$_1$) T_2=FindNearestTemplate(S_2, Dictionary$_2$) Instantiate T_1:T_2	FindNearestTemplate(S, Dictionary) If $\exists$ T$\in$ Dictionary s.t. distance(S,T) $\leq \tau$ Return T Else Add S to Dictionary Return S

Notice that I split the detected spike, S, into two segments; the division occurs at the peak of the spike. This allows two spikes which have similar falling edges but different rising edges to use the same template for S_2. Interestingly, across a wide variety of error tolerances, there are one half to one third as many templates required for S_2 than there are for S_1. I don't know whether this is just for the data set I have or is a more general phenomenon about neurons that I can depend on in future data sets.

The resulting lossy compression ratio and quality depends strongly on the error threshold value τ, as well as other factors such as the similarity of spikes to one another.

3.1 Dictionary

In a real-time system such as the implantable microcomputer we are constructing, we must guarantee that the system will be able to compress the data at the input rate. To achieve this, I first introduce the concept of an active set of templates, called the *Dictionary*. The dictionary contains all templates which may be *instantiated* at any given moment. By limiting the number of active templates that must be searched, I can guarantee that the system will be able to compress spikes in real-time.

Note that the set of templates in the Dictionary is not the same as set of all templates which have been instantiated in the compressed data. In the case when the dictionary is full and a spike segment needs to be converted into a template and added to the dictionary, the *least-used* template currently in the dictionary is removed and the new spike segment takes its place. It is necessary to keep the template which was removed from the dictionary since it was referenced earlier. This template is stored in memory as part of the compressed data.

The dictionary may reduce the number of bits needed to index a template when the template is being instantiated. The dictionary limits the number of templates which can be indexed, which may result in a smaller index.

3.2 Spike Detection

A simple slope-thresholding algorithm is employed for spike detection. The following three conditions must be consecutively satisfied (κ and γ are user-defined constants):

1. $\left|\frac{u(i)-u(i-\kappa)}{\kappa}\right| < \gamma.$ Let $\sigma = sign[u(i)-u(i-\kappa)]$
2. $sign[u(i)-u(i-\kappa)] = -\sigma$
3. $\left|\frac{u(i)-u(i-\kappa)}{\kappa}\right| < \gamma$

Fulfillment of all three steps in the given order constitutes the recognition of a spike, which begins at u(i-κ) when condition 1 was satisfied, and ends at u(i) when condition 3 is satisfied. The window size parameter, κ, helps ensure that impulse noise is not recognized as a spike. The slope threshold parameter, γ, allows the user to adjust where spikes begin and end, and how sharply the data must move to be classified as a spike.

3.3 Template Instantiation and Parameters

To reduce the number of templates, I *parameterize* templates by certain factors, and store these parameters as part of the instantiation. In adding template parameters, I tried to balance many factors, including: increase in computation, reduction in the number of templates, added certainties in the compressed data, and an increase in instantiation size. See figure 2 for a demonstration of the effect of adding parameters to the templates. The parameters are:

1. *Vertical Offset (Baseline)*: Templates are shifted vertically to match the offset of the spike segment This allows two spike segments at different vertical offsets to use the same template.
2. *Width*: Templates are scaled in width to fit the width of the spike segment S
3. *Height*: Templates are scaled in height to fit the height of the spike segment S

Instantiation of the template requires: (1) An index into the current dictionary, (2) a time since the last template, and (3) all parameters used in shifting and scaling the template to match the spike. Figure 3 gives an illustration of all of the instantiation components.

Table 2 gives the minimum entropy for each of the instantiation components illustrated in figure 3. The column *Estimated Bits* in Table 2 gives the number of bits used in estimating the size of an instantiation. An estimation is necessary because the algorithm does not currently actually entropy encode instances. I need an estimate of the size of the encoding so I can provide reasonably accurate results. Note I am using an error margin of over 10% so all results are conservative rather than optimistic (a typical encoding is within 1-3% of the ideal entropy).

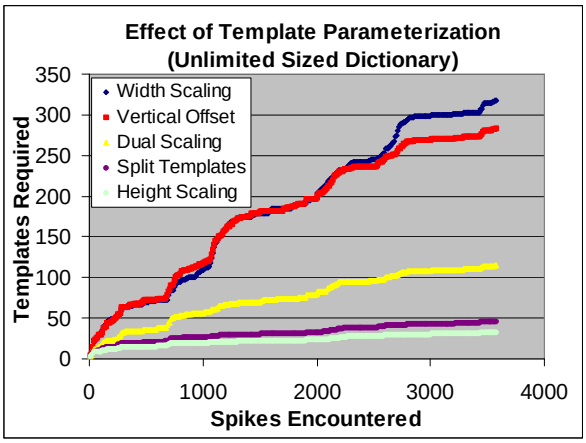


Figure 2: Reducing dictionary growth by using template parameters. Each category is additive — it consists of itself plus all prior categories. Because "Split Templates" results in templates which are approximately half the size, the last two categories represent half the actual number of templates required in order to provide more meaningful comparison with the previous categories.

The *reconstructed signal* contains the following information:

1. The exact time of the peak of the spike (accurate to within the sampling period)
2. The exact height of the spike (accurate within the sampling precision)
3. The exact width of the spike
4. An estimate of the shape of the spike

Some parameters, such as spike height, provide limited reduction in the number of templates required. However, by including the height as a parameter, the user knows the true height of the spike rather than only knowing an estimate. It is useful for the compressed data to contain some lossless parameters which give the user some certainties that the user may know rather than all components of the data being lossy approximations.

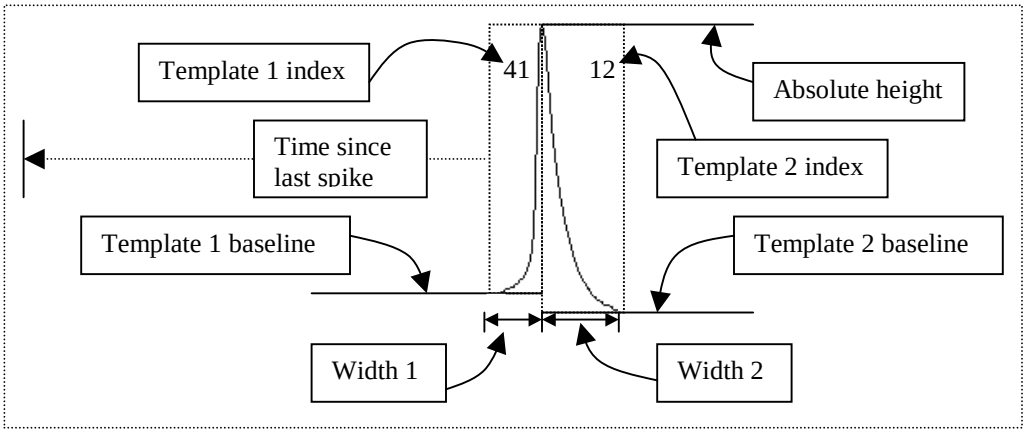


Figure 3: Template instantiation parameters

Category	Entropy	Estimated Bits	Comment
time	12.27	14	Time expired since last spike. *Diff Encoding* reduces to 11.82
baseline1	0	0	Same information is contained in non-spike data compression
baseline2	0	0	Same information is contained in non-spike data compression
width 1	4.65	6	
width 2	5.86	6	Reduces to 5.24 bits if width is difference encoded
index 1	6	6	Dictionary is 64 entries
index 2	5	5	Dictionary is 32 entries
height	3.64	4	Entropy based on difference encoding the spike peak values
Total	37.42	41	Error Margin is over 10%

Table 2: Template instantiation parameter entropy and bits used in results

3.4 Distance

Let T' be the scaled and shifted version of T according to the parameters as defined in section 3.3. Both S and T' have N data points. I define the distance between S and T' by the mean square error:

$$Error(S,T') = \frac{1}{N}\sum_{i=0}^{N-1}(S[i]-T'[i])^2$$

4. ENCODING THE NON-SPIKE AREAS

The non-spike areas fluctuate slowly. In a difference encoding, over 90% of the non-spike data is 0. To take advantage of long sequences (runs) of one value, I use a variation of run length encoding (RLE). Standard RLE replaces a long constant sequence with: (1) a special symbol signifying the start of a run, (2) a symbol for the length of the run, and (3) a symbol for the bit value. In my variation of RLE, the compressed data contains only runs, removing the need for a special symbol. The compressed output of the algorithm is:

[length, delta] [length, delta] [length, delta]…

Where *length* is the length of the run, and *delta* is the difference between the value of this run and the value of the next run. The algorithm assumes an initial value of 0. Note that because runs are always extended until the value changes, there will never be a *delta* of 0 in the RLE compressed data.

A lossless encoding of the non-spike areas using RLE results in a compression ratio of (vs. raw 8 bit 2500hz sampled data) of 19.9, assuming perfect entropy encoding of the lengths and deltas. This corresponds to a bit rate of 0.40 bits/sample. RLE can encode at bit rates less than the zero-order entropy because it does not make the assumption that each sample is independent. In fact, RLE is most effective if each symbol is very dependent (i.e. the same) as the previous symbol. To raise the compression ratio to over 500:1, required to accomplish our goal, the algorithm must perform lossy RLE.

4.1 Lossy RLE

I use three techniques to improve the compression ratio in the non-spike area. First, I reduce the precision of the non-spike data to 7 bits. This allows for longer runs and reduces the entropy of the RLE deltas. Second, I allow runs to ignore short-term deviations. The algorithm ignores any deviation from the value of the current run that returns within α samples. This allows much longer runs by ignoring small (presumably meaningless) noise. Third, the algorithm downsamples the non-spike data to 250Hz (A reduction by a factor of 10). If run lengths were uniformly distributed, this would result in an entropy decrease of $\log_2 10$ bits, since every 10 lengths are decimated to one length. The actual decrease on *dev-data* is 3.0 bits per run length. Downsampling to 250Hz results in a 21% improvement in the overall compression ratio with an almost unnoticeable degradation in quality. An FFT of the non-spike data (see figure 4) demonstrates that the majority of the signal power is less than 125Hz, which is still fully captured by sampling at 250Hz.

The algorithm downsamples simply by observing every 10^{th} point, rather than resampling with a low-pass filter. Filtering is computationally expensive and I believe it would make little difference to the final compressed data quality or size, though I have not run experiments to verify this.

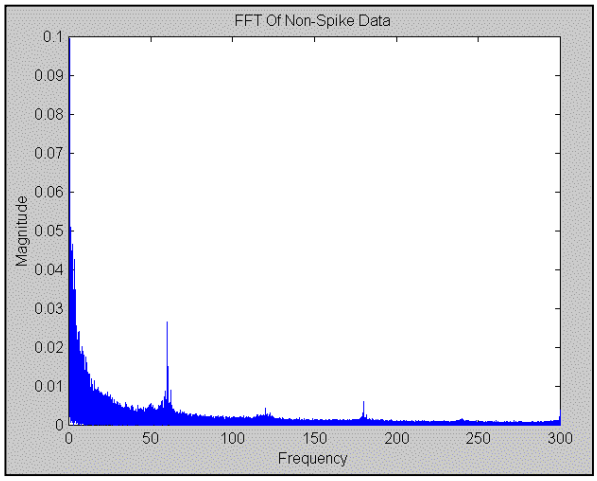


Figure 4: FFT of the first 2 million samples of the non-spike portion of dev-data.

4.2 Entropy Encoding

I encode RLE *lengths* and *deltas* using Huffman coding. The Huffman codes are based on probability distribution statistics gathered from *dev-data*, and are hard-coded in the algorithm.

Because the *lengths* can take on a wide range of values (0–32000 for *dev-data*) it is infeasible for me to directly encode the distribution. Such an encoding would be large and too data-set–specific. Instead, RLE *lengths* are categorized by how many bits are required to represent them (e.g. a *length* of 27 requires 5 bits). The compression algorithm uses Huffman coding to specify the size of the *length*, and then appends the value of the *length* (minus the first 1 bit). Table 3b gives the exact encoding used for RLE *lengths*. For example, a *length* of 27 (11011) would be stored as 111 (the 5-bit length category) concatenated with 1011 (the binary value of 27 minus the first 1), or "1111011". The decompression algorithm reads "111", and knows to read the following 4 bits (1011)and prepend a binary 1, resulting in 11011 (27).

RLE *deltas* have a much more limited distribution (over 99% are between –5 and 5), so they are directly Huffman coded, as shown in Table 3a.

The encoding of RLE *lengths* results in a bit rate that is only 0.9% worse than its ideal entropy. The encoding of RLE *deltas* is 11% worse than its entropy. This inefficiency in the RLE *deltas* encoding is due to their sharply peaked probability distribution; over 88% of the RLE *deltas* are +1 or -1 (see figure 5).

Value	code	bits
-5	11110001	8
-4	11110000	8
-3	111101	6
-2	110	3
-1	10	2
0	unused	0
1	0	1
2	1110	4
3	111110	6
4	1111110	7
5	1111001	7
other	1111111+delta	15

Table 3a: Huffman codes for RLE deltas

Value	code
0	010
1	011
2	1001
3	1010
3-bit	110+2bit length
4-bit	001+3bit length
5-bit	111+4bit …
6-bit	0000+5bit
7-bit	0001+6bit
8-bit	1011+7bit
9-bit	10001+8bit
10-bit	1000001+9bit
11-bit	10000001+10bit
12-bit	100000001+11bit
…	…
0Δ spike	100001

Table 3b:Huffman codes for RLE lengths

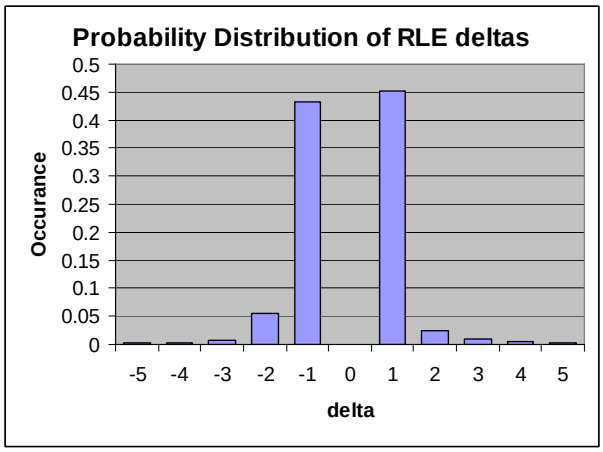


Figure 5: Probability Distribution of the RLE deltas

5. Results and Discussion

5.1 Reconstructed Data Quality

In the evaluation of lossy data compression algorithms, we must examine the quality of the reconstruction. The most credible method for this evaluation on neural data is by visual verification. Though qualitative and non-repeatable, visual verification is necessary because no statistic can capture the qualitative factors as the eyes of a trained neurobiologist.

I have applied my data compression to the data set, and had the reconstruction reviewed by electrophysiologists who would normally use that data in analysis of neuron-neuron interactions, and brain control of behavior. They reported that the compression preserves most useful aspects of the data[7][8]. See figures 6a and 6b for samples of the original and reconstructed data.

Visual evaluation is not an objective, quantitative measurement that I can use in comparing algorithms and techniques. A quantitative evaluation used in the ECG studies is an automatic diagnosis program, which diagnoses heart conditions based on the ECG signal[9]. The diagnoses on the original and reconstructed data are compared to ensure that the data compression does not change the diagnosis. This method is not available for neuron data recordings.

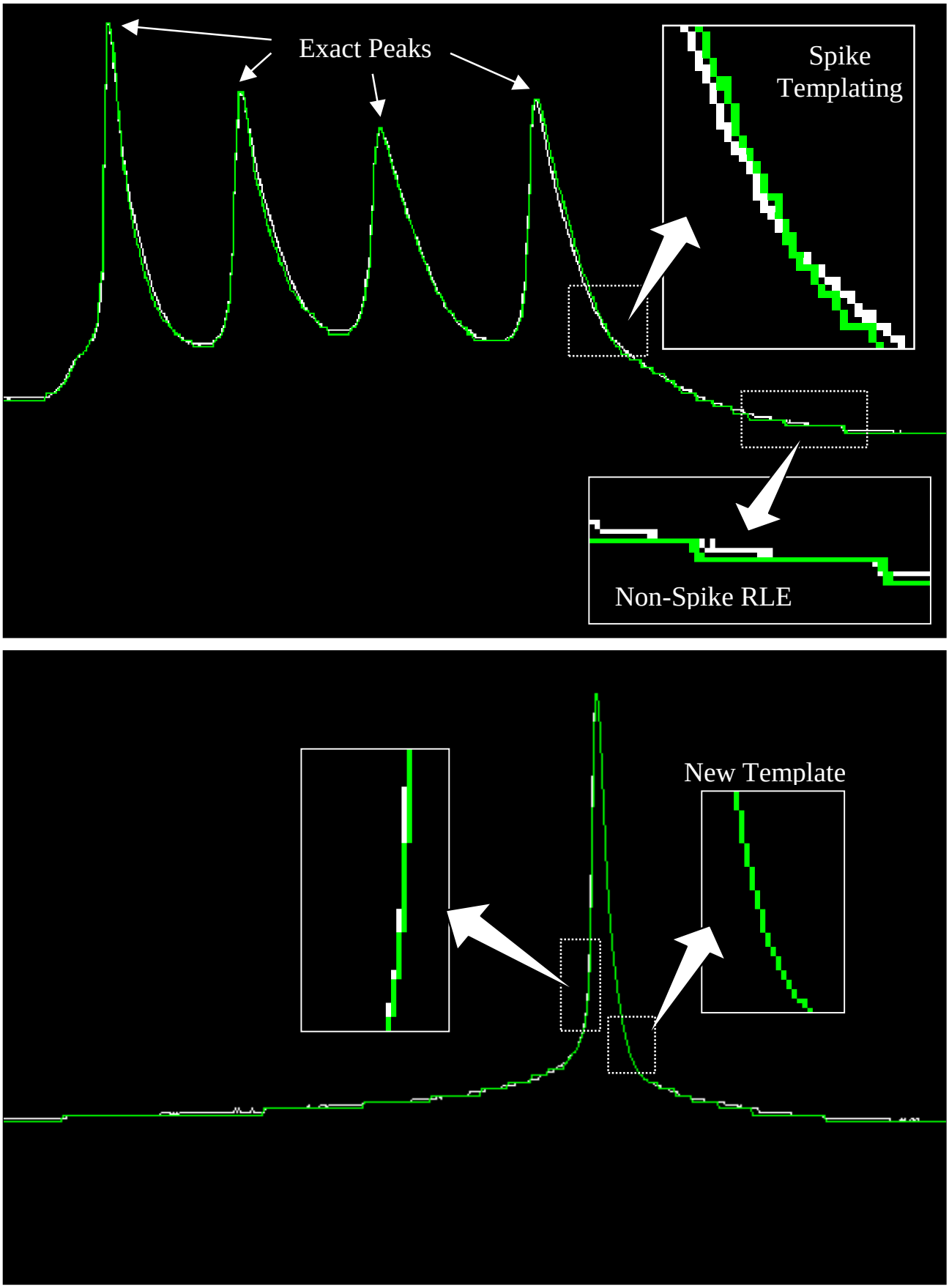
Instead, I use a purely statistical evaluation method. One common measure is the "percent root mean square difference" (PRD) [10], which measures the root mean square error as a percentage of the root mean square value. The PRD is calculated by the formula:

$$PRD = \frac{\sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(u(i)-v(i)\right)^2}}{\sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(u(i)\right)^2}} \cdot 100$$

The PRD is sensitive to a constant vertical offset. If the baseline of the data is not centered at 0 then the PRD will be artificially reduced due to an incorrect inflation of the denominator. To counter this problem when measuring the PRD of the spike areas, I individually shift each spike vertically so that its base value is 0.

Refer to Figure 7a for a graph of how the PRD of the *spike area* affects the compression ratio of the *spike area*. Figure 7b shows how the PRD of the *spike area* affects the overall compression ratio (recall the spike areas are only one part of the data compression). At low error tolerance levels, the algorithm must add many templates to the dictionary, so a larger dictionary results in a higher compression ratio. In contrast, at high error tolerance levels, a smaller dictionary can hold all the templates, and each instantiation requires a smaller index into the dictionary (as discussed in section 3.1) so a smaller dictionary produces a higher compression ratio.

I believe these results are better than would have been achieved by the SAPA algorithm. Takahashi shows an improvement to SAPA giving worse compression at a worse PRD level [9]. However, without running SAPA on this same data set I cannot perform accurate comparisons.



Figures 6a and 6b: These images demonstrate the quality of the reconstruction. The reconstructed data is shaded green. The original data is white (visible in areas where it differs from the reconstructed data). Note how spike height parameterization results in exact peak matching and that different spike shapes are adequately represented by the templates. The exact match on the falling segment of the spike in Figure 6b is due to the construction of a new template from that segment.

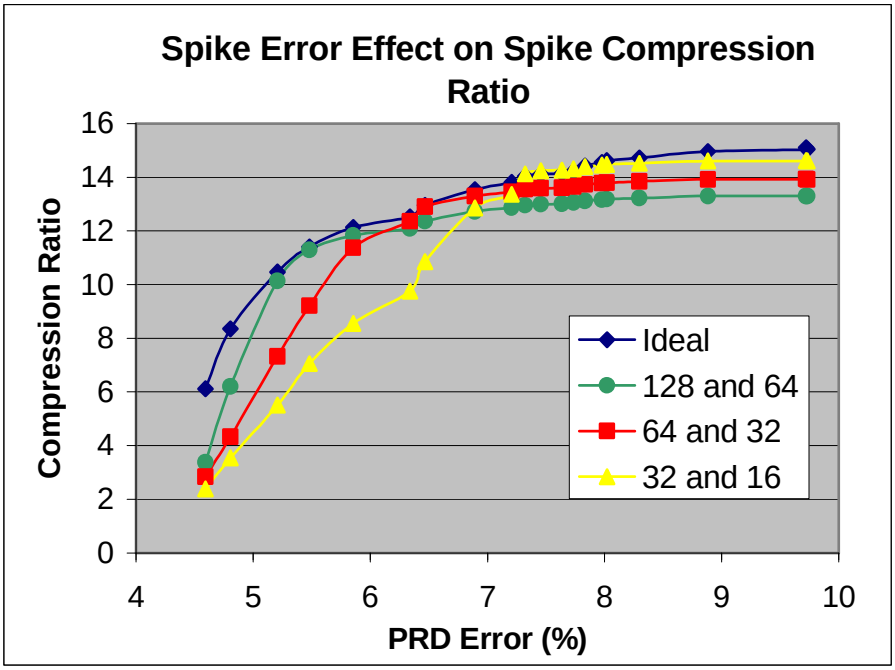


*Figure 7a: Compression ratio of spike data vs. the PRD Error of the reconstruction. The different lines represent different dictionary sizes, where "***m and n***" refer to a maximum dictionary size of* **m** *for the rising segment of the spike, and a maximum dictionary size of* **n** *for the falling spike segment.* **Ideal** *is the result if the dictionaries are the exact size necessary to contain all of the templates.*

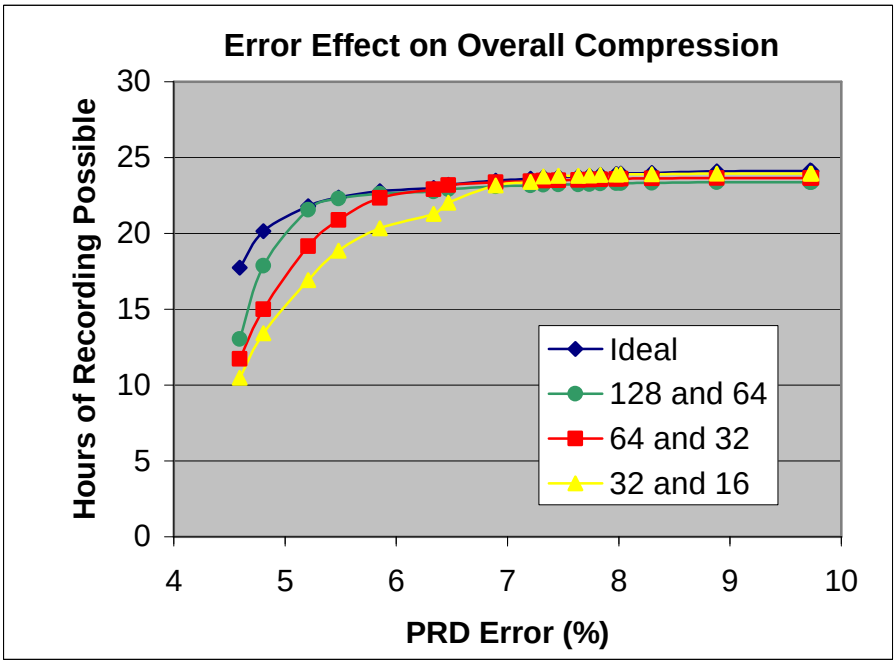


Figure 7b: Recording time to fill the memory (for ten probes) vs. the PRD error of the reconstructed spike data. The difference lines represent difference dictionary sizes, as in figure 7a.

5.2 Evaluation of the Compression Algorithm

I have two independent data sets, *dev-data* (channel 0) and *eval-data* (channels 1-3). I did all development work on *dev-data* so that when I apply the compression algorithm to *eval-data*, I am accurately testing its performance and generalizability. A summary of per-channel results is shown in table 4. Table 6 shows a detailed summary of the results on *eval-data*. Recall that the category *template instancing* is an estimated size, based on the entropy of the components plus a 10% error margin to be conservative. Likewise, I estimate 4 bits per sample for *template storage*, giving an error margin of roughly 50% (entropy is 2-3 bits/sample). The Parameters for spike compression (reviewed in Table 5) are: τ=15, κ=8, γ=0.6, size of Dictionary$_1$ is 64, and size of Dictionary$_2$ is 32. Run length encoding parameters for the non-spike areas are: Record every 10th point, α=4, with 7 bit data samples.

Channel	*0*	*1*	*2*	*3*
Spikes	2920	1750	1699	3236
CR	527	990	737	721
Hours at this rate for ten probes	**23.4**	**44.0**	**32.7**	**32.1**

Table 4: Channel compression rates

Param	*Description*	*Section*
τ	Template error tolerance level	3
κ	Spike detection window size	3.2
γ	Spike detection slope threshold	3.2
α	Maximum length of short-term run-length deviation to ignore	4.1

Table 5: User-specified algorithm parameters

Category	Average *eval-data* results	
Number of Spikes	2228	
Templates 1	38	
Templates 2	28	
Spike PRD	8.70%	
Total Template length (samples)	1997	
Hours at this rate for ten probes	**36.3**	
	(KBits)	
RLE deltas	57	17.5%
RLE lengths	169	51.6%
Template Storage	8	2.5%
Template Instancing (estimated)	93	28.4%
Total	327	100.0%

Table 6: Selected statistics averaged across eval-data channels

Notice that over 70% of the compressed data size derives from the non-spike areas. I can reduce this by decreasing the quality of the non-spike reconstruction. The electrophysiologists who reviewed the algorithm stated that they would be willing to sacrifice quality in the non-spike areas to allow longer recording times. In the limit, eliminating the non-spike data would allow recordings for over 100 hours.

6. Computational Requirements

6.1 Templating

To verify that I can implement this compression algorithm on a low power microchip, I estimated the computational requirement of the algorithm. Spikes may occur at up to 30 times/second, though the average rate is only one every 6 seconds. An 80Kbit input data buffer will reduce the peak rate at which the algorithm must compress spikes to only 2.5 spikes/sec. Given reasonable assumptions[1], the microchip must run at approximately 2Mhz to prevent the

[1] I assume a multiply takes 12 clocks and an add or subtract takes 2 clocks. Each spike must be compared point-for-point with each template in the dictionary, and each point has been scaled both in height and width (two multiplies), and the error (one subtract) is squared (one multiply), resulting in approximately 40 clocks per point per template, not including control logic or memory load time.

buffer from overflowing. This does not include the cost of control logic or memory fetching. I expect a 5Mhz processor to have the computational power necessary to process and compress the incoming spikes. I can reduce this requirement by using a smaller dictionary, a larger buffer, or an optimization which would reduce the amount of computation[2]. Such techniques can reduce the computational requirement to less than 1Mhz.

6.2 Run Length Encoding

Run length encoding must perform sample comparison and slope estimation, along with the occasional Huffman coding table lookup and other computations. I estimate the computational requirement to be less than 30 clocks per sample. Even at an exceptional 300 clocks per sample, the processor would only need to run at 0.75Mhz to encode the non-spike data.

Run length encoding and spike templating together will require a processor that runs at approximately 5-6Mhz in order to process and compress all of the incoming data, though I expect that with appropriate optimization and buffering, the algorithm could be run on a processor running at only 1Mhz.

7. FUTURE WORK

7.1 Algorithm Completion and Implementation

In the current work, the template instantiation sizes are only estimates—I have not entropy encoded each component of the instantiation. Completion of this aspect of the algorithm is necessary to verify the results. The algorithm also needs to be implemented on a microchip to verify the computational estimates, and it should be run on more data sets to verify generality.

7.2 RLE Entropy Reduction

The RLE delta encodings are not optimal. The non-optimality arises from the sharp dual peak in the probability distribution (see Figure 5). One technique to improve the optimality is to encode pairs of deltas. By using one codeword for a sequence of two deltas, the probability distribution will be flatter and thus more efficiently handled by Huffman coding. Alternatively, I could employ a technique such as arithmetic encoding, though this may increase the computational requirements of the algorithm.

I can further reduce the RLE *length* and RLE *delta* entropies by encoding with a technique that closely approximates the first-order entropy. For example, a *delta* of +1 is twice as likely to be followed by another *delta* of +1 than by a *delta* of –1; this fact is ignored by zero-order entropy encoding, but could be used by other methods. Table 7 compares the difference in entropy for RLE lengths and RLE deltas. By using first-order entropy I can achieve a better compression ratio (up to 4% better) with no degradation in reconstructed data quality.

	Zero-Order Entropy	First-Order Entropy
RLE lengths	3.38798	2.6302
RLE deltas	1.69207	1.48172

Table 7: Entropy of distribution of RLE deltas and bit-size distribution of RLE lengths

7.3 Template Instantiation Size Reduction

The 14 bit ***time*** component of the template instantiation contains information that is partially redundant with the non-spike run length encoding. Rather than store the time between spikes in the template instantiation, the non-spike run length encoding segment could incorporate a special symbol that means "*next spike goes here*". Rough estimations show that this technique could improve the overall compression ratio by approximately 5%. The compression could be further improved by requiring a template to begin at one of the 250Hz sample boundaries, raising the overall improvement to approximately 8%.

[2] Rather than scale each template to the size of the spike, the Dictionary could store all templates in a standard size, and the spike scaled to this standard size. This one time scaling is much more efficient.

Further reduction in the size of the template instantiation could occur by using less than $\log_2$(DictionarySize) bits for indexing the templates when the dictionary is not full. By more efficiently indexing into an unfilled dictionary, a larger dictionary could be used without concern about larger index bit sizes.

Dynamic Huffman coding [1] could be employed on the dictionary indices, further reducing the instantiation size. I estimate that a reduction of 1-2 bits per index (2-4 bits overall) could be achieved using such a scheme, an overall improvement of approximately 2%.

7.4 Template Parameterization Tradeoffs

By parameterizing the templates (for instance, by width, height, or vertical offset) and dividing them into smaller pieces (for instance, two segments divided at the peak) the algorithm allows fewer templates to match a wider variety of spikes, but at a cost of more storage space per template instance. There is a continuum of choices (see figure 8). At one end of the spectrum is no parameterization, where instancing a template means no more than giving its index. At the other extreme are algorithms such as SAPA, where there is only one "template", a straight line, that is instanced a multitude of times in a wide variety of angles, lengths, heights, and offsets. Neither end of figure 8 is optimal; no parameterization requires using too many templates, whereas SAPA does not take advantage of the similarity between spikes. I have tried to find an optimal position in the continuum, but optimality depends on many factors such as the similarity of spikes, the length of the recording, the error tolerance level τ, the qualities desired by the neurophysiologist, and the computational cost.

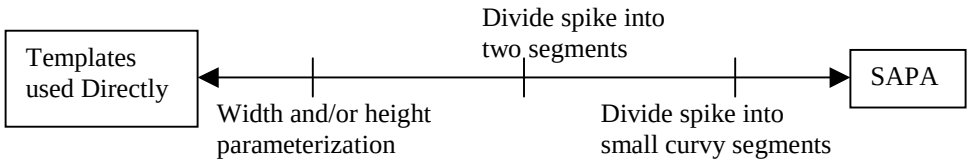


Figure 8: Continuum of template parameterizations

Height scaling reduces the template storage by approximately the same amount as it increases the template instancing, for a roughly zero net effect on compression ratio. However, it gives the neurophysiologist an experimental measurement (peak height) that is known rather than approximated.

Further exploration of this space can be done to identify the optimal parameterization factors.

7.5 Robust Spike Detection

Although my simple spike detection algorithm defined in section 3.2 successfully finds all spikes in both the *dev-data* and *eval-data*, it may not in future data. For example, if the spikes have more than one inflection point, falling to a small valley before rising to a peak, then my algorithm will fail. Furthermore, my spike detection algorithm occasionally labeled noise as spikes, particularly in sections of large 60Hz oscillation (due to power supply and inadequate grounding). 60Hz noise will not be a problem in the implanted system, but other sources of noise (animal motion, ground movement, etc.) may be introduced, and will have to be handled adequately.

7.6 Spike Extension

In many cases, a template would fit a spike better if the algorithm allowed the template to be widened to more than the identified spike width. Currently, my algorithm identifies a spike, determines the beginning and ending of the spike, and tries to find a template which fits the spike at the identified width. An algorithm which identifies a spike, and then finds a template which matches it *at any width*, may reduce the reconstruction error while also using less templates.

By further constraining template position and widths, I can achieve a reduction in the number of bits required to instance the template. For example, requiring the total width of the template to be a multiple of 8 samples would reduce the entropy of the instance width parameter by approximately 3 bits. I have found in preliminary experiments that such a constraint results in similar error but with an improvement in compression due to the reduction in instance size.

The areas near a spike change rapidly; Consequently, the run length encoding algorithm does not compress them well. Extending spikes to include more of the surrounding region results in a large improvement in run length encoding compression with minimal error degradation in spiking areas. Other algorithms such as SAPA for encoding the regions near spikes could be explored. A number of experiments could be performed on the effects of extending spikes or using SAPA in the near-spike areas. The results could decrease width entropies, improve RLE, and give better template fits.

7.7 Distance Function

An ideal distance function would be based on all of the neurophysiologist's qualitative preferences and would be constructed with full knowledge of what features are important and what features are not important. The sum of squares distance is an approximation designed to find templates that just "look similar". A better distance function may weight various aspects of the spike differently. For instance, the rising phase of the spike is more important than the falling phase [11]; a distance function should use knowledge such as this to make more informed decisions. Sum of squared differences also has the disadvantage that in areas of steep slope, a small horizontal offset results in a very large vertical difference and thus a very large squared distance score, yet visually looks almost as good. A possible distance function to correct this problem would be one that measures the distance from each point on the spike to the *closest* point on the template in any direction, rather than just measuring the vertical distance.

CONCLUSIONS

I have devised a technique for compressing intracellularly recorded neural signaling that attains a compression ratio exceeding 800:1 with visually and statistically acceptable error. The algorithm is computationally simple enough to be implemented in a small, power-efficient microchip. This work verifies that data storage and computational limitations will not impede the construction of a chip that intracellularly records from ten neural probes for over 24 hours. With further experimentation as discussed above, I believe the compression ratio can be improved by approximately 20-25% without any degradation in signal fidelity.

ACKNOWLEDGEMENTS

I would like to thank Russell Wyeth and Dennis Willows for their assistance with neurobiology and Udo Lang for the many discussions on neural electrodes and MEMS technology. I would also like to thank James Beck for supplying the neural data. This work was supported by a grant from the David and Lucille Packard Foundation.

REFERENCES

[1] Storer, James A. Data Compression: methods and theory. Computer Science Press, Inc, 1988.

[2] Welch, T.A., "A Technique For High Performance Data Compression," *IEEE Computer*, vol. 17, no. 6, pp.8–19, Jun. 1984.

[3] Jalaleddine, S.M.S., Hutchens, C.G., Strattan, R.D. and Coberly, W.A., "ECG Data Compression Techniques – A Unified Approach," *IEEE Trans. Biomed. Eng.*, vol. 37, no.4, pp.329–343, Apr. 1990.

[4] Ishijima, M., et al., "Scan-Along Polygonal Approximation for Data Compression of Electrocardiograms," *IEEE Trans. Biomed. Eng.*, vol. BME-30, no.11, pp723–729, Nov. 1983

[5] Kato, K. and Shunsuke, S., "Data Compression of a Gaussian Signal by TP Algorithm and Its Application to the ECG," *IEICE Trans. Inf. & Syst.*, vol. E76-D, no. 12, pp.1470–1478, Dec 1993.

[6] Uchiyama, T., Akazawa, K. and Sasamori, A., "Data Compression of Ambulatory ECG by Using Multi-Template Matching and Residual Coding," *IEICE Trans. Inf. & Syst.*, vol. E76-D, no. 12, pp.1419–1423, Dec 1993.

[7] Willows, A.O.D. Personal Correspondence. 3 May 1999.

[8] Willows, A.O.D., James Beck and Russell Wyeth. Personal Correspondence. 13–14 Apr. 1999.

[9] Takahashi, K., Takeuchi, S. and Ohsawa, N., "Performance Evaluation of ECG Compression Algorithms by Reconstruction Error and Diagnostic Response," *IEICE Trans. Inf. & Syst.*, vol. E76-D, no. 12, pp.1404–1408, Dec 1993.

[10] Ishijima, M., "Fundamentals of the Decision of Optimum Factors in the ECG Data Compression," *IEICE Trans. Inf. & Syst.*, vol. E76-D, no. 12, pp.1398–1403, Dec 1993.

[11] Willows, A.O.D. Personal Correspondence. 14 Apr. 1999.

[12] Willows, A.O.D. Personal Communication. 1999